Specification for the FIRRTL Language: Version 0.1.3 PRE-RELEASE VERSION - DO NOT DISTRIBUTE

Patrick S. Li
psli@eecs.berkeley.edu

Adam M. Izraelevitz
adamiz@eecs.berkeley.edu

Jonathan Bachrach
jrb@eecs.berkeley.edu

January 19, 2016

Contents

1 Introduction

1.1 Background

The ideas for FIRRTL originated from work on Chisel, a hardware description language embedded in Scala used for writing highly-parameterized circuit design generators. Chisel designers manipulate circuit components using Scala functions, encode their interfaces in Scala types, and use Scala's object-orientation features to write their own circuit libraries. This form of meta-programming enables expressive, reliable and type-safe generators that improve RTL design productivity and robustness.

The computer architecture research at U.C. Berkeley relied critically on Chisel to allow small teams of graduate students to design sophisticated RTL circuits. Over a three year period with under twelve graduate students, the architecture team has taped-out over ten different designs.

Internally, the investment in developing and learning Chisel was rewarded in huge gains in productivity. However, Chisel's external rate of adoption was slow for the following reasons:

1. Learning a functional programming language (Scala) is difficult for RTL designers with limited software-engineering experience
2. Confounding the previous point, conceptually separating the Chisel HDL from the host language is difficult for new users
3. The output of Chisel (Verilog) is unreadable and slow to simulate
4. Writing custom circuit transformers requires intimate knowledge about the internals of the Chisel compiler
5. Chisel semantics are ill-defined and thus impossible to target from other languages
6. Error checking is unprincipled due to ill-defined semantics resulting in incomprehensible error messages

As a consequence, Chisel needed to be redesigned from the ground up to standardize its semantics, modularize its compilation process for robustness, and cleanly separate its front-end, intermediate representation, and back-ends. A well defined intermediate representation (IR) allows the system to be targeted by other host programming languages, making it possible for RTL designers to work within a language they are already comfortable with. A clearly defined IR with a concrete syntax also allows inspection of the output of circuit generators/transformers thus making clear the distinction

between the host language and the constructed circuit. A clearly defined semantics allows users without knowledge of the compiler implementation to write circuit transformers. Examples include optimization of circuits for simulation speed, and automatic insertion of signal activity counters. An additional benefit of a well defined IR is the structural invariants that can be enforced before and after each compilation stage, resulting in a more robust compiler and structured mechanism for error checking.

1.2 Design Philosophy

FIRRTL (Flexible Intermediate Representation for RTL) represents the standardized elaborated circuit that the Chisel DSL produces. FIRRTL represents the circuit immediately after Chisel's elaboration but before any circuit simplification. It is designed to resemble the Chisel DSL after all meta-programming has executed. Thus a user program that makes little use of meta-programming facilities should look almost indentical to the generated FIRRTL.

For this reason, FIRRTL has first-class support for high-level constructs such as vector types, bundle types, when statements, partial connects, and modules. These high-level constructs are then gradually removed by a sequence of "lowering" transformations. During each lowering transformation the circuit is rewritten into an equivalent circuit using simpler, lower-level constructs. Eventually the circuit is simplified to its most restricted form, resembling a structured netlist, which allows for easy translation to an output language (e.g. Verilog). This form is given the name "lowered FIRRTL" (LoFIRRTL) and is a strict subset of the full FIRRTL language.

Because the host language is now used solely for its meta-programming facilities, the frontend can be very light-weight, and additional frontends in other languages can target FIRRTL and reuse the majority of the compiler toolchain.

Similar to backends, it is often convenient to write transformers that accept only the restricted LoFIRRTL subset. However, the transformed circuit is allowed to contain any FIRRTL construct, as it can be subsequently lowered again. We intentionally designed LoFIRRTL to be a subset of the full FIRRTL language to provide this feature.

2 Acknowledgements

The FIRRTL language could not have been developed without the help of many of the faculty and students in the ASPIRE lab, and the University of California, Berkeley.

This project originated from discussions with our advisor, Jonathan Bachrach, who indicated the need for a structural redesign of the Chisel system around a well-defined intermediate representation. Patrick Li designed and implemented the first prototype of the FIRRTL language, wrote the initial specification for the language, and presented it to the Chisel group consisting of Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, Donggyu Kim, Jack Koenig, Martin Maas, Albert Magyar, Colin Schmidt, Andrew Waterman, Yunsup Lee, Richard Lin, Eric Love, Albert Ou, Stephen Twigg, Jim Lawson, Brian Richards, Krste Asanovic, and John Wawrzynek.

Adam Izraelevitz then led the design and reimplemented FIRRTL, and after many discussions with the Chisel group, refined the design to its present version.

The authors would like to thank the following individuals for their contributions to the FIRRTL project:

1. Andrew Waterman: for his many contributions to the design of FIRRTL's constructs, for his work on Chisel 3.0, and for porting architecture research infrastructure

2. Richard Lin: for improving the Chisel 3.0 code base for release quality

3. Jack Koenig: for implementing the FIRRTL parser in Scala

4. Henry Cook: for porting and cleaning up many aspects of Chisel 3.0, including the testing infrastructure and the parameterization library

5. Stephen Twigg: for his expertise in hardware intermediate representations and for providing many corner cases to consider

6. Palmer Dabbelt, Eric Love, Martin Maas, Christopher Celio, and Scott Beamer: for their feedback on previous drafts of the FIRRTL specification

And finally this project would not have been possible without the continuous feedback and encouragement of our advisor, Jonathan Bachrach, and his leadership on and implementation of Chisel 3.0.

Research is partially funded by DARPA Award Number XXXX, the Center for Future Architectures Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

3 Circuits and Modules

3.1 Circuits

All FIRRTL circuits consist of a list of modules, each representing one hardware block that could be instantiated. The circuit must specify the name of the top-level module.

```
circuit MyTop :
   module MyTop :
      ...
   module MyModule :
      ...
```

All module names in a circuit exist in the same namespace, and thus all modules must have a unique name.

3.2 Modules

Each module has a given name, a list of ports, and a statement representing the circuit connections within the module. A module port is specified by its *direction* , which may be input or output, a name, and the data type for the port. The port names exist in the module identifier namespace, and must be unique.

```
module MyModule :
   input foo: UInt
```

```
   output bar: UInt
   bar <= foo
```

Note that a module definition does *not* indicate that the module will be physically present in the final circuit. Refer to the description of the instance statement for details on how to instantiate a module (section **??**).

3.3 Externally Defined Modules

Externally defined modules consist of a given name, and a list of ports, whose types and names must match its external definition.

```
module MyExternalModule :
   input foo: UInt
   output bar: UInt
   output baz: SInt
```

4 Types

4.1 Ground Types

All types in FIRRTL are either one of the fundamental ground types or are built up from aggregating other types. There are three ground types in FIRRTL, an unsigned integer type, a signed integer type, and a clock type.

4.1.1 Integer Types

Both unsigned and signed integer types may optionally be given a known positive integer bit width.

```
UInt<10>
SInt<32>
```

Alternatively, if the bit width is omitted, it will be automatically inferred by FIRRTL's width inferencer, as detailed in section **??**.

```
UInt
SInt
```

4.1.2 Clock Type

The clock type is used to describe wires and ports meant for carrying clock signals. The usage of components with clock types are restricted. Clock signals cannot be used in most primitive operations, and clock signals can only be connected to components that have been declared with the clock type.

The clock type is specified as follows:

```
Clock
```

4.2 Vector Types

A vector type is used to express an ordered sequence of elements of a given type. The length of the sequence must be non-negative and known. This is akin to array types in the C programming language.

The following example specifies a ten element vector of 16-bit unsigned integers.

```
UInt<16>[10]
```

The next example specifies a ten element vector of unsigned integers of omitted but identical bit widths.

```
UInt[10]
```

Note that any type, including other aggregate types, may be used as the element type in the sequence. The following example specifies a twenty element vector, each of which is a ten element vector of 16-bit unsigned integers.

```
UInt<16>[10][20]
```

4.3 Bundle Types

A bundle type is used to express a collection of nested and named types. All fields in a bundle type must have a given name, and type.

The following is an example of a possible type for representing a complex number. It has two fields, `real`, and `imag`, both 10-bit signed integers.

```
{real:SInt<10>, imag:SInt<10>}
```

Additionally, a field may optionally be declared with a *flipped* orientation.

```
{word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
```

In a connection between elements declared with the same bundle type, the data carried by the flipped fields flow in the opposite direction as the data carried by the non-flipped fields.

As an example, consider a module output port declared with the following type:

```
output mysignal: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
```

In a connection to the `mysignal` port, the data carried by the `word` and `valid` subfields will flow out of the module, while data carried by the `ready` subfield will flow into the module. More details about how the bundle field orientation affects connections are explained in section **??**.

As in the case of vector types, a bundle field may be declared with any type, including other aggregate types.

```
{real: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}
 imag: {word:UInt<32>, valid:UInt<1>, flip ready:UInt<1>}}
```

When calculating the final direction of data flow, the orientation of a field is applied recursively to all nested types in the field. As an example, consider the following module port declared with a bundle type containing a nested bundle type.

```
output myport: {a: UInt, flip b: {c: UInt, flip d:UInt}}
```

In a connection to `myport`, the `a` subfield flows out of the module. The `c` subfield contained in the `b` subfield flows into the module, and the `d` subfield contained in the `b` subfield flows out of the module.

Note that within a bundle, all field names must be unique.

4.4 Passive Types

Intuitively, a passive type is defined as a type where all leaf elements have a non-flipped orientation. Thus all ground types are passive types. Vector types are passive if their element type is passive. And bundle types are passive if no fields are flipped and if all field types are passive.

4.5 Type Equivalence

The type equivalence relation is used to determine whether a connection between two elements is legal. See section **??** for further details about connect statements.

An unsigned integer type is always equivalent to another unsigned integer type regardless of bit width, and is not equivalent to any other type. Similarly, a signed integer type is always equivalent to another signed integer type regardless of bit width, and is not equivalent to any other type.

Clock types are only equivalent to clock types, and are not equivalent to any other type.

Two vector types are equivalent if they have the same length, and if their element types are equivalent.

Two bundle types are equivalent when they have the same number of fields, and the i'th field from each bundle have matching names and orientations, as well as equivalent types. Consequently, `{a:UInt, b:UInt}` is a different type than `{b:UInt, a:UInt}`. `{a: {flip b:UInt}}` is a different type than `{flip a: {b: UInt}}`.

4.6 Weak Type Equivalence

The weak type equivalence relation is used to determine whether a partial connection between two elements is legal. See section **??** for further details about partial connect statements.

Two types are weakly equivalent if their corresponding oriented types are equivalent.

4.6.1 Oriented Types

The weak type equivalence relation requires first a definition of *oriented types*. Intuitively, an oriented type is a type where all orientation information is collated and coupled with the leaf ground types instead of in bundle fields.

An oriented ground type is an orientation coupled with a ground type. An oriented vector type is an ordered sequence of elements of a given oriented type. The length of an oriented vector type must be known and positive. An oriented bundle type is a collection of fields containing a name and an oriented type, but no orientation.

Applying a flip orientation to an oriented type recursively reverses the orientation of every oriented ground type contained within. Applying a non-flip orientation to an oriented type does nothing.

4.6.2 Conversion to Oriented Types

To convert a ground type to an oriented ground type, attach a non-flip orientation to the ground type.

To convert a vector type to an oriented vector type, convert its element type to an oriented type, and retain its length.

To convert a bundle field to an oriented bundle field, first convert its type to an oriented type. Then apply the field orientation to the oriented type, returning a new oriented type. The new oriented type, and the original field's name combine to create the oriented bundle field. To convert a bundle type to an oriented bundle type, convert each bundle field to an oriented bundle field.

4.6.3 Oriented Type Equivalence

Two oriented ground types are equivalent if their orientations and types are equivalent.

Two oriented vector types are equivalent if their element types are equivalent.

Two oriented bundle types are not equivalent if there exists two fields, one from each oriented bundle type, that have identical names but whose oriented types are not equivalent. Otherwise, the oriented bundle types are equivalent.

As stated earlier, two types are weakly equivalent if their corresponding oriented types are equivalent.

5 Statements

Statements are used to instantiate and connect circuit elements together.

5.1 The Connect Statement

The connect statement is used to specify a physically wired connection between two circuit elements.

The following example demonstrates connecting a module's input port to its output port, where port `myinput` is connected to port `myoutput`.

```
module MyModule :
   input myinput: UInt
   output myoutput: UInt
   myoutput <= myinput
```

In order for a connection to be legal the following conditions must hold:

1. The types of the left-hand and right-hand side expressions must be equivalent (see section **??** for details).
2. The bit widths of the two expressions must allow for data to always flow from a smaller to equal size or larger bit width.
3. The gender of the left-hand side expression must be female or bi-gender (see section **??** for an explanation of gender).
4. Either the gender of the right-hand side expression is male or bi-gender, or the right-hand side expression has a passive type.

Connect statements from a narrower ground type element to a wider ground type element will have its value automatically sign-extended to the larger bit width.

5.1.1 The Connection Algorithm

Connect statements between ground types cannot be expanded further.

Connect statements between two vector typed elements recursively connects each subelement in the right-hand side expression to the corresponding subelement in the left-hand side expression.

Connect statements between two bundle typed elements connects the i'th field of the right-hand side expression and the i'th field of the left-hand side expression. If the i'th field is not flipped, then the right-hand side field is connected to the left-hand side field. Conversely, if the i'th field is flipped, then the left-hand side field is connected to the right-hand side field.

5.2 The Partial Connect Statement

The partial connect statement is also used to specify a physically wired connection between two circuit elements. However, it enforces fewer restrictions on the types and widths of the circuit elements it connects.

In order for a partial connect to be legal the following conditions must hold:

1. The types of the left-hand and right-hand side expressions must be weakly equivalent (see section **??** for details).
2. The gender of the left-hand side expression must be female or bi-gender (see section **??** for an explanation of gender).
3. Either the gender of the right-hand side expression is male or bi-gender, or the right-hand side expression has a passive type.

Partial connect statements from a narrower ground type element to a wider ground type element will have its value automatically sign-extended to the larger bit width. Partial connect statements from a wider ground type element to a narrower ground type element will have its value automatically truncated to fit the smaller bit width.

Intuitively, bundle fields with matching names will be connected appropriately, while bundle fields not present in both types will be ignored. Similarly, vectors with mismatched lengths will be connected up to the shorter length, and the remaining elements are ignored.

The following example demonstrates partially connecting a module's input port to its output port, where port `myinput` is connected to port `myoutput`.

```
module MyModule :
   input myinput: {flip a:UInt, b:UInt[2]}
   output myoutput: {flip a:UInt, b:UInt[3], c:UInt}
   myoutput <- myinput
```

The above example is equivalent to the following:

```
module MyModule :
   input myinput: {flip a:UInt, b:UInt[2]}
   output myoutput: {flip a:UInt, b:UInt[3], c:UInt}
   myinput.a <- myoutput.a
   myoutput.b[0] <- myinput.b[0]
   myoutput.b[1] <- myinput.b[1]
```

For details on the syntax and semantics of the subfield and subindex expressions, see section **??**.

5.2.1 The Partial Connection Algorithm

A partial connect statement between two ground type elements connects the right-hand side expression to the left-hand side expression. Conversely, a reverse partial connect statement between two ground type elements connects the left-hand side expression to the right-hand side expression.

A partial (or reverse partial) connect statement between two vector typed elements applies a partial (or reverse partial) connect from the first n subelements in the right-hand side expression to the first n corresponding subelements in the left-hand side expression, where n is the length of the smaller vector.

A partial (or reverse partial) connect statement between two bundle typed elements considers any pair of fields, one from the first bundle type and one from the second, with matching names. If the first field in the pair is not flipped, then we apply a partial (or reverse partial) connect from the right-hand side field to the left-hand side field. However, if the first field is flipped, then we apply a reverse partial (or partial) connect from the right-hand side field to the left-hand side field.

5.3 Wire Declarations

A wire is a named combinational circuit element that can be connected to using the connect statement. A wire with a given name and type can be instantiated with the following statement.

wire name **:** *type*

Declared wires are *bidirectional*, which means that they can be used as both an input (by being on the left-hand side of a connect statement), or as an output (by being on the right-hand side of a connect statement).

5.4 Registers

A register is a named stateful circuit element. A register with a given name, type, clock reference, and reset reference, can be instantiated with the fol-

lowing statement.

reg name **:** *type*, *clk,reset*

Like wires, registers are also *bidirectional*, which means that they can be used as both an input (by being on the left-hand side of a connect statement), or as an output (by being on the right-hand side of a connect statement).

The onreset statement is used to specify the initialization value for a register, which is assigned to the register when the declared *reset* signal is asserted.

smem read from same address as write

5.5 Memories

A memory is a stateful circuit element containing multiple elements. The type for a memory must be completely specified; it cannot contain any unknown widths or bundle types with reverse fields. Unlike registers, memories can *only* be read from or written to through *accessors*, and cannot be initialized using a special FIRRTL construct Instead, the circuit itself must contain the proper logic to initialize the memory.

Additionally, if a memory is written via two or more accessors to the same memory address, the resulting stored value is undefined.

Memories always have a synchronous write, but can either be declared to be read combinatorially or synchronously. A combinatorially read memory with a given name, type, and size integer can be instantiated with the following statement.

cmem name **:** *type*, *size*

A synchronously read memory with a given name, type, and size integer can be instantiated with the following statement.

smem name **:** *type*, *size*

A synchronously read memory has the additional restriction that a read to an address on the same cycle its written returns an undefined value.

5.6 Poisons

A poison component is a named combinational circuit element that holds a random/garbage value. It cannot be connected to using the connect statement, and its type cannot contain a bundle with a flipped value.

A poison component with a given name and type can be instantiated with the following statement.

poison name **:** *type*

Declared poisons are *unidirectional*, which means that they can only be used as a source (being on the right-hand side of a connect statement).

5.7 Nodes

A node is simply a named intermediate value in a circuit, and is akin to a pointer in the C programming language. A node with a given name and value can be instantiated with the following statement.

node name = *exp*

Unlike wires, nodes can only be used in *output* directions. They can be connected from, but not connected to. Consequentially, their expression cannot be a bundle type with any reversed fields.

5.8 Accessors

Accessors are used for either connecting to or from a vector-typed expression, from some *variable* index.

dir **accessor** name = *exp*[index]*, clk*
dir = **infer**|**read**|**write**|**rdwr**

Given an accessor direction, a name, an expression to access, the index at which to access, and the clock domain it is in, the above statement creates an accessor that may be used for connecting to or from the expression. The expression must have a vector type, and the index must be a variable of UInt type.

A read, write, and inferred accessor is conceptually one-way; it must be consistently used to connect to, or to connect from, but not both.

A read-write accessor (**rdwr**) is conceptually two-way; it can be used to connect to, to connect from, or both, *but not on the same cycle*. If it is written to and read from on the same cycle, its behavior is undefined.

The following example demonstrates using accessors to read and write to a memory. The accessor, *reader*, acts as a memory read port that reads from

the index specified by the wire *i*. The accessor, *writer*, acts as a memory write port that writes 42 to the index specified by wire *j*.

wire i : **UInt**<5>
wire j : **UInt**<5>
cmem m : **UInt**<10>, 10
read accessor $reader = m[i], clk$
write accessor $writer = m[j], clk$
$writer$ <= **UInt**<**?**>(42)
node $temp = reader$

As mentioned previously, the only way to read from or write to a memory is through an accessor. However, accessors are not restricted to accessing memories. They can be used to access *any* cmem, smem, or wire/reg with vector-valued type.

An accessor passed a poisoned value as an index writes or reads a poisoned value.

5.9 Instances

An instance refers to a particular instantiation of a FIRRTL module. An instance is constructed with a given name and a given module name.

inst name **:** module

The resulting instance has a bundle type, where the given module's ports are fields and can be accessed using the subfield expression. The orientation of the *output* ports are *default*, and the orientation of the *input* ports are *reverse*. An instance may be directly connected to another element, but it must be on the right-hand side of the connect statement.

The following example illustrates directly connecting an instance to a wire:

```
extmodule Queue :
  input clk : Clock
  input in : UInt<16>
  output out : UInt<16>
module Top :
  input clk : Clock
  inst queue : Queue
  wire connect : {default out : UInt<16>, reverse in : UInt<16>, reverse clk : Clock}
  connect <= queue
```

The output ports of an instance may only be connected from, e.g., the right-hand side of a connect statement. Conversely, the input ports of an instance may only be connected to, e.g., the left-hand side of a connect statement.

The following example illustrates a proper use of creating instances with different clock domains:

```
extmodule AsyncQueue :
  input clk1 : Clock
  input clk2 : Clock
  input in : {default data : UInt<16>, reverse ready : UInt<1>}
  output out : {default data : UInt<16>, reverse ready : UInt<1>}
extmodule Source :
  input clk : Clock
  output packet : {default data : UInt<16>, reverse ready : UInt<1>}
extmodule Sink :
  input clk : Clock
  input packet : {default data : UInt<16>, reverse ready : UInt<1>}
module TwoClock :
  input clk1 : Clock
  input clk2 : Clock
  inst src : Source
  src.clk <= clk1
  inst snk : Sink
  snk.clk <= clk2
  inst queue : AsyncQueue
  queue.clk1 <= clk1
  queue.clk2 <= clk2
  queue.in <= src.packet
  snk.packet <= queue.out
```

There are restrictions upon which modules the user is allowed to instantiate, so as not to create infinitely recursive hardware. We define a module with no instances as a *level 0* module. A module containing only instances of *level 0* modules is a *level 1* module, and a module containing only instances of *level 1* or below modules is a *level 2* module. In general, a *level n* module is only allowed to contain instances of modules of level $n - 1$ or below.

5.10 The OnReset Connect Statement

The onreset connect statement is used to specify the default value for a **reg** element.

onreset r <= output

For a connection to be legal, the types of the two expressions must match exactly, including all field orientations if the elements contain bundle types. The component on the right-hand side must be able to be used as an output, and the component on the left-hand side must be a **reg** element. The widths of the types may mismatch, and the semantics are the same as the connect statements. Memories cannot be initialized with this construct.

By default, a **reg** will not have an initialization value and will maintain its current value under the reset signal specified in their declaration. The following example demonstrates declaring a **reg**, and changing its initialization value to forty-two.

reg r : **UInt**<10>(clk, $reset$)
onreset r <= **UInt**<**?**>(42)

5.11 The Partial Connect Statement

The partial connect statement is a connect statement that does not require both expressions to be the same type. During the lowering pass, the partial connect will expand to some number of connect statements, possibly zero statements. The following statement is used to connect the output of some component, to the input of another component.

input <− output

For a partial connect between two components of a bundle-type, fields that are of the same type, orientation, and name will be connected. Fields that do not match will not be connected. For a partial connect between two components of a vector-type, the number of connected elements will be equal to the length of the shorter vector. A partial connect between two components of the same ground type is equivalent to a normal connect statement. All other combinations of types will not error, but will not generate any connect statements.

5.12 The Conditional Statement

The conditional statement is used to specify a condition that must be asserted under which a list of statements hold. The condition must be a 1-bit unsigned integer. The following statement states that the *conseq* statements hold only when *condition* is assert high, otherwise the *alt* statements hold instead.

when condition **:** conseq **else :** alt

Notationally, for convenience, we omit the **else** branch if it is an empty statement.

5.12.1 Initialization Coverage

Because of the conditional statement, it is possible for wires to be only partially connected to an expression. In the following example, the wire w is connected to 42 when enable is asserted high, but it is not specified what w is connected to when enable is low. This is an illegal FIRRTL circuit, and will throw a **wire not initialized** error during compilation.

wire w : **UInt**<**?**>
when *enable* :
  w <= **UInt**<**?**>(42)

5.12.2 Scoping

The conditional statement creates a new *scope* within its consequent and alternative branches. It is an error to refer to any component declared within a branch after the branch has ended.

Note that there is still only a single identifier namespace in a module. Thus, there cannot be two components with identical names in the same module, *even if* they are in separate scopes. This is to facilitate writing transformational passes, by ensuring that the component name and module name is sufficient to uniquely identify a component.

5.12.3 Conditional Connect Semantics

Inside a when, a connection to a component is conditional only if the component is declared outside the when statement. If the component is both

declared and connected to inside a when, the connection is *not* conditional on that when.

Conceptually, a when creates a mux between the stuff outside and the stuff inside - it acts as type of "conditional barrier". Thus, if you draw a line between a component's declaration and a connection to it, that connection is dependent on all intersected when predicates being true.

The following example shows a *conditional* connection inside a when statement, where the register r is assigned the value of 42 only if *enable* is true.

reg r : **UInt**<**6**>
when *enable* :
  r <= **UInt**<**6**>(42)

The following shows an *unconditional* connection inside a when statement, where the register r is assigned the value of 42 *every cycle*.

when *enable* :
  reg r : **UInt**<**6**>
  r <= **UInt**<**6**>(42)

5.13 Statement Groups

Several statements can be grouped into one using the following construct.

(*stmt**)

Ordering is important in a statement group. Later connect statements take precedence over earlier connect statements, and circuit components cannot be referred to before they are instantiated.

5.13.1 Last Connect Semantics

Because of the connect statement, FIRRTL statements are *ordering* dependent. Later connections take precedence over earlier connections. In the following example, the wire w is connected to 42, not 20.

wire w : **UInt**<?>
w <= **UInt**<?>(20)
w <= **UInt**<?>(42)

By coupling the conditional statement with last connect semantics, many circuits can be expressed in a natural style. In the following example, the wire w is connected to 20 unless the enable expression is asserted high, in which case w is connected to 42.

wire w : **UInt**<**?**>
w <= **UInt**<**?**>(20)
when *enable* :
  w <= **UInt**<**?**>(42)

5.14 The Stop Statement

The stop statement is used to halt simulations of the circuit.

stop

If a backend does not support stop, it will omit emitting this node.

5.15 The Printf Statement

The printf statement is used to print a formatted string during simulations of the circuit.

printf (string, *exp**)

If a backend does not support printf, it will omit emitting this node.

5.16 The Empty Statement

The empty statement is specified using the following.

skip

The empty statement does nothing and is used simply as a placeholder where a statement is expected. It is typically used as the alternative branch in a conditional statement. In addition, it is useful for transformation pass writers.

6 Expressions

FIRRTL expressions are used for creating values corresponding to the ground types, for referring to a declared circuit component, for accessing a nested element within a component, and for performing primitive operations.

6.1 Unsigned Integers

A value of type **UInt** can be directly created using the following expression.

UInt<*width*>(value)

The given value must be non-negative, and the given width, if known, must be large enough to hold the value. If the width is specified as unknown, then FIRRTL infers the minimum possible width necessary to hold the value.

6.2 Signed Integers

A value of type **SInt** can be directly created using the following expression.

SInt<*width*>(value)

The given width, if known, must be large enough to hold the given value in two's complement format. If the width is specified as unknown, then FIRRTL infers the minimum possible width necessary to hold the value.

6.3 References

name

A reference is simply a name that refers to some declared circuit component. A reference may refer to a port, a node, a wire, a register, an instance, a memory, a node, or a structural register.

6.4 Subfields

exp.name

The subfield expression may be used for one of two purposes:

1. To refer to a specific port of an instance, using instance-name.port-name.
2. To refer to a specific field within a bundle-typed expression.

6.5 Subindex

$$exp[\text{index}]$$

The subindex expression is used for referring to a specific element within a vector-valued expression. It is legal to use the subindex expression on any vector-valued expression, except for memories.

6.6 Primitive Operation

$$primop(exp*, \text{int}*)$$

There are a number of different primitive operations supported by FIRRTL. Each operation takes some number of expressions, along with some number of integer literals. Section 7 will describe the format and semantics of each operation.

7 Primitive Operations

All primitive operations expression operands must be ground types. In addition, some operations allow all permutations of operand ground types, while others on allow subsets. When well defined, input arguments are allowed to be differing widths, with the semantics of sign-extending the input arguments prior to executing the operation.

7.1 Add Operation

Input Types	Resultant Type	Resultant Width
add$(op1 : UInt, op2 : UInt)$	$UInt$	$max(width(op1), width(op2)) + 1$
add$(op1 : UInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2)) + 1$
add$(op1 : SInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2)) + 1$
add$(op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2)) + 1$

The resultant's value is 1-bit larger than the wider of the two operands and has a signed type if either operand is signed (otherwise is unsigned).

7.2 Subtract Operation

primop	Resultant Type	Resultant Width
sub$(op1 : UInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2)) + 1$
sub$(op1 : UInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2)) + 1$
sub$(op1 : SInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2)) + 1$
sub$(op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2)) + 1$

The subtraction operation works similarly to the add operation, but always returns a signed integer with a width that is 1-bit wider than the max of the widths of the two operands.

7.3 Add Wrap Operation

primop	Resultant Type	Resultant Width
addw$(op1 : UInt, op2 : UInt)$	$UInt$	$max(width(op1), width(op2))$
addw$(op1 : UInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$
addw$(op1 : SInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2))$
addw$(op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$

The add wrap operation works identically to the normal add operation except that the resultant width is the maximum of the width of the two operands, instead of 1 bit greater than the maximum. In the case of overflow, the result silently rolls over.

7.4 Subtract Wrap Operation

primop	Resultant Type	Resultant Width
subw$(op1 : UInt, op2 : UInt)$	$UInt$	$max(width(op1), width(op2))$
subw$(op1 : UInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$
subw$(op1 : SInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2))$
subw$(op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$

Similarly to the add wrap operation, the subtract wrap operation works identically to the normal subtract operation except that the resultant width is the maximum of the width of the two operands. In the case of overflow, the result silently rolls over.

7.5 Multiply Operation

primop	Resultant Type	Resultant Width
mul($op1 : UInt, op2 : UInt$)	$UInt$	$width(op1) + width(op2)$
mul($op1 : UInt, op2 : SInt$)	$SInt$	$width(op1) + width(op2)$
mul($op1 : SInt, op2 : UInt$)	$SInt$	$width(op1) + width(op2)$
mul($op1 : SInt, op2 : SInt$)	$SInt$	$width(op1) + width(op2)$

The resultant value has width equal to the sum of the widths of its two operands.

7.6 Divide Operation

primop	Resultant Type	Resultant Width
div($op1 : UInt, op2 : UInt$)	$UInt$	$width(op1)$
div($op1 : UInt, op2 : SInt$)	$SInt$	$width(op1) + 1$
div($op1 : SInt, op2 : UInt$)	$SInt$	$width(op1)$
div($op1 : SInt, op2 : SInt$)	$SInt$	$width(op1) + 1$

The first argument is the dividend, the second argument is the divisor. The resultant width of a divide operation is equal to the width of the dividend, plus one if the divisor is an SInt. The resultant value follows the following formula : div(a,b) = round-towards-zero(a/b) + mod(a,b)

7.7 Modulus Operation

primop	Resultant Type	Resultant Width
mod($op1 : UInt, op2 : UInt$)	$UInt$	$width(op2)$
mod($op1 : UInt, op2 : SInt$)	$UInt$	$width(op2)$
mod($op1 : SInt, op2 : UInt$)	$SInt$	$width(op2) + 1$
mod($op1 : SInt, op2 : SInt$)	$SInt$	$width(op2)$

The first argument is the dividend, the second argument is the divisor. The resultant width of a modulus operation is equal to the width of the divisor, except when the modulus is positive and the result can be negative. The resultant value follows the following formula : div(a,b) = round-towards-zero(a/b) + mod(a,b)

7.8 Quotient Operation

primop	Resultant Type	Resultant Width
quo($op1 : UInt, op2 : UInt$)	$UInt$	$width(op1) + 1$
quo($op1 : UInt, op2 : SInt$)	$SInt$	$width(op1)$
quo($op1 : SInt, op2 : UInt$)	$SInt$	$width(op1) + 1$
quo($op1 : SInt, op2 : SInt$)	$SInt$	$width(op1)$

The first argument is the dividend, the second argument is the divisor. The resultant width of a quotient operation is equal to the width of the dividend, plus one if the divisor is an SInt. The resultant value follows the following formula : quo(a,b) = floor(a/b) + rem(a,b)

7.9 Remainder Operation

primop	Resultant Type	Resultant Width
rem($op1 : UInt, op2 : UInt$)	$UInt$	$width(op2)$
rem($op1 : UInt, op2 : SInt$)	$SInt$	$width(op2)$
rem($op1 : SInt, op2 : UInt$)	$UInt$	$width(op2) + 1$
rem($op1 : SInt, op2 : SInt$)	$SInt$	$width(op2)$

The first argument is the dividend, the second argument is the divisor. The resultant width of a modulus operation is equal to the width of the divisor, except when the divisor is positive and the result can be negative. The resultant value follows the following formula : quo(a,b) = floor(a/b) + rem(a,b)

7.10 Comparison Operations

primop	Resultant Type	Resultant Width
lt$(op1 : UInt, op2 : UInt)$	$UInt$	1
lt$(op1 : UInt, op2 : SInt)$	$UInt$	1
lt$(op1 : SInt, op2 : UInt)$	$UInt$	1
lt$(op1 : SInt, op2 : SInt)$	$UInt$	1
leq$(op1 : UInt, op2 : UInt)$	$UInt$	1
leq$(op1 : UInt, op2 : SInt)$	$UInt$	1
leq$(op1 : SInt, op2 : UInt)$	$UInt$	1
leq$(op1 : SInt, op2 : SInt)$	$UInt$	1
gt$(op1 : UInt, op2 : UInt)$	$UInt$	1
gt$(op1 : UInt, op2 : SInt)$	$UInt$	1
gt$(op1 : SInt, op2 : UInt)$	$UInt$	1
gt$(op1 : SInt, op2 : SInt)$	$UInt$	1
geq$(op1 : UInt, op2 : UInt)$	$UInt$	1
geq$(op1 : UInt, op2 : SInt)$	$UInt$	1
geq$(op1 : SInt, op2 : UInt)$	$UInt$	1
geq$(op1 : SInt, op2 : SInt)$	$UInt$	1

Each operation accepts any combination of SInt or UInt input arguments, and always returns a single-bit unsigned integer.

7.11 Equality Comparison

primop	Resultant Type	Resultant Width
eq$(op1 : UInt, op2 : UInt)$	$UInt$	1
eq$(op1 : UInt, op2 : SInt)$	$UInt$	1
eq$(op1 : SInt, op2 : UInt)$	$UInt$	1
eq$(op1 : SInt, op2 : SInt)$	$UInt$	1

The equality comparison operator accepts either two unsigned or signed integers and checks whether they are arithmetically equal. The resulting value is a 1-bit unsigned integer.

7.12 Not-Equality Comparison

primop	Resultant Type	Resultant Width
neq$(op1 : UInt, op2 : UInt)$	$UInt$	1
neq$(op1 : UInt, op2 : SInt)$	$UInt$	1
neq$(op1 : SInt, op2 : UInt)$	$UInt$	1
neq$(op1 : SInt, op2 : SInt)$	$UInt$	1

The not-equality comparison operator accepts either two unsigned or signed integers and checks whether they are arithmetically not equal. The resulting value is a 1-bit unsigned integer.

7.13 Equivalence Comparison

primop	Resultant Type	Resultant Width
eqv$(op1 : UInt, op2 : UInt)$	$UInt$	1
eqv$(op1 : SInt, op2 : SInt)$	$UInt$	1

The equivalence comparison operator accepts either two unsigned or two signed integers and checks whether they are bitwise equivalent. The resulting value is a 1-bit unsigned integer.

For an arithmetic equals between a signed and unsigned integer, use the equality operator.

7.14 Not-Equivalence Comparison

primop	Resultant Type	Resultant Width
neqv$(op1 : UInt, op2 : UInt)$	$UInt$	1
neqv$(op1 : SInt, op2 : SInt)$	$UInt$	1

The not-equivalent comparison operator accepts either two unsigned or two signed integers and checks whether they are not bitwise equivalent. The resulting value is a 1-bit unsigned integer.

If an arithmetic not-equals between a signed and unsigned integer is desired, use the not-equals operator.

7.15 Multiplex

primop	Resultant Type	Resultant Width
mux$(condition : UInt, op1 : UInt, op2 : UInt)$	$UInt$	$max(width(op1), width(op2))$
mux$(condition : UInt, op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$

The multiplex operation accepts three signals, a 1-bit unsigned integer for the condition expression, followed by either two unsigned integers, or two signed integers. If the condition is high, then the result is equal to the first of the two following operands. If the condition is low, then the result is the second of the two following operands.

The output is of the same width as the max width of the inputs.

7.16 Padding Operation

primop	Resultant Type	Resultant Width
pad$(op : UInt, \text{num})$	$UInt$	num
pad$(op : SInt, \text{num})$	$SInt$	num

A pad operation is provided which either zero-extends or sign-extends an expression to a specified width. The given width, num, must be equal to or greater than the existing width of the expression.

7.17 Reinterpret Bits as UInt

primop	Resultant Type	Resultant Width
asUInt$(op1 : UInt)$	$UInt$	$width(op1)$
asUInt$(op1 : SInt)$	$UInt$	$width(op1)$

Regardless of input type, primop returns a UInt with the same width as the operand.

7.18 Reinterpret Bits as SInt

primop	Resultant Type	Resultant Width
asSInt$(op1 : UInt)$	$SInt$	$width(op1)$
asSInt$(op1 : SInt)$	$SInt$	$width(op1)$

Regardless of input type, primop returns a SInt with the same width as the operand.

7.19 Shift Left Operation

primop	Resultant Type	Resultant Width
shl$(op : UInt, \text{num})$	$UInt$	$width(op) + num$
shl$(op : SInt, \text{num})$	$SInt$	$width(op) + num$

The shift left operation accepts either an unsigned or a signed integer, plus a non-negative integer literal specifying the number of bits to shift. The resultant value has the same type as the operand. The output of a shift left operation is equal to the original signal concatenated with n zeros at the end, where n is the shift amount.

7.20 Shift Right Operation

primop	Resultant Type	Resultant Width
shr$(op : UInt, \text{num})$	$UInt$	$width(op) - num$
shr$(op : SInt, \text{num})$	$SInt$	$width(op) - num$

The shift right operation accepts either an unsigned or a signed integer, plus a non-negative integer literal specifying the number of bits to shift. The resultant value has the same type as the operand. The shift amount must be less than or equal to the width of the operand. The output of a shift right operation is equal to the original signal with the least significant num bits truncated, where num is the shift amount.

7.21 Dynamic Shift Left Operation

primop	Resultant Type	Resultant Width
dshl$(op1 : UInt, op2 : UInt)$	$UInt$	$width(op1) + pow(2, width(op2))$
dshl$(op1 : SInt, op2 : UInt)$	$SInt$	$width(op1) + pow(2, width(op2))$

The dynamic shift left operation accepts either an unsigned or a signed integer, plus an unsigned integer dynamically specifying the number of bits to shift. The resultant value has the same type as the operand. The output of a dynamic shift left operation is equal to the original signal concatenated with n zeros at the end, where n is the dynamic shift amount. The output width of a dynamic shift left operation is the width of the original signal plus 2 raised to the width of the dynamic shift amount.

7.22 Dynamic Shift Right Operation

primop	Resultant Type	Resultant Width
dshr$(op1 : UInt, op2 : UInt)$	$UInt$	$width(op1)$
dshr$(op1 : SInt, op2 : UInt)$	$SInt$	$width(op1)$

The shift right operation accepts either an unsigned or a signed integer, plus a non-negative integer literal specifying the number of bits to shift. The resultant value has the same type as the operand. The shift amount must be less than or equal to the width of the operand. The output of a shift right operation is equal to the original signal with the least significant n bits truncated, where n is the dynamic shift amount. The output width of a dynamic shift right operation is the width of the original signal.

7.23 Logical Convert to Signed

primop	Resultant Type	Resultant Width
cvt$(op : UInt)$	$SInt$	$width(op) + 1$
cvt$(op : SInt)$	$SInt$	$width(op)$

The convert operation accepts either an unsigned or a signed integer. The resultant value is always a signed integer. The output of a convert operation will be the same arithmetic value as the input value. The output width is the same as the input width if the input is signed, and increased by one if the input is unsigned.

7.24 Negate

primop	Resultant Type	Resultant Width
neg$(op1 : UInt)$	$SInt$	$width(op1) + 1$
neg$(op1 : SInt)$	$SInt$	$width(op1)$

If the input type is UInt, primop returns the negative value as an SInt with the width of the operand plus one. If the input type is SInt, primop returns -1 * input value, as an SInt with the same width of the operand.

7.25 Bitwise Operations

primop	Resultant Type	Resultant Width
not$(op1{:}UInt)$	$UInt$	$width(op1)$
and$(op1{:}UInt, op2{:}UInt)$	$UInt$	$max(width(op1), width(op2))$
or$(op1{:}UInt, op2{:}UInt)$	$UInt$	$max(width(op1), width(op2))$
xor$(op1{:}UInt, op2{:}UInt)$	$UInt$	$max(width(op1), width(op2))$

The above operations correspond to bitwise not, and, or, and exclusive or respectively. The operands must be unsigned integers, and the resultant width is equal to the width of the wider of the two operands.

7.26 Reduce Bitwise Operations

primop	Resultant Type	Resultant Width
andr(*op:UInt*)	$UInt$	1
orr(*op:UInt*)	$UInt$	1
xorr(*op:UInt*)	$UInt$	1

The above operations correspond to bitwise not, and, or, and exclusive or respectively, reduced over every bit of a single unsigned integer. The resultant width is always one.

7.27 Concatenation

primop	Resultant Type	Resultant Width
cat($op1 : UInt, op2 : UInt$)	$UInt$	$width(op1) + width(op2)$

The concatenation operator accepts two unsigned integers and returns the bitwise concatenation of the two values as an unsigned integer. The resultant width is the sum of the widths of the two operands.

7.28 Bit Extraction Operation

primop	Resultant Type	Resultant Width
bit($op : UInt$, index)	$UInt$	1

The bit extraction operation accepts an unsigned integer, plus an integer literal specifying the index of the bit to extract. The resultant value is a 1-bit unsigned integer. The index must be non-negative and less than the width of the operand. An index of zero indicates the least significant bit in the operand, and an index of one less than the width the operand indicates the most significant bit in the operand.

7.29 Bit Range Extraction Operation

primop	Resultant Type	Resultant Width
bits($op : UInt$, high, low)	$UInt$	$high - low + 1$

The bit range extraction operation accepts either an unsigned integer, plus two integer literals that specify the high (inclusive) and low (inclusive) index of the bit range to extract. The index must be non-negative and less than the width of the operand. Regardless of the type of the operand, the resultant value is a n-bit unsigned integer, where $n = \text{high} - \text{low} + 1$.

8 FIRRTL Forms

To simplify the writing of transformation passes, any FIRRTL implementation will provide a *resolving* pass, which resolves all types, widths, and checks the legality of the circuit, and a *lowering* pass, which rewrites any FIRRTL circuit into an equivalent *lowered form*, or LoFIRRTL.

8.1 Resolved Form

The resolved form is guaranteed to be well-formed, meaning all restrictions to a FIRRTL circuit have been checked. In addition, all unknown widths and inferred accessor directions have been resolved.

8.2 Lowered Form

The lowered form, LoFIRRTL, is a structured subset of FIRRTL, making it a minimal representation that is convenient for low-level transforms.

The body of a lowered module consists of a list of declarations, connect statements, and *predicated single connect statements*. A predicated single connect statement is a conditional statement containing a single connect statement and no else branch.

The following circuit is lowered:

```
module MyCounter :
  clock clk : UInt<1>
  input reset : UInt<1>
  input inc : UInt<1>
  output out : UInt<3>
  reg counter : UInt<3>, clk, reset
  when inc : counter <= addw(counter, UInt<1>(1))
  out <= counter
```

The following restrictions also hold for modules in LoFIRRTL.

- **No Nested Expressions** : In the declaration of the structural elements, the only nested expressions allowed are references, and unsigned and signed literals. All other nested expressions must be lifted to a named node, and referred to through a reference.
- **No Composite Types** : No module port or wire may be declared with a bundle or vector type. The lowering pass will recursively expand ports into its constituent elements until all ports are declared with ground types.
- **Single Connect** : Every declared component can only be connected to once within a module. This connect could be a predicated single connect.
- **No Nested Whens** : Other than predicated single connect statements, no other conditional statements are allowed.
- **Inlined Lowered Form** : A further (and optional) pass provided by FIRRTL is the inlining pass, which recursively inlines all instances in the top-level module until the top-level module is the only remaining module in the circuit. Inlined LoFIRRTL is essentially a flat netlist which specifies every component used in a circuit and their input connections.

9 Annotations - IN PROGRESS

Supporting annotations is a critical piece of FIRRTL, yet is a very difficult problem to solve properly. We are in the experimental phase of supporting annotations, and our philosophy is outlined below. It remains to be seen whether our philosophy is correct - if not, we will certainly devise a new strategy.

1. Writing a correct circuit is difficult - avoid silent failures at all costs.
2. If annotations are held in the graph, every pass must properly propagate all possible annotations.
3. A pass incorrectly propagating an annotation cannot be easily detected (silent failure).

4. If annotations are held in an exteral data structure mapping names to annotations, the structure must updated after every pass.
5. Incorrectly updating the structure will cause a mismatching of names between circuit components and annotation entries, which is easily detected.
6. Thus, we feel the ability to detect failure outweighs the additional burden on annotation writers.

To implement this philosophy, we encourage passes to either preserve names in the graph, use simple algorithms to transform names, or provide a rename table after a pass. The annotation writer then updates their data structure accordingly.

10 Concrete Syntax

This section describes the text format for FIRRTL that is supported by the provided readers and writers.

General Principles

FIRRTL's text format is human-readable and uses indentation to indicate block structuring. The following characters are allowed in identifiers: upper and lower case letters, digits, as well as the punctuation characters `~!@#$%^*-_+=?/`. Identifiers cannot begin with a digit.

Comments begin with a semicolon and extend until the end of the line. Commas are treated as whitespace, and may be used by the user for clarity if desired.

Statements are grouped into statement groups using parenthesis, however a colon at the end of a line will automatically assume the next indented region is a statement group. This mechanism is used for indicating block structuring.

The following circuit, module, port and statement examples all exclude the info token `@[filename:line.col]`, which can be optionally included at the beginning of the first line of each elements' concrete syntax.

Circuits and Modules

A circuit is specified the following way.

```
circuit name : (modules ...)
```

Or by taking advantage of indentation structuring:

```
circuit name :
   modules ...
```

A module is specified the following way.

```
module name : (ports ... stmts ...)
```

The module body consists of a sequence of ports followed immediately by a sequence of statements. If there is more than one statement they are grouped into a statement group by the parser. By using indentation structuring:

```
module name :
   ports ...
   stmts ...
```

The following shows an example of a simple module.

```
module mymodule :
   input a: UInt<1>
   output b: UInt<1>
   clock clk: UInt<1>
   b <= a
```

Types

The unsigned and signed integer types are specified the following way. The following examples demonstrate an unsigned integer with known bit width, signed integer with known bit width, an unsigned integer with unknown bit width, and signed integer with unknown bit width.

```
UInt<42>
SInt<42>
UInt<?>
SInt<?>
```

The bundle type consists of a number of fields surrounded with braces. The following shows an example of a decoupled bundle type. Note that the commas are for clarity only and are not necessary.

```
{default data: UInt<10>,
 default valid: UInt<1>,
 reverse ready: UInt<1>}
```

The vector type is specified by immediately postfixing a type with a bracketed integer literal. The following example demonstrates a ten-element vector of 16-bit unsigned integers.

```
UInt<16>[10]
```

Statements

The following examples demonstrate declaring wires, registers, memories, nodes, instances, poisons, and accessors.

```
wire mywire : UInt<10>
reg myreg : UInt<10>, clk, reset
cmem mycombmem : UInt<10>,16
smem myseqmem : UInt<10>,16
inst myinst : MyModule
poison mypoison : UInt<10>
infer accessor myaccessor = e[i],clk
```

The connect statement is specified using the `<=` operator.

```
x <= y
```

The onreset connect statement is specified using the onreset keyword and the `<=` operator.

```
onreset x <= y
```

The partial connect statement is specified using the `<-` operator.

```
x <- y
```

The assert statement is specified using the assert keyword.

```
assert x
```

The conditional statement is specified with the `when` keyword.

```
when x : x <= y else : x <= z
```

Or by using indentation structuring:

```
when x :
   x <= y
else :
   x <= z
```

If there is no alternative branch specified, the parser will automatically insert an empty statement.

```
when x :
   x <= y
```

For convenience when expressing nested conditional statements, the colon following the `else` keyword may be elided if the next statement is another conditional statement.

```
when x :
   x <= y
else when y :
   x <= z
else :
   x <= w
```

Expressions

The UInt and SInt constructors create literal integers from a given value and bit width. The following examples demonstrate creating literal integers of both known and unknown bit width.

```
UInt<4>(42)
SInt<4>(-42)
UInt<?>(42)
SInt<?>(-42)
```

References are specified with an identifier.

```
x
```

Subfields are expressed using the dot operator.

```
x.data
```

Subindices are expressed using the `[]` operator.

```
x[10]
```

Primitive operations are expressed by following the name of the primitive with a list containing the operands.

```
add(x, y)
add(x, add(x, y))
shl(x, 42)
```

11 FIRRTL Language Definition

11.1 Abstract Syntax Tree

circuit	=	**circuit** id **:** (*module**)	Circuit
module	=	[*info*] **module** id **:** (*port* stmt*)	Module
	|	[*info*] **extmodule** id **:** (*port**)	External Module
port	=	[*info*] *dir* id **:** *type*	Port
dir	=	**input**|**output**	Port Direction
type	=	**UInt**<*width*>	Unsigned Integer
	|	**SInt**<*width*>	Signed Integer
	|	**Clock**	Clock
	|	{*field** }	Bundle
	|	*type*[int]	Vector
field	=	*orientation* id **:** *type*	Bundle Field
orientation	=	**default**|**reverse**	Orientation
width	=	int|**?**	Known/Unknown Integer Width
stmt	=	[*info*] **wire** id **:** *type*	Wire Declaration
	|	[*info*] **reg** id **:** *type*, *exp*, *exp*, *exp*	Register Declaration
	|	[*info*] **mem** id **:** *type*, int, int, int, (id*), (id*), (id*)	Memory Declaration
	|	[*info*] **inst** id **:** id	Instance Declaration
	|	[*info*] **poison** id **:** *type*	Poison Declaration
	|	[*info*] **node** id = *exp*	Node Declaration
	|	[*info*] *exp* **<=** *exp*	Connect
	|	[*info*] *exp* <− *exp*	Partial Connect
	|	[*info*] **when** *exp* **:** *stmt* **else :** *stmt*	Conditional
	|	[*info*] **stop** (*exp*, *exp*, int)	Stop Statement
	|	[*info*] **printf** (*exp*, *exp*, string, *exp**)	Printf Statement
	|	[*info*] **skip**	Empty Statement
	|	[*info*] (*stmt**)	Statement Group
exp	=	[*info*] **UInt**<*width*>(int)	Literal Unsigned Integer
	|	[*info*] **SInt**<*width*>(int)	Literal Signed Integer
	|	[*info*] id	Reference
	|	[*info*] *exp*.id	Subfield
	|	[*info*] *exp*[int]	Subindex
	|	[*info*] *exp*[*exp*]	Subaccess
	|	[*info*] *primop*(*exp**, int*)	Primitive Operation
info	=	filename **:** line . col	File Location
	|	**noinfo**	No File Location

primop	=	**add**	Unsigned/Signed Add
	|	**sub**	Unsigned/Signed Subtract
	|	**addw**	Unsigned/Signed Add Wrap
	|	**subw**	Unsigned/Signed Subtract Wrap
	|	**mul**	Unsigned/Signed Multiply
	|	**div**	Unsigned/Signed Divide
	|	**mod**	Unsigned/Signed Modulo
	|	**quo**	Unsigned/Signed Quotient
	|	**rem**	Unsigned/Signed Remainder
	|	**lt**	Unsigned/Signed Less Than
	|	**leq**	Unsigned/Signed Less or Equal
	|	**gt**	Unsigned/Signed Greater Than
	|	**geq**	Unsigned/Signed Greater or Equal
	|	**eq**	Unsigned/Signed Equal
	|	**neq**	Unsigned/Signed Not-Equal
	|	**mux**	Unsigned/Signed/Clock Multiplex
	|	**pad**	Unsigned/Signed Pad to Length
	|	**asUInt**	Unsigned/Signed Reinterpret Bits as UInt
	|	**asSInt**	Unsigned/Signed Reinterpret Bits as SInt
	|	**shl**	Unsigned/Signed Shift Left
	|	**shr**	Unsigned/Signed Shift Right
	|	**dshl**	Unsigned/Signed Dynamic Shift Left
	|	**dshr**	Unsigned/Signed Dynamic Shift Right
	|	**cvt**	Unsigned/Signed to Signed Logical Conversion
	|	**neg**	Unsigned/Signed Negate
	|	**not**	Unsigned Not
	|	**and**	Unsigned And
	|	**or**	Unsigned Or
	|	**xor**	Unsigned Xor
	|	**andr**	Unsigned And Reduce
	|	**orr**	Unsigned Or Reduce
	|	**xorr**	Unsigned Xor Reduce
	|	**cat**	Unsigned Concatenation
	|	**bit**	Single Bit Extraction
	|	**bits**	Multiple Bit Extraction
	|	**toClock**	Interpret Unsigned Bit as Clock
	|	**fromClock**	Interpret Clock as Unsigned Bit

11.2 Notation

The above definition specifies the structure of the abstract syntax tree corresponding to a FIRRTL circuit. Nodes in the abstract syntax tree are

italicized. Keywords are shown in **bold**. The special productions id, int, and string, indicates an identifier, an integer literal, and a string respectively. Tokens followed by an asterisk, *e.g. field**, indicates a list formed from repeated occurrences of the token.

Keep in the mind that the above definition is only the *abstract* syntax tree, and is a representation of the in-memory FIRRTL data structure. Readers and writers are provided for converting a FIRRTL data structure into a purely textual representation, which is defined in Section 10.

12 Future Plans

Some choices were made during the design of this specification which were intentionally conservative, so that future versions could lift the restrictions if suitable semantics and implementations are determined. By restricting this version and potentially lifting these restrictions in future versions, all existing FIRRTL circuits will remain valid.

The following design decisions could potentially be changed in future spec revisions:

1. Disallowing zero-width types
2. Always expanding memories into smaller memories (if its type is a non-ground-type)
3. Not including a **ROM** node
4. Custom annotations are not held in FIRRTL nodes
5. Not requiring that all names are unique

13 Questions and Answers

1. Why are there three connect operators? Each is needed for a particular use case - the better question is why did we chose to create multiple connect statements instead of other constructs. Statements, as opposed to expressions, are very restricted in how they nest. Thus, the desired supported behavior (partial connects, full connects, and resets) will never be used in an arbitrary nested expression where the semantics would be unintuitive. In addition, both the implementation and the user only needs to look at the single statement to implement it.

2. Aren't there a lot of idiosyncrasies in FIRRTL? The FIRRTL specification is an ongoing process, and as we push more code through it, it is likely to change. In our opinion, the idiosyncrasies are necessary for a cohesive design (and all languages have idiosyncrasies). It remains an unknown whether there are too many idiosyncrasies for frontend writers. Because the spec is not frozen, we can certainly adapt it if necessary. However, at this point, we just need to push more code through.
3. Why have a separate construct for initializing a register? The problem is initializing a register with a vector/bundle type, where a subset of the fields are initialized. If the initial value is kept with the declaration, we would need a new construct to specify a subset of values of ALL (potentially) nested vector/bundle types. It makes much more sense to separate initialization from the declaration, and use something like a ¡= to initialize the fields/vector sub-components of the register. The next question is why not just have users specify the initial value using their own "when reset :" statement. This doesn't work because of last connect semantics - the user could easily clobber their initialization when statement without knowing. Creating an onreset statement does two things: (1) specifies to the USER exactly what the reset value will be for any sub-component of a register, (2) encapsulates the reset value in a way that is easy for the implementation to special case it (so it doesn't get clobbered).
4. Why do operations allow inputs of differing widths? We tried restricting widths, but it actually complicated width inference and made supporting front-ends with more lax width restrictions very difficult. Because there is perfectly well defined semantics, we opted to allow differing widths. In line with the Linux "funnel" philosophy of being accepting with your inputs and restrictive with your outputs.
5. Why require all names unique? Passes usually need unique names, so there needs to be a renaming pass somewhere. Standardizing how names gets mangled requires a lot of thought, and we didn't feel comfortable putting this into the spec at the moment and potentially regretting it later. For now, names have to be unique, and it is the front-end's responsibility to do this.
6. Why allow declaring components in when statements? We want the important property that a module is just a box of components inside

- for any jumble of components, you can always lace them in the box, and it will preserve the semantics. You need to declare wires inside whens - because generators could run within a when in a front-end. You should always be able to pull them into a module if we want. Now its inconsistent if you can't declare registers in the scope.

7. Why not just have LoFIRRTL? LoFIRRTL leaves out general when usage, vector and bundle types, and requires a single connect. For performance backends, we will need to emit arrays and structs. If there is only a lowered circuit, we lose that ability. We cannot simply add vector/bundle types to LoFIRRTL as front-ends cannot easily remove whens without removing the complex types as well. Instead, one will need the expressiveness in FIRRTL to write a performant backend which does not need to operate on LoFIRRTL.
8. Why the stop statement have no arguements? Like the enable for write-accessors, the lowering step will preserve the sequence of when statements under which a simulation will stop.
9. Why disallow zero-width wires? Very tricky to get the semantics correct. On the todo list.
10. Why not require default value for wires? Isn't this a SAT problem? We do the same thing that is done in Java, and is standard programming language practice.
11. Why did/didn't you include XXX primop? Up for debate.
12. How do you support subword assignment? We decided to not support subword assignment directly, and instead require the user to separate the subword assignment into a vector type. Then, the user uses the subindex expression to assign to an element in the vector.