Specification for the FIRRTL Language

Version 4.1.0-14-gb0faeda

The FIRRTL Specification Contributors

February 27, 2025

Contents

1 Revision History

- 4.1.0-14-gb0faeda
  - Correct grammar for modulus primitive operator.
- 4.1.0
  - Add a second standard layer convention, “inline”.
- 4.0.0
  - Fix mistake in layer-associated probe grammar.
  - Define “Storable Type”.
  - Change the minimum width of the result of “Shift Right” on UInt to 0-bit.
  - Allow assert and assume statements to have a format string
  - Add Property primitive operations, starting with integer addition
  - Define/clarify layer-colored probe semantics.
  - Add Property primitive operation for integer multiplication
  - Add Property primitive operation for integer shift right
  - Drop “ref statement” support.
  - Correct mistakes in code examples.
  - Drop support for input probes.
  - Add “enablelayer” to grammar.
  - Main module must be public.
  - Make commas mandatory, not whitespace.
  - Restore flow description.
  - Update intrinsic module example to use a real intrinsic.
  - Restore id and info in printf grammar, add to verif commands.
  - Add intrinsic expressions and statements.
  - Remove intrinsic modules.
  - Allow layerbocks anywhere in a module.
  - Add language clarifying behavior statements affected by conditionals.
  - No abstract reset on externally-defined modules.
  - Add Property type and primitive operations for List.
  - Add `formal` unit test construct.
  - Add Property primitive operation for integer shift left
- 3.3.0
  - Add public modules.
  - Rename a “FIRRTL Language Definition” to “Grammar”.
  - Rename “Details about Syntax” to “Notes on Syntax”.
  - Add section “Circuit Components”.
  - Reorganized statements section.
  - Rewrite of the Types section.
- 3.2.0
  - Add optional groups.
  - Fix position of “Type Alias” (it used to be in the middle of “Reference Types”).
- 3.1.0
  - Add Integer property literals.
  - Add property assignment.

  - Add Integer property type.
  - Add initial description of property types.
  - Change/clarify mux selector width inference to align with other operations (must infer to some width by itself, pad if infers to less than 1-bit).
  - Fix printf grammar, expect commas between arguments.
  - Fix spec bug where string-encoded literals were still used in examples of "Constant Integer Expression".
  - Fix bug in grammar where int was incorrectly specified as being binary instead of decimal.
- 3.0.0
  - Add intrinsic modules to syntax highlighting
  - Add connect, invalidate to syntax highlighting
  - Add alternative `regreset` syntax
  - Add literal identifiers to allow for legal numeric fields
  - Simplify last-connect semantics explanation, remove "statement groups" (which are not part of the spec) which are only used in the original explanation
  - Add enumeration types, match statements, and enumeration expressions
  - Fixup probe endpoint and non-passive force examples.
  - Add type alias
  - Restrict string-encoded integers to only being usable in the construction of hardware literals.
  - Change string-encoded integers to radix-encoded integers.
  - Remove legacy connect (`<=`) and invalidate (`is invalid`) syntax
  - Make connect disallow implicit truncation (again).
- 2.4.0
  - Add radix-encoded integer literals as alternative syntax for string-encoded integer literals.
  - Add missing deprecation notice for "reg with" syntax.
- 2.3.0
  - Add intrinsic modules to syntax highlighting
  - Add connect, invalidate to syntax highlighting
  - Add alternative `regreset` syntax
- 2.2.0
  - Add 'asAsyncReset' to primop_1expr_keyword in "FIRRTL Language Definition"
  - Fix grammar for force_release statements
  - Add a description of conventions for modules
- 2.1.1
  - Fix typos in force/release examples, force takes expr not int literal.
  - Delineate string and single-quoted/double-quoted string in grammar.
  - Deprecate reference-first statements.
  - Tweak grammar of 'read' to support 'read(probe(x))' as in examples.
- 2.0.1
  - Clarify int/string types and their allowed usage.
- 2.0.0
  - Remove Fixed Point Types.

  - Remove conditionally valid expression (`validif`)
  - Remove partial connect ("<-")
  - Remove FIRRTL forms and lowering, indicate that high-level constructs may be preserved by a FIRRTL compiler
  - Add Compiler Implementation Details documenting Lower Types pass
  - Define constant type modifier.
  - Remove stray language leftover from removing conditionally valid.
  - Render inline annotations as JSON, fix typo in example.
  - Fix rendering of type modifiers (const) in document.
  - Fix grammar for registers.
  - Add reference types and related statements.
- 1.2.0
  - Specify behavior of zero bit width integers, add zero-width literals
  - Specify behavior of indeterminate values
  - Add an explicit section about "Aggregate Types" and move "Vector Type" and "Bundle Type" under it.
  - Move "head" and "tail" from primop_1expr_keyword to primop_1expr1int_keyword in the "FIRRTL Language Definition".
  - Add in-line annotation format
  - Specify behavior of combinational loops
  - Change connect to truncate widths to align with all existing FIRRTL Compiler implementations
  - Fix spelling/grammar issues
  - Allow out-of-bounds errors to be caught at compile time.
  - Clarify the string argument for cover is a comment, not a message as it is for assert and assume.
  - Add intrinsics.
  - Fix parameter grammar to include name of parameter.
- 1.1.0
  - Add version information to FIRRTL files
  - Specify "As-If" limited to boolean
- 1.0.0
  - Document the versioning scheme of this specification.
- 0.4.0
  - Add documentation for undocumented features of the Scala-based FIRRTL Compiler (SFC) that are de facto a part of the FIRRTL specification due to their widespread use in Chisel and the SFC: Annotations, Targets, Asynchronous Reset, Abstract Reset
  - Minor typo corrections and prose clarifications.
- 0.3.1
  - Clarify analog usage in registers
  - Rework authorship as "The FIRRTL Specification Contributors"
  - Add version information as subtitle
  - Formatting fixes
- 0.3.0

- Document moved to Markdown

2 Introduction

2.1 Background

The ideas for FIRRTL (Flexible Intermediate Representation for RTL) originated from work on Chisel, a hardware description language (HDL) 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 group at U.C. Berkeley relies 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 group has taped-out over ten different designs.

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

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

As a consequence, Chisel needed to be redesigned from the ground up to standardize its semantics, modularize its compilation process, and cleanly separate its front-end, intermediate representation, and backends. A well defined intermediate representation (IR) allows the system to be targeted by other HDLs embedded in 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 for inspection of the output of circuit generators and transformers thus making clear the distinction between the host language and the constructed circuit. 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.

2.2 Design Philosophy

FIRRTL represents the standardized elaborated circuit that the Chisel HDL produces. FIRRTL represents the circuit immediately after Chisel's elaboration. It is designed to resemble the Chisel HDL after all meta-programming has executed. Thus, a user program that makes little use of meta-programming facilities should look almost identical to the generated FIRRTL.

For this reason, FIRRTL has first-class support for high-level constructs such as vector types, bundle types, conditional statements, and modules. A FIRRTL compiler may choose to convert high-level constructs into low-level constructs before generating Verilog.

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

3 Acknowledgments

The FIRRTL specification was originally published as a UC Berkeley Tech Report (UCB/EECS-2016-9) authored by Adam Izraelevitz (@azidar), Patrick Li (@CuppoJava), and Jonathan Bachrach (@jackbackrack). The vision for FIRRTL was then expanded in an ICCAD paper and in Adam's thesis.

During that time and since, there have been a number of contributions and improvements to the specification. To better reflect the work of contributors after the original tech report, the FIRRTL specification was changed to be authored by *The FIRRTL Specification Contributors*. A list of these contributors is below:

- SpriteOvO
- albert-magyar
- azidar
- ben-marshall
- boqwxp
- chick
- dansvo
- darthscsi
- debs-sifive
- dobios
- donggyukim
- dtzSiFive
- eigenform
- ekiwi
- ekiwi-sifive

- fabianschuiki
- felixonmars
- grebe
- jackkoenig
- jared-barocsi
- keszocze
- maerhart
- mbty
- mikeurbach
- mmaloney-sf
- mwachs5
- nathsou
- prithayan
- richardxia
- rwy7
- seldridge
- sequencer
- shunshou
- smarter
- tdb-alcorn
- tymcauley
- uenoku
- youngar

4 File Preamble

A FIRRTL file begins with a magic string and version identifier indicating the version of this standard the file conforms to (see Section 29). This will not be present on files generated according to versions of this standard prior to the first versioned release of this standard to include this preamble.

```
FIRRTL version 2.0.0
circuit Foo :
```

5 Circuits and Modules

5.1 Circuits

A FIRRTL circuit is a named collection of FIRRTL modules. Each module is a hardware "unit" that has ports, registers, wires, and may instantiate other modules (see: Section 5.2). (This is the same concept as a Verilog `module`.)

Each FIRRTL circuit must have exactly one *main module*. The main module is a module that has the same name as the FIRRTL circuit. The main module must be public (see Section 5.2.1).

The following circuit contains a FIRRTL circuit with two modules. `Foo` is the main module:

```
circuit Foo :
  public module Foo :

  module Bar :
```

5.2 Modules

Each module has a given name, a list of ports, and a list of statements 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 of the port.

The following example declares a module with one input port, one output port, and one statement connecting the input port to the output port. See Section 8 for details on the connect statement.

```
  module MyModule :
    input foo: UInt<3>
    output bar: UInt<3>
    connect bar, foo
```

Refer to the description of the instance statement for details on how to instantiate a module (Section 6.1.5).

Modules may be either public or private.

5.2.1 Public Modules

A public module is marked with the **`public`** keyword. A public module has a stable interface allowing it to be instantiated by *other* circuits. A public module will always be physically present in a compiled circuit. In effect, public modules are the exported identifiers of a circuit.

The example below shows a public module `Foo`:

```
  public module Foo:
    input a: UInt<1>
    output b: UInt<1>

    connect b, a
```

A public module has a number of restrictions:

1. A public module may have no ports of uninferred width.
2. A public module may have no ports of abstract reset type.
3. A **`RWProbe`** may not be used to access a public module's ports.
4. A public module may be instantiated by other modules within a circuit, but the behavior of the module must not be affected by these instantiations.

For more information on the lowering of public modules, see the FIRRTL ABI Specification.

5.2.2 Private Modules

A private module is any module which is not marked with the `public` keyword. Private modules have none of the restrictions of public modules. Private modules have no stable, defined interface and may not be used outside the current circuit. A private module may not be physically present in a compiled circuit.

5.3 Externally Defined Modules

Externally defined modules are modules whose implementation is not provided in the current circuit. Only the ports and name of the externally defined module are specified in the circuit. An externally defined module may include, in order, an optional *defname* which sets the name of the external module in the resulting Verilog, zero or more name–value *parameter* statements. Each name–value parameter statement will result in a value being passed to the named parameter in the resulting Verilog.

The widths of all externally defined module ports must be specified. Width inference, described in Section 7.10.1, is not supported for externally defined module ports.

An externally-defined module must have no ports of or containing the abstract reset type.

A common use of an externally defined module is to represent a Verilog module that will be written separately and provided together with FIRRTL-generated Verilog to downstream tools.

An example of an externally defined module with parameters is:

```
extmodule MyExternalModule :
  input foo: UInt<2>
  output bar: UInt<4>
  output baz: SInt<8>
  defname = VerilogName
  parameter x = "hello"
  parameter y = 42
```

The types of parameters may be any of the following literal types. See Section 27.1 for more information:

1. Integer literal, e.g. `42`
2. String literal, e.g., `"hello"`
3. Raw String Literal, e.g., `'world'`

An integer literal is lowered to a Verilog literal. A string literal is lowered to a Verilog string. A raw string literal is lowered verbatim to Verilog.

As an example, consider the following external module:

```
extmodule Foo:
  parameter foo = '`hello'
```

```
    parameter bar = "world"
    parameter baz = 42
```

This is lowered to a Verilog instantiation site as:

```
Foo #(
  .a(`hello),
  .b("world"),
  .c(42)
) foo();
```

5.4 Layers

Layers are collections of functionality which will not be present in all executions of a circuit.

During execution, each layer is either enabled or disabled. When a layer is enabled, the FIRRTL circuit behaves *as-if* all the optional functionality of that layer was included in the normal execution of the circuit. When a layer is disabled, the circuit behaves *as-if* all the optional functionality of that layer was removed from the execution of the circuit. Layers are intended to be used to keep verification, debugging, or other collateral, not relevant to the operation of the circuit, in a separate area. Each layer can then be optionally included in the resulting design.

The `layer` keyword declares a layer with a specific identifier. A layer may be declared in a circuit or in another layer declaration. A layer's identifier must be unique within the current namespace. I.e., the identifier of a top-level layer declared in a circuit must not conflict with the identifier of a module, external module, or implementation defined module.

Layers may be nested, with the exceptions of convention constraints described below. Nested layers are declared with the `layer` keyword indented under an existing `layer` keyword.

Each layer must include a string that sets the *lowering convention* for that layer. The conventions defines both how the optional functionality is represented when the circuit is compiled to a backend language and the mechanism by which the layer's optional functionality can be enabled. For the purposes of FIRRTL execution, all layer conventions are the same—enabling the layer enables the optional functionality.

There are two standard lowering conventions: `bind` and `inline`. For details of the representation and mechanism of these conventions, see the FIRRTL ABI Specification. Compilers may define other non-standard lowering conventions.

A layer with the `bind` convention may not be nested, or transitively nested, under a layer of `inline` convention.

The example below shows a circuit with layers `A`, `B`, and `B.C`:

```
circuit Foo :
  layer A, bind :
  layer B, bind :
    layer C, bind :
```

Functionality enabled by a layer is put in one or more layer blocks inside modules. For more information on layer blocks see Section 17.

Probe types may be colored with a layer. Layer coloring indicates which layers must be enabled for a probe to be used. For more information on layer-colored probes see Section 7.3 and Section 18.2.

5.4.1 Modules with Enabled Layers

Modules may be declared with enabled layers. A module with enabled layers colors the body of the module with the color of all enabled layers. This affects the legality of operations which use probes. See Section 18.2 for more information on layer coloring.

To declare a module with layers enabled, use the **enablelayer** keyword. The circuit below shows a module with one layer enabled:

```
FIRRTL version 4.0.0
circuit Foo :
  layer A, bind :
  public module Foo enablelayer A :
```

5.5 Formal Unit Tests

The **formal** keyword declares a formal unit test. Tools may look for these constructs and automatically run them as part of a regression test suite.

A formal unit test has a unique name and refers to a module declaration that contains the design to be verified and any necessary test harness. Additionally, the test may also declare a list of user-defined named parameters which can be integers, strings, arrays, or dictionaries. These parameters are passed on to any tools that execute the tests. No parameters with well-known semantics are defined yet. During execution of the formal test, all input ports of the target module are treated as symbolic values that may change in every cycle. All output ports are ignored.

The circuit below shows a module run as a formal unit test:

```
FIRRTL version 4.0.0
circuit Foo :
  public module Foo :
  formal myTest of Foo :
```

A module may be reused in multiple unit tests, for example to run with different testing parameters:

```
FIRRTL version 4.0.0
circuit Foo :
  public module Foo :
  formal myTestA of Foo :
    mode = "bmc"
```

```
    bound = 42
  formal myTestB of Foo :
    mode = "induction"
    bound = 100
```

5.5.1 Test Harnesses

It may be necessary to define additional verification constructs and hardware in addition to the design that should be run in a formal unit test. To achieve this, an additional module may act as a test harness and instantiate the design to be tested alongside any other necessary verification collateral:

```
FIRRTL version 4.0.0
circuit Foo :
  public module Foo :
    input a : UInt<42>
    output b : UInt<42>
    ; ...

  public module FooTest :
    input clk : Clock
    input a : UInt<42>
    inst foo of Foo
    connect foo.a, a
    assert(clk, eq(foo.b, add(a, UInt(42))), UInt<1>(h1), "foo computes a + 42")
    assume(clk, gt(a, UInt(2)), UInt<1>(h1), "input a > 2")

  formal testFoo of FooTest :
```

6 Circuit Components

Circuit components are the named parts of a module corresponding to hardware.

There are seven **kinds** of circuit components. They are: nodes, wires, registers, output ports, input ports, submodule instances, and memories.

Circuit components can be connected together (see Section 8).

Each circuit component in a module has a type (Section 7). It is used to determine the legality of connections.

6.1 Kinds

6.1.1 Nodes

Nodes are named expressions in FIRRTL.

Example:

```
node mynode = and(in, UInt<4>(1))
```

The type of a node is the type of the expression given in the definition.

6.1.2 Wires

Wires represent named expressions whose value is determined by FIRRTL `connect` statements (see Section 8).

Example:

```
wire mywire : UInt<1>
connect mywire, UInt<1>(0)
```

Unlike nodes, the type of a wire must be explicitly declared. The type of a wire is given after the colon (`:`).

6.1.3 Registers

Registers are stateful elements of a design.

The state of a register is controlled through what is connected to it (see Section 8). The state may be any storable type (see Section 7.6). Registers are always associated with a clock. Optionally, registers may have a reset signal.

On every cycle, a register will drive its current value and then latch the value it will take on for the next cycle.

The `regreset` keyword is used to declare a register with a reset. The `reg` keyword is used to declare a register without a reset.

Examples:

```
wire myclock : Clock
reg myreg : SInt, myclock
```

```
wire myclock : Clock
reg myreg : SInt, myclock
```

```
wire myclock : Clock
wire myreset : UInt<1>
wire myinit : SInt
regreset myreg : SInt, myclock, myreset, myinit
```

For both variants of register, the type is given after the colon (`:`).

Semantically, registers become flip-flops in the design. The next value is latched on the positive edge of the clock. The initial value of a register is indeterminate (see Section 23.1).

6.1.4 Output Ports and Input Ports

The way a module interacts with the outside world is through its output and input ports.

Example:

```
input myinput : UInt<1>
output myoutput : SInt<8>
```

For both variants of port, the type is given after the colon (`:`).

The two kinds of ports differ in the rules for how they may be connected (see Section 8).

6.1.5 Submodule Instances

A module in FIRRTL may contain submodules.

Example:

```
inst passthrough of Passthrough
```

This assumes you have a `module` or `extmodule` named `Passthrough` declared elsewhere in the current circuit. The keyword `of` is used instead of a colon (`:`).

The type of a submodule instance is bundle type determined by its ports. Each port creates a field with the same name in the bundle. Among these fields, `output` ports are flipped, while `input` fields are unflipped.

For example:

```
public module Passthrough :
  input in : UInt<8>
  output out : UInt<8>
  connect out, in
```

The type of the submodule instance `passthrough` above is thus:

```
input a:
  { flip in : UInt<8>, out : UInt<8> }
```

6.1.6 Memories

Memories are stateful elements of a design.

Example:

```
mem mymem :
  data-type => { real:SInt<16>, imag:SInt<16> }
  depth => 256
  reader => r1
  reader => r2
  writer => w
  read-latency => 0
```

```
    write-latency => 1
    read-under-write => undefined
```

The type of a memory is a bundle type derived from the declaration.

The type named in `data-type` must be storable (see Section 7.6). It indicates the type of the data being stored inside of the memory.

6.2 Subcomponents

Each circuit component factors into subcomponents which can be accessed through references.

6.2.1 Examples

To motivate the notion of subcomponents, let's look at a few examples.

First, let's look at a wire with a vector type:

```
public module Foo :
  wire v : UInt<8>[3]
  connect v[0], UInt(0)
  connect v[1], UInt(10)
  connect v[2], UInt(42)
```

Here, we have declared a wire `v` with a vector type with length 3. We can index into the wire `v` with the expressions `v[0]`, `v[1]`, and `v[2]`, and target these with a `connect` statement (see Section 8). Each of these is a subcomponent of `v` and each acts like a wire with type `UInt<8>`.

Next, let's look at a port with a bundle type:

```
public module Bar :
  output io : { x : UInt<8>, flip y : UInt<8> }
  connect io.x, add(io.y, UInt(1))
```

The bundle of port `io` has type `{ x : UInt<8>, flip y : UInt<8> }`. It has two fields, `x` and `y`, and `y` is flipped. In the connect expression, we read from `io.y`, add 1 to it, and then assign it to `io.x`. Both `io.x` and `io.y` are subcomponents of `io` and both have type `UInt<8>`. Note that while `io.x` is an output port, `io.y` is an input port.

6.2.2 Definition

Every circuit component declared within a module results in a tree of **subcomponents**. Circuit subcomponents have both a kind and a type.

We define this tree of subcomponents recursively by defining the **direct subcomponent** relation.

A circuit component with a ground type, an enumeration type, a probe type, or property type (see Section 7) has no direct subcomponents. For example, `wire w : UInt<8>` has no

direct subcomponents.

When a circuit component has a vector type (see Section 7.2.1), it has as many direct subcomponents as its length. Each subcomponent has the same kind as its parent and has the element type. For example, if we declare `wire v : UInt<8>[3]`, it has three direct subcomponents: `v[0]`, `v[1]`, and `v[2]`. All three are wires and all three have type `UInt<8>`.

When a circuit component has a bundle type (see Section 7.2.2), it has one direct subcomponent for each field. The kind of the subcomponent depends on both the kind and the type of the parent:

- For nodes, wires, and registers, the kind of each direct subcomponent is the same.
- For output and input ports, the kind of each direct subcomponent depends on whether or not the field is flipped. When the field is not flipped, the kind remains the same. When the field is flipped, it changes from output to input or vice versa.
- For submodule instances and memories, the kind of each direct subcomponent depends on whether or not the field is flipped. When the field is not flipped, the kind is an input port. When the field is flipped, the kind is an output port.

If the bundle is not `const`, the type of each subcomponent is simply the type of the corresponding field. However, if the bundle is `const`, the type of each subcomponent is the `const` version of the type of the corresponding field.

A circuit component is a **subcomponent** of another if there is a way to get from the first component to the second through the direct subcomponent relation. A circuit component is trivially considered to be a subcomponent of itself. If we need to speak of the subcomponents excluding the component itself, we call these the **proper subcomponents**.

A **root component** is a circuit component that is not the direct subcomponent of any other component. Equivalently, these are the circuit components which are declared inside a module.

A **leaf component** is a circuit component that has no direct subcomponents. Leaf components are useful for checking initialization.

Two components are **disjoint** if they have no common subcomponent. For example, given `wire w : { x : UInt<1>, y : UInt<8>[2] }`, `w.x` and `w.y` are disjoint, but `w.y[0]` and `w.y` are not.

6.2.3 References

A **reference** is a name that refers to a previously declared circuit component. These are constructed using the names of root components, and optionally, through the dot and bracket operators.

For instance, suppose we declare `wire w : { x : UInt<1>, v : UInt<8>[2] }`. Then `w`, `w.x`, `w.y`, `w.y[0]`, and `w.y[1]`, and `w.y[i]` are all examples of references.

When a circuit component has a vector type, you can perform both static and dynamic indexing. For example, given `wire v : UInt<8>[2]`, both `v[0]` and `v[1]` are references

using static indexing. On the other hand, `v[i]` is a reference using dynamic indexing, given an expression `i` with an unsigned integer type.

The result of an out-of-bounds dynamic index is an indeterminate value (see Section 23.1).

A **static reference** is a reference where all indexing is static. For instances, `v[0]`, `v[1]`, and `v[2]` are static references. Static references are checked to ensure they are always in-bounds. A static reference identifies a concrete subcomponent within the module definition. Static references may be used in `probe` or `rwprobe` expressions (see Section 18).

A **dynamic reference** is one which is not a static reference.

References may appear as the target of `connect` statements (see Section 8). A connection to a dynamic reference is equivalent to a (potentially large) conditional statement consisting of only connects to static references.

7 Types

FIRRTL has four classes of types: **ground** types, **aggregate** types, **probe** types, and **property** types.

7.1 Ground Types

Ground types are types which are not composed of other types. They are the integer types, clocks, and resets.

7.1.1 Integer Types

FIRRTL supports both signed and unsigned integer types.

`UInt<n>` is an `n`-bit wide unsigned integer. `SInt<n>` is an `n`-bit wide signed integer.

```
UInt<10> ; a 10-bit unsigned integer
SInt<32> ; a 32-bit signed integer
```

Both `UInt<0>` and `SInt<0>` are valid types. They are considered to have only a single value representing zero (written as either `UInt<0>(0)` or `SInt<0>(0)`). They behave like zero when they are extended to a positive width integer type.

Integer types also have the inferred forms: `UInt` and `SInt` (see Section 7.10.1).

7.1.2 Clock Type

Clocks require special physical considerations in hardware. FIRRTL defines the `Clock` type to track clocks throughout a design. All registers are linked to a clock (see Section 6.1.3).

7.1.3 Reset Types

Once a circuit is powered on, it may require an explicit reset in order to put it into a known state. For this, we use a reset type. In FIRRTL, we have the option of using both synchronous or asynchronous resets.

The synchronous reset type is simply a 1-bit unsigned integer: `UInt<1>`. The asynchronous reset type is `AsyncReset`.

Registers may be declared linked to a reset (see Section 12.2).

The reset types also have the inferred form: `Reset` (see Section 7.10.2).

7.1.4 Analog Type

The analog type specifies that a wire or port can be attached to multiple drivers. `Analog` cannot be used as part of the type of a node or register, nor can it be used as part of the datatype of a memory. In this respect, it is similar to how `inout` ports are used in Verilog, and FIRRTL analog signals are often used to interface with external Verilog or VHDL IP.

In contrast with all other ground types, analog signals cannot appear on either side of a connection statement. Instead, analog signals are attached to each other with the commutative `attach` statement. An analog signal may appear in any number of attach statements, and a legal circuit may also contain analog signals that are never attached. The only primitive operations that may be applied to analog signals are casts to other signal types.

When an analog signal appears as a field of an aggregate type, the aggregate cannot appear in a standard connection statement.

As with integer types, an analog type can represent a multi-bit signal. When analog signals are not given a concrete width, their widths are inferred according to a highly restrictive width inference rule, which requires that the widths of all arguments to a given attach operation be identical.

```
Analog<1>  ; 1-bit analog type
Analog<32> ; 32-bit analog type
```

The analog type also has the inferred form: `Analog` (see Section 7.10.1).

7.2 Aggregate Types

FIRRTL supports three aggregate types: vectors, bundles, and enumerations.

7.2.1 Vector Types

A vector type is used to express an ordered sequence of elements of a given type.

The following example specifies a 10-element vector of 16-bit unsigned integers.

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

Note that the element type of a vector can be any type, including another aggregate type. The following example specifies a 20-element vector, each of which is a 10-element vector of 16-bit unsigned integers.

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

Vectors with length 0 are permitted. For details on how 0-length vectors are lowered, see the FIRRTL ABI Specification.

7.2.2 Bundle Types

A bundle type is used to represent a collection of values. They can also be used to facilitate bidirectional connections between circuit components.

A bundle type consists of zero or more fields. Each field has a name and a type. Fields may also be flipped.

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> }
```

The types of each field may be 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> } }
```

Here is an example of a bundle with a flipped field. Because the `ready` field is marked with the keyword `flip`, it will indicate the flow will be opposite of the `word` and `valid` fields.

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

As an example of how `flip` works in context, consider a module declared like this:

```
public module Processor :
  input enq : { word : UInt<32>, valid : UInt<1>, flip ready : UInt<1> }
  ; ...
```

This defines a module `Processor` with a single input port `enq` which enqueues data using a ready-valid interface. Because `enq` is a single port, we can connect to it with a single `connect` statement (see Section 8). In the final hardware, however, while the `word` and `valid` fields point *into* the module, this port has a flipped field `ready` which points *out of* the module instead.

7.2.3 Enumeration Types

Enumerations are disjoint union types.

Each enumeration consists of a set of variants. Each variant is named with a tag. Each variant also has a type associated with it which must be connectable (see Section 7.5) and passive (see Section 7.7).

In the following example, the first variant has the tag `a` with type `UInt<8>`, and the second variant has the tag `b` with type `UInt<16>`.

```
{|a : UInt<8>, b : UInt<16>|}
```

A variant may optionally omit the type, in which case it is implicitly defined to be `UInt<0>`. In the following example, all variants have the type `UInt<0>`.

```
{|a, b, c|}
```

7.3 Probe Types

Probe types expose and provide access to circuit components contained inside of a module. They are intended for verification. Ports with a probe type do not necessarily result in physical hardware. Special verification constructs enable the value of a probe to be read or forced remotely.

There are two probe types, `Probe<T>` is a read-only variant and `RWProbe<T>` is a read-write variant.

Examples:

```
Probe<UInt<8>>
RWProbe<UInt<8>>
```

`Probe` and `RWProbe` types may be *colored* with a layer (see Section 5.4). When *layer-colored*, there are restrictions placed on the use of the probe. See Section 18.2 for a description of these restrictions.

For example:

```
Probe<UInt<8>, A.B>      ; A.B is a layer
RWProbe<UInt<8>, A.B>
```

Probes are generally lowered to hierarchical names in Verilog. For details, see the FIRRTL ABI Specification.

For more information on probes, see Section 18.

7.4 Property Types

FIRRTL property types represent information about the circuit that is not hardware. This is useful to capture domain-specific knowledge and design intent alongside the hardware description within the same FIRRTL.

Property types cannot affect hardware functionality or the hardware ABI. They cannot be used in any hardware types, including aggregates and references. They only exist to augment the hardware description with extra information.

Handling of property types is completely implementation-defined. A valid FIRRTL compiler implementation may do anything with property types as long as the existence of property

types does not affect hardware functionality or the hardware ABI. For example, it is valid to drop property types from the IR completely.

Property types are legal in the following constructs:

- Port declarations on modules and external modules

7.4.1 Integer Type

The `Integer` type represents an numeric property. It can represent arbitrary-precision signed integer values.

```
public module Example:
  input intProp : Integer ; an input port of Integer property type
```

7.4.2 List Type

The `List` type represents an ordered sequence of properties. It is parameterized by the type of its elements, which can be any legal Property type.

```
public module Example:
  input listProp : List<Integer> ; an input port of List property type
```

7.5 Connectable Types

A **connectable type** is one which may be the type of expressions which may participate in the `connect` statement.

A connectable type is defined recursively:

- unsigned integers,
- signed integers,
- a vector type where the element type is a connectable type,
- bundles where each field type is a connectable type, or
- an enumeration type

7.6 Storable Types

Stateful elements, such as registers and memories, are restricted in what types they can store. A **storable type** is a type which may appear as the type of a register or the element type of a memory.

A storable type is defined recursively:

- All non-`const` integer types are storable.
- A non-`const` enumeration types are storable if and only if the type of each of its variants is storable.
- A non-`const` vector type is storable if and only if the element type is storable.

- A non-`const` bundle type is storable if and only if it contains no field which is marked `flip` and the type of each field is storable.

7.7 Passive Types

In certain contexts, the notion of `flip` does not make sense. A **passive type** is a type which does not make use of `flip`.

A passive type is defined recursively:

- All ground types are passive.
- All probe types are passive.
- All property types are passive.
- A vector type is passive if and only if the element type is passive.
- A bundle type is passive if and only if it contains no field which is marked `flip` and the type of each field is passive.
- All enumeration types are passive.

7.8 Constant Types

For certain situations, it is useful to guarantee that a signal holds a value that doesn't change during simulation. For example, when a register has a reset, the reset value is required to be held constant (see Section 12.2).

Ground types and aggregate types maybe marked as constant using the `const` modifier.

For example:

```
const UInt<3>
const SInt<8>[4]
const { real: UInt<32>, imag : UInt<32> }
```

All integer literals are `const`. For example, `UInt<8>(42)` has type `const UInt<8>`.

Ports can have a `const` type, and thus, a module may receive constant values from its parent module. This may even happen in `public` modules, and so the value of a `const` type need not be known statically (see Section 5.2.1).

Typically, primitive operations will result in a `const` type whenever each of its inputs are `const` (see Section 25). For example, `add(x, y)` will be `const` if both `x` and `y` are `const`.

The resulting type of a multiplexer expression, `mux(s, a, b)`, will be `const` if all of `s`, `a`, and `b` are `const` (see Section 19.4).

An expression of type `const T` is implicitly upcast to type `T` whenever it would be required to make a primitive operation, mux expression, or connect statement typecheck.

Expressions with `const` may be used as the target of a connect statement as long as the following hold:

- the source of the connect is `const`

- the conditions of all containing **when** blocks the connect statement is nested in must have conditions of type **const UInt**<1>
- the subject of any containing **match** blocks the connect statement is nested in must have a **const** type

Constant types may not be the type of a stateful circuit components. Thus, registers are memories may not be declared with a **const** type.

References to a subcomponent of a circuit component with a **const** vector or bundle type results in a **const** of the inner type.

For example:

```
input c : const { real : SInt<8>, imag : SInt<8> }
; c.real has type const SInt<8>
```

7.9 Type Alias

Type aliases allow us to give names to types. This is useful for bundle types, especially when they have many fields. It is also useful for hinting at what the value represents.

Examples:

```
type WordType = UInt<32>
type ValidType = UInt<1>
type Data = { w : WordType, valid : ValidType, flip ready : UInt<1> }
type AnotherWordType = UInt<32>
public module TypeAliasMod:
  input in : Data
  output out : Data
  wire w : AnotherWordType
  connect w, in.w
  ; ...
```

Type aliases have structural identity. In other words, when we compare two types, we expand all type aliases recursively until we are left with type expressions that have no aliases. For instance, in the above example, we can connect `in.w` to `w` since their types, `WordType` and `AnotherWordType` respectively, both expand to **UInt**<32>.

The **type** declaration is globally defined and all named types exist in the same namespace and thus must all have a unique name. Type aliases do not share the same namespace as modules; hence it is allowed for type aliases to conflict with module names.

7.10 Type Inference

FIRRTL has support for limited type inference. This comes in two flavors: Width inference and reset inference.

7.10.1 Width Inference

Normally, the three integer ground types are written with explicit bit widths. The bit widths are given in angle brackets, such as the `8` in `UInt<8>`. This is called the **uninferred** variant of the type.

FIRRTL also supports an **inferred** variant of these types. They are written as follows:

```
UInt
SInt
Analog
```

When an inferred variant of an integer ground type is used, FIRRTL will calculate the minimum width needed for it. It is an error if the minimum size cannot be calculated.

For example, the width of a multiplexer expression is the maximum of its two corresponding input widths.

The width of each primitive operation is detailed in Section 25.

7.10.2 Reset Inference

The uninferred `Reset` type will be inferred to either a synchronous reset `UInt<1>` or to an asynchronous reset `AsyncReset`.

The following example shows an inferred reset that will get inferred to a synchronous reset.

```
input a : UInt<1>
wire reset : Reset
connect reset, a
```

After reset inference, `reset` is inferred to the synchronous `UInt<1>` type:

```
input a : UInt<1>
wire reset : UInt<1>
connect reset, a
```

The following example demonstrates usage of an asynchronous reset.

```
input clock : Clock
input reset : AsyncReset
input x : UInt<8>
regreset y : UInt<8>, clock, reset, UInt(123)
; ...
```

Inference rules are as follows:

1. An uninferred reset driven by and/or driving only asynchronous resets will be inferred as asynchronous reset.
2. An uninferred reset driven by and/or driving both asynchronous and synchronous resets is an error.

3. Otherwise, the reset is inferred as synchronous (i.e. the uninferred reset is only invalidated or is driven by or drives only synchronous resets).

`Reset`s, whether synchronous or asynchronous, can be cast to other types. Casting between reset types is also legal:

```
input a : UInt<1>
output y : AsyncReset
output z : UInt<1>
wire r : Reset
connect r, a
connect y, asAsyncReset(r)
connect z, asUInt(y)
```

See Section 25 for more details on casting.

7.10.3 Probes and Type Inference

Given a `Probe<T>` or `RWProbe<T>`, the inner type `T` may be an inferred type (see Section 7.3). The inner type `T` will be inferred, however, they must do so in a way which does not affect the hardware.

A module with a probe with an inferred inner type must resolve all other inferred types to the same uninferred types as it would if the probe were to be removed.

Additionally, inference constraints may only flow in few restricted ways:

- Inference constraints may flow from child to parent through its `output` ports.
- Inference constraints may flow from parent to child through its `input` ports.
- In a `define` expression, only the right-hand side may impose inference constraints on the left-hand side.
- Neither `read` expressions nor `force` statements may impose inference constraints.

8 Connections

8.1 Flow

The direction that signals travel across wires is determined by multiple factors: the kind of circuit component (e.g., `input` vs `output`), the side of a connect statement it appears on, and the presence of `flip`s if the signal is a bundle type.

To ensure connections are meaningful when taking directionality into account, every expression in FIRRTL has a **flow**. The flow of an expression can be one of **source**, **sink**, or **duplex**.

A source expression supplies a signal and can be used to drive a circuit component. A sink expression can be driven by another expression. A duplex expression is an expression that is both a source and sink.

The rules for the flow of an expression are as follows.

If the expression is an identifier, we look at the kind of the circuit component the identifier refers to:

- Nodes are sources.
- Wires and registers are duplex.
- For ports, `input` ports are sources and `output` ports are duplex.
- Submodule instances are sources.
- Memories are sources.

The flow of a sub-index or sub-access expression is the flow of the vector-typed expression it indexes or accesses. The flow of a sub-field expression depends upon the orientation of the field. If the field is not flipped, its flow is the same flow as the bundle-typed expression it selects its field from. If the field is flipped, then its flow is the reverse of the flow of the bundle-typed expression it selects its field from. The reverse of source is sink, and vice-versa. The reverse of duplex remains duplex.

The flow of all other expressions are source, including mux and cast.

8.2 Type Equivalence

The type equivalence relation is used to determine whether a connection between two components is legal.

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

An uninferred `Reset` can be connected to another `Reset`, `UInt` of unknown width, `UInt<1>`, or `AsyncReset` (see Section 7.10.2). It cannot be connected to both a `UInt` and an `AsyncReset`.

The `AsyncReset` type can be connected to another `AsyncReset` or to a `Reset`.

Two enumeration types are equivalent if both have the same number of variants, and both the enumerations' i'th variants have matching names and equivalent types.

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

Two bundle types are equivalent if they have the same number of fields, and both the bundles' i'th fields have matching names and orientations, as well as equivalent types. Consequently, `{a:UInt, b:UInt}` is not equivalent to `{b:UInt, a:UInt}`, and `{a: {flip b:UInt}}` is not equivalent to `{flip a: {b: UInt}}`.

Two property types are equivalent if they are the same concrete property type.

8.3 The Connect Statement

The components of a module can be connected together using `connect` statements.

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

```
public module MyModule :
  input myinput: UInt<2>
  output myoutput: UInt<2>
  connect 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 8.2 for details).
2. The flow of the left-hand side expression must be sink or duplex (see Section 8.1 for an explanation of flow).
3. Either the flow of the right-hand side expression is source or duplex, or the right-hand side expression has a passive type.
4. The left-hand side and right-hand side types are not property types.

Connect statements from a narrower ground type component to a wider ground type component will have its value automatically sign-extended or zero-extended to the larger bit width. The behavior of connect statements between two circuit components with aggregate types is defined by the connection algorithm in Section 8.3.1.

8.3.1 The Connection Algorithm

Connect statements between ground types cannot be expanded further.

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

Connect statements between two bundle typed components 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.

8.3.2 Last Connect Semantics

Ordering of connects is significant. Later connects take precedence over earlier ones. In the following example port `b` will be connected to `myport1`, and port `a` will be connected to `myport2`:

```
public module MyModule :
  input a: UInt<5>
```

```
    input b: UInt<5>
    output myport1: UInt<5>
    output myport2: UInt<5>

    connect myport1, a
    connect myport1, b
    connect myport2, a
```

Conditional statements are affected by last connect semantics. For details see Section 13.5.

When a connection to a component with an aggregate type is followed by a connection to a sub-element of that same component, only the connection to the sub-element is overwritten. Connections to the other sub-elements remain unaffected. In the following example the `c` sub-element of port `portx` will be connected to the `c` sub-element of `myport`, and port `porty` will be connected to the `b` sub-element of `myport`.

```
public module MyModule :
  input portx: {b: UInt<1>, c: UInt<2>}
  input porty: UInt<1>
  output myport: {b: UInt<1>, c: UInt<2>}
  connect myport, portx
  connect myport.b, porty
```

The above circuit can be rewritten as:

```
public module MyModule :
  input portx: {b: UInt<1>, c: UInt<2>}
  input porty: UInt<2>
  output myport: {b: UInt<1>, c: UInt<2>}
  connect myport.b, porty
  connect myport.c, portx.c
```

When a connection to a sub-element of an aggregate component is followed by a connection to the entire circuit component, the later connection overwrites the earlier sub-element connection.

```
public module MyModule :
  input portx: {b: UInt<1>, c: UInt<2>}
  input porty: UInt<2>
  output myport: {b: UInt<1>, c: UInt<2>}
  connect myport.b, porty
  connect myport, portx
```

The above circuit can be rewritten as:

```
public module MyModule :
  input portx: {b: UInt<1>, c: UInt<2>}
  input porty: UInt<1>
  output myport: {b: UInt<1>, c: UInt<2>}
```

```
connect myport, portx
```

See Section 6.2.3 for more details about indexing.

8.4 Invalidates

The `invalidate` statement allows a circuit component to be left uninitialized or only partially initialized. The uninitialized part is left with an indeterminate value (see Section 23.1).

It is specified as follows:

```
wire w: UInt
invalidate w
```

The following example demonstrates the effect of invalidating a variety of circuit components with aggregate types. See Section 8.4.1 for details on the algorithm for determining what is invalidated.

```
public module MyModule :
  input in: {flip a: UInt<1>, b: UInt<2>}
  output out: {flip a: UInt<1>, b: UInt<2>}
  wire w: {flip a: UInt<1>, b: UInt<2>}
  invalidate in
  invalidate out
  invalidate w
```

is equivalent to the following:

```
public module MyModule :
  input in: {flip a: UInt<1>, b: UInt<2>}
  output out: {flip a: UInt<1>, b: UInt<2>}
  wire w: {flip a: UInt<1>, b: UInt<2>}
  invalidate in.a
  invalidate out.b
  invalidate w.a
  invalidate w.b
```

The handing of invalidated components is covered in Section 23.1.

8.4.1 The Invalidate Algorithm

Invalidating a component with a ground type indicates that the component's value is undetermined if the component has sink or duplex flow (see Section 8.1). Otherwise, the component is unaffected.

Invalidating a component with a vector type recursively invalidates each sub-element in the vector.

Invalidating a component with a bundle type recursively invalidates each sub-element in the bundle.

Components of reference and analog type are ignored, as are any reference or analog types within the component (as they cannot be connected to).

8.5 Combinational Loops

Combinational logic is a section of logic with no registers between gates. A combinational loop exists when the output of some combinational logic is fed back into the input of that combinational logic with no intervening register. FIRRTL does not support combinational loops even if it is possible to show that the loop does not exist under actual mux select values. Combinational loops are not allowed and designs should not depend on any FIRRTL transformation to remove or break such combinational loops.

The module `Foo` has a combinational loop and is not legal, even though the loop will be removed by last connect semantics.

```
public module Foo:
  input a: UInt<1>
  output b: UInt<1>
  connect b, b
  connect b, a
```

The following module `Foo2` has a combinational loop, even if it can be proved that `n1` and `n2` never overlap.

```
public module Foo2 :
  input n1: UInt<2>
  input n2: UInt<2>
  wire tmp: UInt<1>
  wire vec: UInt<1>[3]
  connect tmp, vec[n1]
  connect vec[n2], tmp
```

Module `Foo3` is another example of an illegal combinational loop, even if it only exists at the word level and not at the bit-level.

```
public module Foo3:
  wire a : UInt<2>
  wire b : UInt<1>
  wire c : UInt<1>

  connect a, cat(b, c)
  connect b, bits(a, 0, 0)
```

9 Attaches

The `attach` statement is used to attach two or more analog signals, defining that their values be the same in a commutative fashion that lacks the directionality of a regular connection. It

can only be applied to signals with analog type, and each analog signal may be attached zero or more times.

```
wire x: Analog<2>
wire y: Analog<2>
wire z: Analog<2>
attach(x, y)      ; binary attach
attach(z, y, x)   ; attach all three signals
```

10 Property Assignments

Connections between property typed expressions (see Section 7.4) are not supported in the **connect** statement (see Section 8).

Instead, property typed expressions are assigned with the **propassign** statement.

Property typed expressions have the normal rules for flow (see Section 8.1), but otherwise use a stricter, simpler algorithm than **connect**. In order for a property assignment to be legal, the following conditions must hold:

1. The left-hand and right-hand side expressions must be of property types.
2. The types of the left-hand and right-hand side expressions must be the same.
3. The flow of the left-hand side expression must be sink.
4. The flow of the right-hand side expression must be source.
5. The left-hand side expression may be used as the left-hand side in at most one property assignment.
6. The property assignment must not occur within a conditional scope.

Note that property types are not legal for any expressions with duplex flow.

The following example demonstrates a property assignment from a module's input property type port to its output property type port.

```
public module Example:
  input propIn : Integer
  output propOut : Integer
  propassign propOut, propIn
```

The following example demonstrates a property assignment from a property literal expression to a module's output property type port.

```
public module Example:
  output propOut : Integer
  propassign propOut, Integer(42)
```

11 Empty Statement

The empty statement does nothing and is used simply as a placeholder where a statement is expected. It is specified using the **skip** keyword.

The following example:

```
connect a, b
skip
connect c, d
```

can be equivalently expressed as:

```
connect a, b
connect c, d
```

The empty statement is most often used as the **else** branch in a conditional statement, or as a convenient placeholder for removed components during transformational passes. See Section 13 for details on the conditional statement.

12 Registers

A register is a stateful circuit component. Reads from a register return the current value of the element, writes are not visible until after the next positive edge of the register's clock.

The clock signal for a register must be of type **Clock**. The type of a register must be a storable type (see Section 7.6).

Registers may be declared without a reset using the **reg** syntax and with a reset using the **regreset** syntax.

12.1 Registers without Reset

The following example demonstrates instantiating a register with the given name **myreg**, type **SInt**, and is driven by the clock signal **myclock**.

```
wire myclock: Clock
reg myreg: SInt, myclock
; ...
```

12.2 Registers with Reset

A register with a reset is declared using **regreset**. A **regreset** adds two expressions after the type and clock arguments: a reset signal and a reset value. The register's value is updated with the reset value when the reset is asserted. The reset signal must be a **Reset**, **UInt**<1>, or **AsyncReset**, and the type of initialization value must be equivalent to the declared type of the register (see Section 8.2 for details). If the reset signal is an **AsyncReset**, then the reset value must be a constant type. The behavior of the register depends on the type of

the reset signal. `AsyncReset` will immediately change the value of the register. `UInt<1>` will not change the value of the register until the next positive edge of the clock signal (see Section 7.1.3). `Reset` is an abstract reset whose behavior depends on reset inference. In the following example, `myreg` is assigned the value `myinit` when the signal `myreset` is high.

```
wire myclock: Clock
wire myreset: UInt<1>
wire myinit: SInt
regreset myreg: SInt, myclock, myreset, myinit
; ...
```

A register is initialized with an indeterminate value (see Section 23.1).

13 Conditionals

Conditional statements define one or more blocks, each with an associated condition. These blocks contain other statements (including nested conditional statements). The behavior of the contained statements is affected by the conditions associated to the blocks containing them.

13.1 Conditional Statements

FIRRTL provides several kinds of conditional statements.

13.1.1 When Statements

When statements define a condition and two conditional blocks: a "then" block and an "else" block. The condition must be a 1-bit unsigned integer type. Statements within the "then" block are conditionally executed when the condition is true. Statements within the "else" block are conditionally executed when the condition is false. Statements within blocks are executed using the rules in Section 13.2.

In the following example, the wire `x` is connected to the input `a` only when the `en` signal is high. Otherwise, the wire `x` is connected to the input `b`.

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input en: UInt<1>
  wire x: UInt
  when en :
    connect x, a
  else :
    connect x, b
```

13.1.1.1 Syntactic Shorthands

The `else` branch of a conditional statement may be omitted, in which case a default `else` branch is supplied consisting of the empty statement.

Thus the following example:

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input en: UInt<1>
  wire x: UInt
  when en :
    connect x, a
```

can be equivalently expressed as:

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input en: UInt<1>
  wire x: UInt
  when en :
    connect x, a
  else :
    skip
```

To aid readability of long chains of conditional statements, the colon following the `else` keyword may be omitted if the `else` branch consists of a single conditional statement.

Thus the following example:

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input c: UInt<3>
  input d: UInt<3>
  input c1: UInt<1>
  input c2: UInt<1>
  input c3: UInt<1>
  wire x: UInt
  when c1 :
    connect x, a
  else :
    when c2 :
      connect x, b
    else :
      when c3 :
        connect x, c
      else :
```

```
        connect x, d
```

can be equivalently written as:

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input c: UInt<3>
  input d: UInt<3>
  input c1: UInt<1>
  input c2: UInt<1>
  input c3: UInt<1>
  wire x: UInt
  when c1 :
    connect x, a
  else when c2 :
    connect x, b
  else when c3 :
    connect x, c
  else :
    connect x, d
```

To additionally aid readability, a conditional statement where the contents of the **when** branch consist of a single line may be combined into a single line. If an **else** branch exists, then the **else** keyword must be included on the same line.

The following statement:

```
when c :
  connect a, b
else :
  connect e, f
```

can have the **when** keyword, the **when** branch, and the **else** keyword expressed as a single line:

```
when c : connect a, b else :
  connect e, f
```

The **else** branch may also be added to the single line:

```
when c : connect a, b else : connect e, f
```

13.1.2 Match Statements

Match statements define blocks whose statements are executed when the value of an enumeration typed expression is equal to an enumeration variant. Statements within blocks are executed using the rules in Section 13.2. A match statement must exhaustively test every

variant of an enumeration. An optional binder may be specified to extract the data of the variant.

```
match x:
  some(v):
    connect a, v
  none:
    connect e, f
```

13.2 Conditional Execution

Statements that appear in a conditional block behave differently based on the type of statement and on the conditions of the blocks containing it.

For connections (see Section 8), appearing inside of a conditional block affects whether or not the connect executes. The sink of a connect operator will correspond to a circuit component declared earlier in the module. We consider the conditions associated to the blocks of all conditional blocks which contain the connect statement, but which do *not* contain the declaration of the circuit component. We call this set of conditions the **interleaving conditions** of the connect. Whenever we determine the last connect semantics (see Section 8.3.2) for that component, if each of the interleaving conditions is true, then that connect is executed.

For commands (see Section 16), we instead consider the conditions associated with *each* containing conditional block.

For hardware component declarations, conditional blocks have no effect.

These semantics may cause different behavior for trivial inlining (substitution of an instantiation's module body in place of the instantiation). A register in a conditional block and an instantiation in a conditional block where the instantiated module contains a register will execute the same after a trivial inlining. A command in a conditional block and a module instantiation in a conditional block where the instantiated module contains a command will not execute the same after a trivial inlining. In this latter case, the command is not conditional before trivial inlining and conditional after trivial inlining.

13.3 Initialization Coverage

Because of the conditional statement, it is possible to syntactically express circuits containing wires that have not been connected to under all conditions.

In the following example, the wire `a` is connected to the wire `w` when `en` is high, but it is not specified what is connected to `w` when `en` is low.

```
public module MyModule :
input en: UInt<1>
input a: UInt<3>
```

```
wire w: UInt
when en :
  connect w, a
```

This is an illegal FIRRTL circuit and an error will be thrown during compilation. All wires, memory ports, instance ports, and module ports that can be connected to must be connected to under all conditions. Registers do not need to be connected to under all conditions, as it will keep its previous value if unconnected.

13.4 Declarations within Conditional Blocks

The names of circuit components declared within conditional blocks must be unique within their module's namespace (see Section 20). Circuit components declared within condition blocks may only be referred to within that block.

The behavior is also a bit non-intuitive at first. This differs from the behavior in most programming languages, where variables in a local scope can shadow variables declared outside that scope. Additionally, while the names exist *in the module's namespace, those names cannot be* used *outside of the block which they are declared.*

13.5 Conditional Last Connect Semantics

In the case where a connection to a circuit component is followed by a conditional statement containing a connection to the same component, the connection is overwritten only when the condition holds. Intuitively, a multiplexer is generated such that when the condition is low, the multiplexer returns the old value, and otherwise returns the new value. For details about the multiplexer, see Section 19.4.

The following example:

```
wire a: UInt
wire b: UInt
wire c: UInt<1>
wire w: UInt
connect w, a
when c :
  connect w, b
```

can be rewritten equivalently using a multiplexer as follows:

```
wire a: UInt
wire b: UInt
wire c: UInt<1>
wire w: UInt
connect w, mux(c, b, a)
```

Because invalid statements assign indeterminate values to components, a FIRRTL Compiler

is free to choose any specific value for an indeterminate value when resolving last connect semantics. E.g., in the following circuit `w` has an indeterminate value when `c` is false.

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
invalidate w
when c :
  connect w, a
```

A FIRRTL compiler is free to optimize this to the following circuit by assuming that `w` takes on the value of `a` when `c` is false.

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
  connect w, a
```

See Section 23.1 for more information on indeterminate values.

The behavior of conditional connections to circuit components with aggregate types can be modeled by first expanding each connect into individual connect statements on its ground elements (see Section 8.3.1 for the connection algorithm) and then applying the conditional last connect semantics.

For example, the following snippet:

```
wire x: {a: UInt, b: UInt}
wire y: {a: UInt, b: UInt}
wire c: UInt<1>
wire w: {a: UInt, b: UInt}
connect w, x
when c :
  connect w, y
```

can be rewritten equivalently as follows:

```
wire x: {a:UInt, b:UInt}
wire y: {a:UInt, b:UInt}
wire c: UInt<1>
wire w: {a:UInt, b:UInt}
connect w.a, mux(c, y.a, x.a)
connect w.b, mux(c, y.b, x.b)
```

Similar to the behavior of aggregate types under last connect semantics (see Section 8.3.2), the conditional connects to a sub-element of an aggregate component only generates a multiplexer for the sub-element that is overwritten.

For example, the following snippet:

```
wire x: {a: UInt, b: UInt}
wire y: UInt
wire c: UInt<1>
wire w: {a: UInt, b: UInt}
connect w, x
when c :
  connect w.a, y
```

can be rewritten equivalently as follows:

```
wire x: {a: UInt, b: UInt}
wire y: UInt
wire c: UInt<1>
wire w: {a: UInt, b: UInt}
connect w.a, mux(c, y, x.a)
connect w.b, x.b
```

14 Memory Instances

A memory is an abstract representation of a hardware memory. It is characterized by the following parameters.

1. A storable type representing the type of each element in the memory (see Section 7.6).
2. A positive integer literal representing the number of elements in the memory.
3. A variable number of named ports, each being a read port, a write port, or readwrite port.
4. A non-negative integer literal indicating the read latency, which is the number of cycles after setting the port's read address before the corresponding element's value can be read from the port's data field.
5. A positive integer literal indicating the write latency, which is the number of cycles after setting the port's write address and data before the corresponding element within the memory holds the new value.
6. A read-under-write flag indicating the behavior when a memory location is written to while a read to that location is in progress.

Integer literals for the number of elements and the read/write latencies *may not be radix-encoded integer literals*.

The following example demonstrates instantiating a memory containing 256 complex numbers, each with 16-bit signed integer fields for its real and imaginary components. It has two read ports, `r1` and `r2`, and one write port, `w`. It is combinationally read (read latency is zero cycles) and has a write latency of one cycle. Finally, its read-under-write behavior is undefined.

```
mem mymem :
  data-type => {real:SInt<16>, imag:SInt<16>}
  depth => 256
  reader => r1
  reader => r2
  writer => w
  read-latency => 0
  write-latency => 1
  read-under-write => undefined
```

In the example above, the type of `mymem` is:

```
{flip r1: {addr: UInt<8>,
           en: UInt<1>,
           clk: Clock,
           flip data: {real: SInt<16>, imag: SInt<16>}},
 flip r2: {addr: UInt<8>,
           en: UInt<1>,
           clk: Clock,
           flip data: {real: SInt<16>, imag: SInt<16>}},
 flip w:  {addr: UInt<8>,
           en: UInt<1>,
           clk: Clock,
           data: {real: SInt<16>, imag: SInt<16>},
           mask: {real: UInt<1>, imag: UInt<1>}}}
```

The following sections describe how a memory's field types are calculated and the behavior of each type of memory port.

14.1 Read Ports

If a memory is declared with element type `T`, has a size less than or equal to 2^N, then its read ports have type:

```
{addr: UInt<N>, en: UInt<1>, clk: Clock, flip data: T}
```

If the `en` field is high, then the element value associated with the address in the `addr` field can be retrieved by reading from the `data` field after the appropriate read latency. If the `en` field is low, then the value in the `data` field, after the appropriate read latency, is undefined. The port is driven by the clock signal in the `clk` field.

14.2 Write Ports

If a memory is declared with element type `T`, has a size less than or equal to 2^N, then its write ports have type:

```
{addr: UInt<N>, en: UInt<1>, clk: Clock, data: T, mask: M}
```

where M is the mask type calculated from the element type T. Intuitively, the mask type mirrors the aggregate structure of the element type except with all ground types replaced with a single bit unsigned integer type. The *non-masked portion* of the data value is defined as the set of data value leaf sub-elements where the corresponding mask leaf sub-element is high.

If the en field is high, then the non-masked portion of the data field value is written, after the appropriate write latency, to the location indicated by the addr field. If the en field is low, then no value is written after the appropriate write latency. The port is driven by the clock signal in the clk field.

14.3 Readwrite Ports

Finally, the readwrite ports have type:

```
{addr: UInt<N>, en: UInt<1>, clk: Clock, flip rdata: T, wmode: UInt<1>,
 wdata: T, wmask: M}
```

A readwrite port is a single port that, on a given cycle, can be used either as a read or a write port. If the readwrite port is not in write mode (the wmode field is low), then the rdata, addr, en, and clk fields constitute its read port fields, and should be used accordingly. If the readwrite port is in write mode (the wmode field is high), then the wdata, wmask, addr, en, and clk fields constitute its write port fields, and should be used accordingly.

14.4 Read Under Write Behavior

The read-under-write flag indicates the value held on a read port's data field if its memory location is written to while it is reading. The flag may take on three settings: old, new, and undefined.

If the read-under-write flag is set to old, then a read port always returns the value existing in the memory on the same cycle that the read was requested.

Assuming that a combinational read always returns the value stored in the memory (no write forwarding), then intuitively, this is modeled as a combinational read from the memory that is then delayed by the appropriate read latency.

If the read-under-write flag is set to new, then a read port always returns the value existing in the memory on the same cycle that the read was made available. Intuitively, this is modeled as a combinational read from the memory after delaying the read address by the appropriate read latency.

If the read-under-write flag is set to undefined, then the value held by the read port after the appropriate read latency is undefined.

For the purpose of defining such collisions, an "active write port" is a write port or a readwrite port that is used to initiate a write operation on a given clock edge, where en is set and, for a readwriter, wmode is set. An "active read port" is a read port or a readwrite port that is used to initiate a read operation on a given clock edge, where en is set and, for a

readwriter, `wmode` is not set. Each operation is defined to be "active" for the number of cycles set by its corresponding latency, starting from the cycle where its inputs were provided to its associated port. Note that this excludes combinational reads, which are simply modeled as combinationally selecting from stored values.

For memories with independently clocked ports, a collision between a read operation and a write operation with independent clocks is defined to occur when the address of an active write port and the address of an active read port are the same for overlapping clock periods, or when any portion of a read operation overlaps part of a write operation with a matching addresses. In such cases, the data that is read out of the read port is undefined.

14.5 Write Under Write Behavior

In all cases, if a memory location is written to by more than one port on the same cycle, the stored value is undefined.

14.6 Constant memory type

A memory with a constant data-type represents a ROM and may not have write-ports. It is beyond the scope of this specification how ROMs are initialized.

15 Submodule Instances

FIRRTL modules are instantiated with the instance statement. The following example demonstrates creating an instance named `myinstance` of the `MyModule` module within the top level module `Top`.

```
circuit Top :
  module MyModule :
    input a: UInt
    output b: UInt
    connect b, a
  public module Top :
    inst myinstance of MyModule
```

The resulting instance has a bundle type. Each port of the instantiated module is represented by a field in the bundle with the same name and type as the port. The fields corresponding to input ports are flipped to indicate their data flows in the opposite direction as the output ports. The `myinstance` instance in the example above has type `{flip a:UInt, b:UInt}`.

Modules have the property that instances can always be *inlined* into the parent module without affecting the semantics of the circuit.

To disallow infinitely recursive hardware, modules cannot contain instances of itself, either directly, or indirectly through instances of other modules it instantiates.

16 Commands

16.1 Stops

The stop statement is used to halt simulations of the circuit. Backends are free to generate hardware to stop a running circuit for the purpose of debugging, but this is not required by the FIRRTL specification.

A stop statement requires a clock signal, a halt condition signal that has a single bit unsigned integer type, and an integer exit code.

For clocked statements that have side effects in the environment (stop, print, and verification statements), the order of execution of any such statements that are triggered on the same clock edge is determined by their syntactic order in the enclosing module. The order of execution of clocked, side-effect-having statements in different modules or with different clocks that trigger concurrently is undefined.

The stop statement has an optional name attribute which can be used to attach metadata to the statement. The name is part of the module level namespace. However it can never be used in a reference since it is not of any valid type.

```
wire clk: Clock
wire halt: UInt<1>
stop(clk, halt, 42) : optional_name
```

16.2 Formatted Prints

The formatted print statement is used to print a formatted string during simulations of the circuit. Backends are free to generate hardware that relays this information to a hardware test harness, but this is not required by the FIRRTL specification.

A `printf` statement requires a clock signal, a print condition signal, a format string, and a variable list of argument signals. The condition signal must be a single bit unsigned integer type, and the argument signals must each have a ground type.

For information about execution ordering of clocked statements with observable environmental side effects, see Section 16.1.

The `printf` statement has an optional name attribute which can be used to attach metadata to the statement. The name is part of the module level namespace. However it can never be used in a reference since it is not of any valid type.

```
wire clk: Clock
wire cond: UInt<1>
wire a: UInt
wire b: UInt
printf(
  clk, cond, "a in hex: %x, b in decimal:%d.\n", a, b
) : optional_name
```

On each positive clock edge, when the condition signal is high, the `printf` statement prints out the format string where its argument placeholders are substituted with the value of the corresponding argument.

16.2.1 Format Strings

Format strings support the following argument placeholders:

- `%b` : Prints the argument in binary
- `%d` : Prints the argument in decimal
- `%x` : Prints the argument in hexadecimal
- `%%` : Prints a single `%` character

Format strings support the following escape characters:

- `\n` : New line
- `\t` : Tab
- `\\` : Back slash
- `\"` : Double quote
- `\'` : Single quote

16.3 Verification

To facilitate simulation, model checking and formal methods, there are three non-synthesizable verification statements available: assert, assume and cover. Assert and assume statements require a clock signal, a predicate signal, an enable signal, a format string and a variable list of argument signals. Cover statement requires a clock signal, a predicate signal, an enable signal, and a string literal. The predicate and enable signals must have single bit unsigned integer type. Assert and assume statement may print the format string as an explanatory message where its argument placeholders are substituted. See Section 16.2.1 for information about format strings. For cover statements the string indicates a suggested comment. When an assert or assume is violated the explanatory message may be issued as guidance. The explanatory message may be phrased as if prefixed by the words "Verifies that...".

Backends are free to generate the corresponding model checking constructs in the target language, but this is not required by the FIRRTL specification. Backends that do not generate such constructs should issue a warning. For example, the SystemVerilog emitter produces SystemVerilog assert, assume and cover statements, but the Verilog emitter does not and instead warns the user if any verification statements are encountered.

For information about execution ordering of clocked statements with observable environmental side effects, see Section 16.1.

Any verification statement has an optional name attribute which can be used to attach metadata to the statement. The name is part of the module level namespace. However it can never be used in a reference since it is not of any valid type.

16.4 Assert

The assert statement verifies that the predicate is true on the rising edge of any clock cycle when the enable is true. In other words, it verifies that enable implies predicate. When the predicate is false, the assert statement may print out the format string where its argument placeholders are substituted.

```
wire clk: Clock
wire pred: UInt<1>
wire en: UInt<1>
connect pred, eq(X, Y)
connect en, Z_valid
assert(
  clk, pred, en, "X equals Y when Z is valid but got X=%d Y=%d", X, Y
) : optional_name
```

16.5 Assume

The assume statement directs the model checker to disregard any states where the enable is true and the predicate is not true at the rising edge of the clock cycle. In other words, it reduces the states to be checked to only those where enable implies predicate is true by definition. In simulation, assume is treated as an assert. When the predicate is false in simulation, the assume statement may print out the format string where its argument placeholders are substituted.

```
wire clk: Clock
wire pred: UInt<1>
wire en: UInt<1>
connect pred, eq(X, Y)
connect en, Z_valid
assume(
  clk, pred, en, "X equals Y when Z is valid but got X=%d Y=%d", X,
  Y
) : optional_name
```

16.6 Cover

The cover statement verifies that the predicate is true on the rising edge of some clock cycle when the enable is true. In other words, it directs the model checker to find some way to make both enable and predicate true at some time step. The string argument may be emitted as a comment with the cover.

```
wire clk: Clock
wire pred: UInt<1>
wire en: UInt<1>
connect pred, eq(X, Y)
connect en, Z_valid
cover(clk, pred, en, "X equals Y when Z is valid") : optional_name
```

16.7 Intrinsic Statements

Intrinsics may be used as statements. If the intrinsic has a return type, it is unused. See Section 21.

17 Layer Blocks

The **layerblock** keyword declares a *layer block* and associates it with a layer. Statements inside a layer block are only executed when the associated layer is enabled.

A layer block must be associated with a layer declared in the current circuit. A layer block forms a lexical scope (as with Section 13.4) for all identifiers declared inside it—statements and expressions in a layer block may use identifiers declared outside the layer block, but identifiers declared in the layer block may not be used in parent lexical scopes. The statements in a layer block are restricted in what identifiers they are allowed to drive. A statement in a layer block may drive no sinks declared outside the layer block *with one exception*: a statement in a layer block may drive reference types declared outside the layer block if the reference types are associated with the layer in which the statement is declared (see: Section 7.3).

The circuit below contains one layer, `Bar`. Module `Foo` contains a layer block that creates a node computed from a port defined in the scope of `Foo`.

```
circuit Foo:
  ; Declaration of layer Bar with convention "bind"
  layer Bar, bind:

  public module Foo:
    input a: UInt<1>

    ; Declaration of a layer block associated with layer Bar
    ; inside module Foo
    layerblock Bar:
      node notA = not(a)
```

Multiple layer blocks may reference the same layer declaration. Each layer block is a new lexical scope even if they reference the same declaration. The circuit below shows module `Foo` having two layer blocks both referencing layer `Bar`. The first layer block may not reference node `c`. The second layer block may not reference node `b`.

```
circuit Foo:
  layer Bar, bind:

  public module Foo:
    input a: UInt<1>

    layerblock Bar: ; First layer block
      node b = a

    layerblock Bar: ; Second layer block
      node c = a
```

Layers may be nested. Layers are declared with the `layer` keyword indented under an existing `layer` keyword. The circuit below contains four layers, three of which are nested. `Bar` is the top-level layer. `Baz` and `Qux` are nested under `Bar`. `Quz` is nested under `Qux`.

```
circuit Foo:
  layer Bar, bind:
    layer Baz, bind:
    layer Qux, bind:
      layer Quz, bind:
```

Layer block nesting must match the nesting of declared layers. Layer blocks are declared under existing layer blocks with the `layerblock` keyword indented under an existing `layerblock` keyword. For the four layers in the circuit above, the following is a legal nesting of layerblocks:

```
  public module Foo:
    input a: UInt<1>

    layerblock Bar:
      node notA = not(a)
      layerblock Baz:
      layerblock Qux:
        layerblock Quz:
          node notNotA = not(notA)
```

Statements in a layer block may only read from ports or declarations of the current module, the current layer, or a parent layer—statements in a layer block may not drive components declared outside the layer block except reference types associated with the same layer block. In the above example, `notA` is accessible in the layer block associated with `Quz` because `notA` is declared in a parent layer block.

In the example below, module `Baz` defines a layer block associated with layer `Bar`. Module `Baz` has an output port, `_a`, that is also associated with layer `Bar`. This port can then be driven from the layer block. In module `Foo`, the port may be read from inside the layer block. *Stated differently, module `Baz` has an additional port `_a` that is only accessible in a layer block associated with `Bar`.*

```
circuit Foo:
  layer Bar, bind:

  module Baz:
    output _a: Probe<UInt<1>, Bar>

    wire a: UInt<1>

    layerblock Bar:
      node notA = not(a)
      define _a = probe(notA)

  public module Foo:

    inst baz of Baz

    layerblock Bar:
      node _b = read(baz._a)
```

If a port is associated with a nested layer then a period is used to indicate the nesting. E.g., the following circuit has a port associated with the nested layer `Bar.Baz`:

```
circuit Foo:
  layer Bar, bind:
    layer Baz, bind:

  public module Foo:
    output a: Probe<UInt<1>, Bar.Baz>
```

Layer blocks will be compiled to modules whose ports are derived from what they capture from their visible scope. For full details of the way layers are compiled, see the FIRRTL ABI specification.

18 Probes

Probes are a feature which facilitate graybox testing of FIRRTL modules.

Probes allow modules to expose circuit components contained inside of them. Given a probe, you can use it to read the value of the circuit component it references. For read-writeable probes, you can also force it, causing it to take on a specified value.

Since they are intended for verification, ports with a probe type do not necessarily result in physical hardware.

18.1 Types

There are two probe types, `Probe<T>` is a read-only variant and `RWProbe<T>` is a read-write variant. Probes may be layer-colored. See Section 7.3 for more information.

18.2 Layer Coloring

Probe types may be colored with a layer (see Section 5.4). A layer-colored probe type places restrictions on the operations which use it.

An operation may only "read" from a probe whose layer color is enabled when the operation is enabled. An operation that "writes" to a probe must be in a block whose layer color is enabled when the probe's layer color is enabled.

18.3 The `probe` and `rwprobe` Expressions

Given a reference to a circuit component, you can capture it in a probe with `probe` for a read-only probe or with `rwprobe` for a read-writeable probe.

18.4 The `define` Statement

Since neither `Probe<T>` nor `RWProbe<T>` is a connectable type (Section 7.5), circuit components with these types cannot be connected to. Instead, we use the **`define`** statement to route probes through a FIRRTL circuit.

The **`define`** statement works similarly to **`connect`**, but it has special interactions with conditional statements (Section 13). **`define`** statements may appear within conditional blocks, and when they do, they may refer to circuit components defined in a local scope. However, **`define`** are not subject to last connect semantics, and the **`define`** statement is always and unconditionally in effect.

All probes in a module definition must be initialized with a **`define`** statement.

Here is an example of a module which uses **`define`** to pass a number of probes to its ports:

```
public module Refs:
  input clock:  Clock
  ; read-only ref. to wire 'p'
  output a : Probe<{x: UInt<1>, y: UInt<2>}>
  ; force-able ref. to node 'q', inferred width.
  output b : RWProbe<UInt<1>>
  output c : Probe<UInt<3>> ; read-only ref. to register 'r'
  output d : Probe<Clock> ; ref. to input clock port

  wire p : {x: UInt<1>, flip y : UInt<2>}
  define a = probe(p) ; probe is passive
  wire q: UInt<1>
  connect q, UInt<1>(0)
```

```
define b = rwprobe(q)
reg r: UInt<3>, clock
define c = probe(r)
define d = probe(clock)
```

The target may also be a subcomponent of a circuit component:

```
public module Foo:
  input x : UInt<3>
  output y : { x: UInt<3>, p: Probe<UInt<3>> }
  output z : Probe<UInt<3>>[2]

  wire w : UInt<3>
  connect w, x
  connect y.x, w

  define y.p = probe(w)
  define z[0] = probe(w)
  define z[1] = probe(w)
```

18.5 The `read` Expressions

Both `Probe` and `RWProbe` may be read from using the `read` expression.

Keep in mind that probes are intended for verification and aren't intended to be synthesizable. A `read` expression can reach arbitrarily far into the module hierarchy to read from a distant circuit component, which is arguably unphysical at worst and potentially expensive at best.

18.6 Forcing

A `RWProbe` may be forced using the `force` and `force_initial` commands. When you force a component, it will ignore all other drivers and take on the value supplied by the force statement instead. The `release` and `release_initial` statements end the forcing, reconnecting the circuit component to its usual driver.

Like `read`, the force statements are verification constructs.

Name	Arguments	Argument Types
force_initial	(ref, val)	(`RWProbe<T>`, T)
release_initial	(ref)	(`RWProbe<T>`)
force	(clock, condition, ref, val)	(`Clock`, `UInt<1>`, `RWProbe<T>`, T)
release	(clock, condition, ref)	(`Clock`, `UInt<1>`, `RWProbe<T>`)

Backends optionally generate corresponding constructs in the target language, or issue an warning.

The following `AddRefs` module is used in the examples that follow for each construct.

```
public module AddRefs:
  output a : RWProbe<UInt<2>>
  output b : RWProbe<UInt<2>>
  output c : RWProbe<UInt<2>>
  output sum : UInt<3>

  wire x: UInt<2>
  connect x, UInt<2>(0)
  wire y: UInt<2>
  connect y, UInt<2>(0)
  wire z: UInt<2>
  connect z, UInt<2>(0)
  connect sum, add(x, add(y, z))

  define a = rwprobe(x)
  define b = rwprobe(y)
  define c = rwprobe(z)
```

18.6.1 Initial Force and Initial Release

These variants force and release continuously:

Example:

```
public module ForceAndRelease:
  output o : UInt<3>

  inst r of AddRefs
  connect o, r.sum

  force_initial(r.a, UInt<2>(0))
  force_initial(r.a, UInt<2>(1))
  force_initial(r.b, UInt<2>(2))
  force_initial(r.c, UInt<2>(3))
  release_initial(r.c)
```

In this example, the output `o` will be `3`. Note that globally the last force statement overrides the others until another force or release including the target.

Sample SystemVerilog output for the force and release statements would be:

```
initial begin
  force ForceAndRelease.AddRefs.x = 2'd0;
  force ForceAndRelease.AddRefs.x = 2'd1;
  force ForceAndRelease.AddRefs.y = 2'd2;
  force ForceAndRelease.AddRefs.z = 2'd3;
```

```
  release ForceAndRelease.AddRefs.z;
end
```

The `force_initial` and `release_initial` statements may occur under `when` blocks which becomes a check of the condition first. Note that this condition is only checked once and changes to it afterwards are irrelevant, and if executed the force will continue to be active. For more control over their behavior, the other variants should be used. Example:

```
    when c : force_initial(ref, x)
```

would become:

```
initial if (c) force Foo.a.b = x;
```

18.6.2 Force and Release

These more detailed variants allow specifying a clock and condition for when activating the force or release behavior continuously:

```
  public module ForceAndRelease:
    input a: UInt<2>
    input clock : Clock
    input cond : UInt<1>
    output o : UInt<3>

    inst r of AddRefs
    connect o, r.sum

    force(clock, cond, r.a, a)
    release(clock, not(cond), r.a)
```

Which at the positive edge of `clock` will either force or release `AddRefs.x`. Note that once active, these remain active regardless of the condition, until another force or release.

Sample SystemVerilog output:

```
always @(posedge clock) begin
  if (cond)
    force ForceAndRelease.AddRefs.x = a;
  else
    release ForceAndRelease.AddRefs.x;
end
```

Condition is checked in procedural block before the force, as shown above. When placed under `when` blocks, condition is mixed in as with other statements (e.g., `assert`).

18.6.3 Non-Passive Force Target

Force on a non-passive bundle drives in the direction of each field's orientation.

Example:

```
circuit Top:
  public module Top:
    input x : {a: UInt<2>, flip b: UInt<2>}
    output y : {a: UInt<2>, flip b: UInt<2>}

    inst d of DUT
    connect d.x, x
    connect y, d.y

    wire val : {a: UInt<2>, b: UInt<2>}
    connect val.a, UInt<2>(1)
    connect val.b, UInt<2>(2)

    ; Force takes a RWProbe and overrides the target with 'val'.
    force_initial(d.xp, val)

  module DUT :
    input x : {a: UInt<2>, flip b: UInt<2>}
    output y : {a: UInt<2>, flip b: UInt<2>}
    output xp : RWProbe<{a: UInt<2>, b: UInt<2>}>

    ; Force drives p.a, p.b, y.a, and x.b, but not y.b and x.a
    wire p : {a: UInt<2>, flip b: UInt<2>}
    define xp = rwprobe(p)
    connect p, x
    connect y, p
```

18.7 Limitations

18.7.1 `RWProbe` with Aggregate Types

Probes of aggregates do not work under all ABI versions. In these cases, it is recommended that if you have a need to probe a bundle, you instead use a bundle of probes, each targeting an individual subcomponent.

For more information on the how probes are lowered, see the FIRRTL ABI Specification.

18.7.2 No `RWProbes` of Ports on Public Modules

Conceptually, in order to force a value onto a circuit component, you need to be able to disconnect it from its former driver. However, the driver of a port is another port in some other module. And so, to force a port, you must have control over both "ends" of a port.

Trouble arises with ports on a public module (Section 5.2.1). Since we don't control the remote side of ports (since they might live in another FIRRTL file), we cannot force these

ports either. For this reason, it is an error to use `rwprobe` on any port on a public module.

18.7.3 Input Ports

Probes may not be inputs to modules.

19 Expressions

FIRRTL expressions are used for creating constant integers, for creating literal property type expressions, for referencing a circuit component, for statically and dynamically accessing a nested element within a component, for creating multiplexers, for performing primitive operations, for reading a remote reference to a probe, and for intrinsics (Section 21).

19.1 Constant Integer Expressions

A constant unsigned or signed integer expression can be created from an integer literal or radix-specified integer literal. An optional positive bit width may be specified. Constant integer expressions are of constant type. All of the following examples create a 10-bit unsigned constant integer expressions representing the number `42`:

```
UInt<10>(42)
UInt<10>(0b101010)
UInt<10>(0o52)
UInt<10>(0h2A)
UInt<10>(0h2a)
```

Note that it is an error to supply a bit width that is not large enough to fit the given value. If the bit width is omitted, then the minimum number of bits necessary to fit the given value will be inferred. All of the following will infer a bit width of five:

```
UInt(42)
UInt(0b101010)
UInt(0o52)
UInt(0h2A)
UInt(0h2a)
```

Signed constant integer expressions may be created from a signed integer literal or signed radix-encoded integer literal. All of the following examples create a 10-bit signed hardware integer representing the number `-42`:

```
SInt<10>(-42)
SInt<10>(-0b101010)
SInt<10>(-0o52)
SInt<10>(-0h2A)
SInt<10>(-0h2a)
```

Signed constant integer expressions may also have an inferred width. All of the following examples create and infer a 6-bit signed integer with value `-42`:

```
SInt(-42)
SInt(-0b101010)
SInt(-0o52)
SInt(-0h2A)
SInt(-0h2a)
```

19.2 Property Literal Expressions

A literal property type expression can be created for a given property type. The property type name is followed by an appropriate literal value for the property type, enclosed in parentheses.

19.2.1 Integer Property Literal Expresssions

A literal `Integer` property type expression can be created from an integer literal. The following examples show literal `Integer` property type expressions.

```
Integer(42)
Integer(-42)
```

19.3 Enum Expressions

An enumeration can be constructed by applying an enumeration type to a variant tag and a data value expression. The data value expression may be omitted when the data type is `UInt<0>(0)`, where it is implicitly defined to be `UInt<0>(0)`.

```
{|a, b, c|}(a)
{|some: UInt<8>, none|}(some, x)
```

19.4 Multiplexers

A multiplexer outputs one of two input expressions depending on the value of an unsigned selection signal.

The following example connects to the `c` port the result of selecting between the `a` and `b` ports. The `a` port is selected when the `sel` signal is high, otherwise the `b` port is selected.

```
public module MyModule :
  input a: UInt<3>
  input b: UInt<3>
  input sel: UInt<1>
  output c: UInt<3>
  connect c, mux(sel, a, b)
```

A multiplexer expression is legal only if the following holds.

1. The type of the selection signal is an unsigned integer.
2. The width of the selection signal is any of:

    1. Zero-bit
    2. One-bit
    3. Unspecified, but is illegal if infers to wider than one-bit
3. The types of the two input expressions are equivalent.
4. The types of the two input expressions are passive (see Section 7.7).

19.5 Primitive Operations

All fundamental operations on ground types are expressed as a FIRRTL primitive operation. In general, each operation takes some number of argument expressions, along with some number of integer literal parameters.

The general form of a primitive operation is expressed as follows:

```
op(arg0, arg1, ..., argn, int0, int1, ..., intm)
```

The following examples of primitive operations demonstrate adding two expressions, `a` and `b`, shifting expression `a` left by 3 bits, selecting the fourth bit through and including the seventh bit in the `a` expression, and interpreting the expression `x` as a Clock typed signal.

```
add(a, b)
shl(a, 3)
bits(a, 7, 4)
asClock(x)
```

Section 25 will describe the format and semantics of each primitive operation.

19.6 Primitive Property Operations

All fundamental operations on property types are expressed as a FIRRTL primitive operation. In general, each operation takes some number of property type expressions as arguments.

The general form of a primitive property operation is expressed as follows:

```
op(arg0, arg1, ..., argn)
```

The following examples of primitive property operations demonstrate adding two property expressions, `a` and `b`, and adding an integer property literal to expression `a`.

```
integer_add(a, b)
integer_add(a, Integer(2))
```

Section 26 will describe the format and semantics of each primitive property operation.

19.7 Reading Probe References

Probes are read using the `read` operation.

Read expressions have source flow and can be connected to other components:

```
module Foo :
  output p : Probe<UInt>
  ; ...

public module Bar :
  output x : UInt<3>

  inst f of Foo
  connect x, read(f.p) ; indirectly access the probed data
```

Indexing statically (sub-field, sub-index) into a probed value is allowed as part of the read:

```
module Foo :
  output p : Probe<{a: UInt, b: UInt}>
  ; ...

public module Bar :
  output x : UInt<3>

  inst f of Foo
  connect x, read(f.p.b) ; indirectly access the probed data
```

Read operations can be used anywhere a signal of the same underlying type can be used, such as the following:

```
  connect x, add(read(f.p).a, read(f.p).b)
```

The source of the probe must reside at or below the point of the `read` expression in the design hierarchy.

19.8 Probe

Probe references are generated with probe expressions.

The probe expression creates a reference to a read-only or force-able view of the data underlying the specified reference expression.

The type of the produced probe reference is always passive, but the probed expression may not be. Memories and their ports are not supported as probe targets.

There are two probe varieties: `probe` and `rwprobe` for producing probes of type **`Probe`** and **`RWProbe`**, respectively.

The following example exports a probe reference to a port:

```
public module MyModule :
  input in: UInt<5>
  output r : Probe<UInt<5>>
```

```
define r = probe(in)
```

The probed expression must be a static reference.

See Sections 7.3, 19.8 for more details on probe references and their use.

20 Namespaces

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

Each module has an identifier namespace containing the names of all port and circuit component declarations. Thus, all declarations within a module must have unique names.

Within a bundle type declaration, all field names must be unique.

Within a memory declaration, all port names must be unique.

Any modifications to names must preserve the uniqueness of names within a namespace.

21 Intrinsics

Intrinsics are expressions and statements which represent implementation-defined, compiler-provided functionality.

Intrinsics generally are used for functionality which requires knowledge of the implementation or circuit not available to a library writer. Which intrinsics are supported by an implementation is defined by the implementation.

Intrinsics first specify the intrinsic: the name of the intrinsic, any parameters, and return type if applicable. Any inputs to the intrinsic follow. An implementation shall type-check the specification and all operands.

Below are some examples of intrinsics. These are for demonstration and their meaning or validity is determined by the implementation.

The following shows an intrinsic expression for the intrinsic named "circt_ltl_delay" with two parameters, returns `UInt<1>`, and has one operand.

```
public module Foo :
  input in : UInt<1>

  node d = intrinsic(circt_ltl_delay<delay = 1, length = 0> : UInt<1>, in)
```

The following has an intrinsic statement with an intrinsic expression as its operand. The statement is for the intrinsic named "circt_verif_assert". The expression is for the intrinsic named "circt_isX" which returns a `UInt<1>` and takes an operand.

```
public module Foo :
  input data : UInt<5>

  intrinsic(circt_verif_assert, intrinsic(circt_isX: UInt<1>, data))
```

Operands and the return type of intrinsics must be passive and either ground or aggregate. When used as an expression, the intrinsic must have a return type. The types of intrinsic module parameters may only be literal integers or string literals.

22 Annotations

Annotations encode arbitrary metadata and associate it with zero or more targets (Section 22.1) in a FIRRTL circuit.

Annotations are represented as a dictionary, with a required "class" field which describes which annotation it is. An annotation has an optional "target" field which represents the IR object it is attached to. Annotations may have arbitrary additional fields attached. Some annotation classes extend other annotations, which effectively means that the subclass annotation implies to effect of the parent annotation.

Annotations are serializable to JSON.

Below is an example annotation used to mark some module `foo`:

```
{
  "class":"myannotationpackage.FooAnnotation",
  "target":"MyModule>foo"
}
```

Below is an example of an annotation which does not have a target:

```
{
  "class":"myannotationpackage.BarAnnotation"
}
```

22.1 Targets

A circuit is described, stored, and optimized in a folded representation. For example, there may be multiple instances of a module which will eventually become multiple physical copies of that module on the die.

Targets are a mechanism to identify specific hardware in specific instances of modules in a FIRRTL circuit. A target consists of a root module, an optional instance hierarchy, and an optional reference. A target can only identify hardware with a name, e.g., a module, instance, register, wire, or node. References may further refer to specific fields or subindices in aggregates. A target with no instance hierarchy is local. A target with an instance hierarchy is non-local.

Targets use a shorthand syntax of the form:

```
target = module , [ { "/" (instance) ":" (module) } , [ ">" , ref ] ]
```

A reference is a name inside a module and one or more qualifying tokens that encode subfields (of a bundle) or subindices (of a vector):

```
ref = name , { ( "[" , index , "]" ) | ( "." , field ) }
```

Targets are specific enough to refer to any specific module in a folded, unfolded, or partially folded representation.

To show some examples of what these look like, consider the following example circuit. This consists of four instances of module `Baz`, two instances of module `Bar`, and one instance of module `Foo`:

```
circuit Foo:
  public module Foo:
    inst a of Bar
    inst b of Bar
  module Bar:
    inst c of Baz
    inst d of Baz
  module Baz:
    skip
```

This circuit can be represented in a *folded*, completely *unfolded*, or in some *partially folded* state. Figure Figure 1 shows the folded representation. Figure Figure 2 shows the completely unfolded representation where each instance is broken out into its own module.

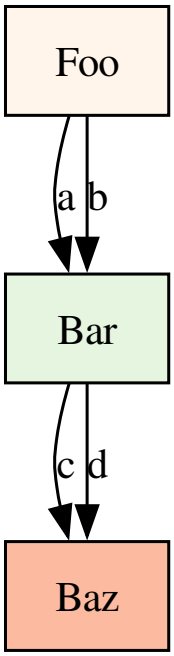

Figure 1: A folded representation of circuit Foo

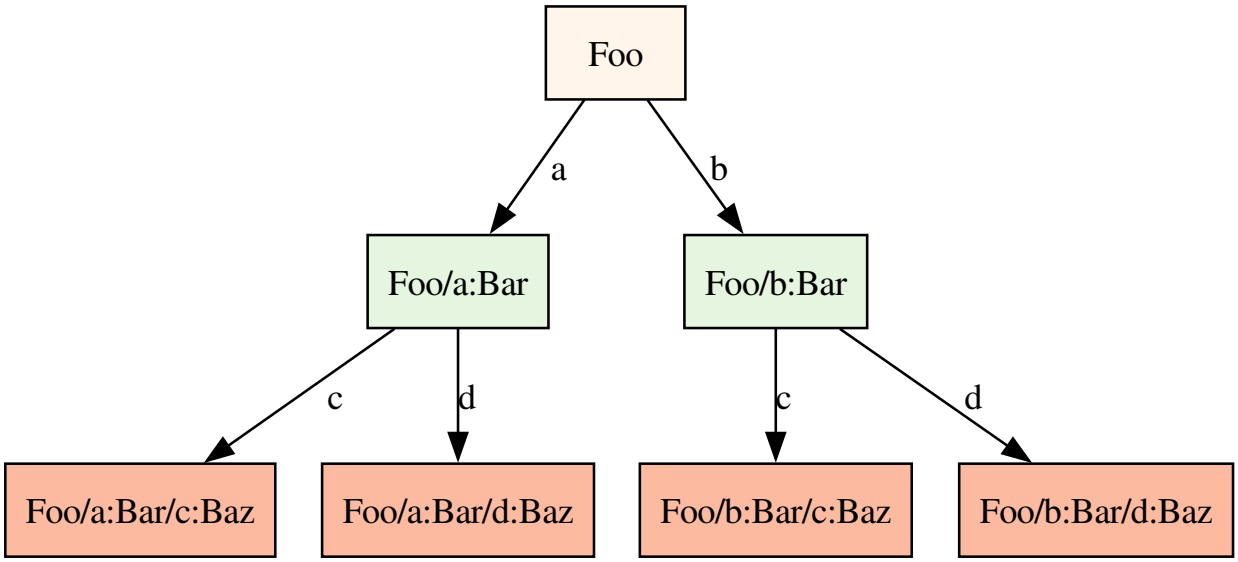

Figure 2: A completely unfolded representation of circuit Foo

Using targets (or multiple targets), any specific module, instance, or combination of instances can be expressed. Some examples include:

Target	Description
`Foo`	refers to module `Foo` (or the only instance of module `Foo`)
`Bar`	refers to module `Bar` (or both instances of module `Bar`)
`Foo/a:Bar`	refers just to one instance of module `Bar`
`Foo/b:Bar/c:Baz`	refers to one instance of module `Baz`
`Bar/d:Baz`	refers to two instances of module `Baz`

If a target does not contain an instance path, it is a *local* target. A local target points to all instances of a module. If a target contains an instance path, it is a *non-local* target. A non-local target *may* not point to all instances of a module. Additionally, a non-local target may have an equivalent local target representation.

22.2 Annotation Storage

Annotations may be stored in one or more JSON files using an array-of-dictionaries format. The following shows a valid annotation file containing two annotations:

```
[
  {
    "class":"hello",
    "target":"~Foo|Bar"
  },
  {
```

```
    "class":"world",
    "target":"~Foo|Baz"
  }
]
```

Annotations may also be stored in-line along with the FIRRTL circuit by wrapping Annotation JSON in `%[ ... ]`. The following shows the above annotation file stored in-line:

```
circuit Foo: %[[
  {
    "class":"hello",
    "target":"~Foo|Bar"
  },
  {
    "class":"world",
    "target":"~Foo|Baz"
  }
]]
  module Baz :
  module Bar :
  public module Foo :
```

Any legal JSON is allowed, meaning that the above JSON may be stored "minimized" all on one line.

23 Semantics of Values

FIRRTL is defined for 2-state boolean logic. The behavior of a generated circuit in a language, such as Verilog or VHDL, which have multi-state logic, is undefined in the presence of values which are not 2-state. A FIRRTL compiler need only respect the 2-state behavior of a circuit. This is a limitation on the scope of what behavior is observable (i.e., a relaxation of the "as-if" rule).

23.1 Indeterminate Values

An indeterminate value represents a value which is unknown or unspecified. Indeterminate values are generally implementation defined, with constraints specified below. An indeterminate value may be assumed to be any specific value (not necessarily literal), at an implementation's discretion, if, in doing so, all observable behavior is as if the indeterminate value always took the specific value.

This allows transformations such as the following, where when `a` has an indeterminate value, the implementation chooses to consistently give it a value of `v`. An alternate, legal mapping, lets the implementation give it the value `42`. In both cases, there is no visibility of `a` when it has an indeterminate value which is not mapped to the value the implementation chooses.

```
public module IValue :
  output o : UInt<8>
  input c : UInt<1>
  input v : UInt<8>

  wire a : UInt<8>
  invalidate a
  when c :
    connect a, v
  connect o, a
```

is transformed to:

```
public module IValue :
  output o : UInt<8>
  input c : UInt<1>
  input v : UInt<8>

  connect o, v
```

Note that it is equally correct to produce:

```
public module IValue :
  output o : UInt<8>
  input c : UInt<1>
  input v : UInt<8>

  wire a : UInt<8>
  when c :
    connect a, v
  else :
    connect a, UInt<8>(0h42)
  connect o, a
```

The behavior of constructs which cause indeterminate values is implementation defined with the following constraints.

- Register initialization is done in a consistent way for all registers. If code is generated to randomly initialize some registers (or 0 fill them, etc), it should be generated for all registers.
- All observations of a unique instance of an expression with indeterminate value must see the same value at runtime. Multiple readers of a value will see the same runtime value.
- Indeterminate values captured in stateful elements are not time-varying. Time-aware constructs, such as registers, which hold an indeterminate value will return the same runtime value unless something changes the value in a normal way. For example, an uninitialized register will return the same value over multiple clock cycles until it is written (or reset).

- The value produced at runtime for an expression which produced an intermediate value shall only be a function of the inputs of the expression. For example, an out-of-bounds vector access shall produce the same value for a given out-of-bounds index and vector contents.
- Two constructs with indeterminate values place no constraint on the identity of their values. For example, two uninitialized registers, which therefore contain indeterminate values, do not need to be equal under comparison.

24 FIRRTL Compiler Implementation Details

This section provides auxiliary information necessary for developers of a FIRRTL Compiler *implementation*. A FIRRTL Compiler is a program that converts FIRRTL text to another representation, e.g., Verilog, VHDL, a programming language, or a binary program.

24.1 Module Conventions

A module's convention describes how its ports are lowered to the output format, and serves as a kind of ABI for modules.

24.1.1 The "Scalarized" Convention

The scalarized convention lowers aggregate ports to ground values. The scalarized convention should be the default convention for "public" modules, such as the top module of a circuit, "device under test", or an extmodule.

The lowering algorithm for the scalarized convention operates as follows:

1. Ports are scalarized in the order they are declared.
2. Ground-typed ports' names are unmodified.
3. Vector-typed ports are scalarized to ground-typed ports by appending a suffix, `_<i>`, to the i^{th} element of the vector. Elements are scalarized recursively, depth-first, and left-to-right.
4. Bundle-typed ports are scalarized to ground-typed ports by appending a suffix, `_<name>`, to the field called `name`. Fields are scalarized recursively, depth-first, and left-to-right.

E.g., consider the following port:

```
public module Top :
  input a : { b: UInt<1>, c: UInt<2> }[2]
```

Scalarization breaks `a` into the following ports:

```
public module Top :
  input a_0_b : UInt<1>  ; a[0].b
  input a_0_c : UInt<2>  ; a[0].c
```

```
    input a_1_b : UInt<1>  ; a[1].b
    input a_1_c : UInt<2>  ; a[1].c
```

The body of a module definition introduces a new, empty namespace. As new port names are added, these names must be unique with respect to this namespace. In the case of a collision during renaming, priority will be given to values that are converted first.

If a name is already taken, that name will be made unique by appending a suffix `_<i>` to the name, where `i` is the lowest nonnegative integer that gives a unique name.

E.g., consider the following ports:

```
public module Top :
  input a : { b: UInt<1>[2], b_0: UInt<2>, b_1: UInt<3> }
  input a_b : UInt<4>[2]
  input a_b_0 : UInt<5>
```

Scalarization breaks these ports into the following ports:

```
public module Top :
  input a_b_0: UInt<1>    ; a.b[0]
  input a_b_1: UInt<1>    ; a.b[1]
  input a_b_0_0: UInt<2>  ; a.b_0
  input a_b_1_0: UInt<3>  ; a.b_1
  input a_b_0_1: UInt<4>  ; a_b[0]
  input a_b_1_1: UInt<4>  ; a_b[1]
  input a_b_0_2: UInt<5>  ; a_b_0
```

Named components in the body of a module will be renamed as needed to ensure port names follow this convention.

24.2 The "Internal" Convention

Private modules (i.e. modules that are *not* the top of a circuit, device under test, or an extmodule) have no specified ABI. The compiler is free to transform the ports of a private module in any way, or not at all. Private modules are said to have "internal" convention.

25 Primitive Operations

The arguments of all primitive operations must be expressions with ground types, while their parameters are integer literals. Each specific operation can place additional restrictions on the number and types of their arguments and parameters. Primitive operations may have all their arguments of constant type, in which case their return type is of constant type. If the operation has a mixed constant and non-constant arguments, the result is non-constant.

Notationally, the width of an argument e is represented as w_e.

25.1 Add Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
add	(e1,e2)	()	(UInt,UInt)	UInt	$\max(w_{e1},w_{e2})+1$
			(SInt,SInt)	SInt	$\max(w_{e1},w_{e2})+1$

The add operation result is the sum of e1 and e2 without loss of precision.

25.2 Subtract Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
sub	(e1,e2)	()	(UInt,UInt)	UInt	$\max(w_{e1},w_{e2})+1$
			(SInt,SInt)	SInt	$\max(w_{e1},w_{e2})+1$

The subtract operation result is e2 subtracted from e1, without loss of precision.

25.3 Multiply Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
mul	(e1,e2)	()	(UInt,UInt)	UInt	$w_{e1}+w_{e2}$
			(SInt,SInt)	SInt	$w_{e1}+w_{e2}$

The multiply operation result is the product of e1 and e2, without loss of precision.

25.4 Divide Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
div	(num,den)	()	(UInt,UInt)	UInt	w_{num}
			(SInt,SInt)	SInt	$w_{num}+1$

The divide operation divides num by den, truncating the fractional portion of the result. This is equivalent to rounding the result towards zero. The result of a division where den is zero is undefined.

25.5 Modulus Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
rem	(num,den)	()	(UInt,UInt) (SInt,SInt)	UInt SInt	$\min(w_{num},w_{den})$ $\min(w_{num},w_{den})$

The modulus operation yields the remainder from dividing num by den, keeping the sign of the numerator. Together with the divide operator, the modulus operator satisfies the relationship below:

```
num = add(mul(den,div(num,den)),rem(num,den))}
```

25.6 Comparison Operations

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
eq,neq	(e1,e2)	()	(UInt,UInt) (SInt,SInt)	UInt UInt	1 1

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
lt,leq	(e1,e2)	()	(UInt,UInt) (SInt,SInt)	UInt UInt	1 1

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
gt,geq	(e1,e2)	()	(UInt,UInt) (SInt,SInt)	UInt UInt	1 1

The comparison operations return an unsigned 1 bit signal with value one if e1 is less than (lt), less than or equal to (leq), greater than (gt), greater than or equal to (geq), equal to (eq), or not equal to (neq) e2. The operation returns a value of zero otherwise.

25.7 Padding Operations

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
pad	(e)	(n)	(UInt) (SInt)	UInt SInt	$\max(w_e,n)$ $\max(w_e,n)$

If e's bit width is smaller than n, then the pad operation zero-extends or sign-extends e up to the given width n. Otherwise, the result is simply e. n must be non-negative.

25.8 Interpret As UInt

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
asUInt	(e)	()	(UInt)	UInt	w_e
			(SInt)	UInt	w_e
			(Clock)	UInt	1
			(Reset)	UInt	1
			(AsyncReset)	UInt	1

The interpret as UInt operation reinterprets e's bits as an unsigned integer.

25.9 Interpret As SInt

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
asSInt	(e)	()	(UInt)	SInt	w_e
			(SInt)	SInt	w_e
			(Clock)	SInt	1
			(Reset)	SInt	1
			(AsyncReset)	SInt	1

The interpret as SInt operation reinterprets e's bits as a signed integer according to two's complement representation.

25.10 Interpret as Clock

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
asClock	(e)	()	(UInt)	Clock	n/a
			(SInt)	Clock	n/a
			(Clock)	Clock	n/a
			(Reset)	Clock	n/a
			(AsyncReset)	Clock	n/a

The result of the interpret as clock operation is the Clock typed signal obtained from interpreting a single bit integer as a clock signal.

25.11 Interpret as AsyncReset

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
asAsyncReset	(e)	()	(AsyncReset)	AsyncReset	n/a

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
			(UInt)	AsyncReset	n/a
			(SInt)	AsyncReset	n/a
			(Interval)	AsyncReset	n/a
			(Clock)	AsyncReset	n/a
			(Reset)	AsyncReset	n/a

The result of the interpret as asynchronous reset operation is an AsyncReset typed signal.

25.12 Shift Left Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
shl	(e)	(n)	(UInt)	UInt	w_e+n
			(SInt)	SInt	w_e+n

The shift left operation concatenates n zero bits to the least significant end of e. n must be non-negative.

25.13 Shift Right Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
shr	(e)	(n)	(UInt)	UInt	max(w_e-n, 0)
			(SInt)	SInt	max(w_e-n, 1)

The shift right operation truncates the least significant n bits from e. If n is greater than or equal to the bit-width of e, the resulting value will be zero for unsigned types and the sign bit for signed types. n must be non-negative.

25.14 Dynamic Shift Left Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
dshl	(e1, e2)	()	(UInt, UInt)	UInt	w_{e1} + 2^w_{e2} - 1
			(SInt, UInt)	SInt	w_{e1} + 2^w_{e2} - 1

The dynamic shift left operation shifts the bits in e1 e2 places towards the most significant bit. e2 zeroes are shifted in to the least significant bits.

25.15 Dynamic Shift Right Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
dshr	(e1, e2)	()	(UInt, UInt)	UInt	w_{e1}
			(SInt, UInt)	SInt	w_{e1}

The dynamic shift right operation shifts the bits in e1 e2 places towards the least significant bit. e2 signed or zeroed bits are shifted in to the most significant bits, and the e2 least significant bits are truncated.

25.16 Arithmetic Convert to Signed Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
cvt	(e)	()	(UInt)	SInt	w_e+1
			(SInt)	SInt	w_e

The result of the arithmetic convert to signed operation is a signed integer representing the same numerical value as e.

25.17 Negate Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
neg	(e)	()	(UInt)	SInt	w_e+1
			(SInt)	SInt	w_e+1

The result of the negate operation is a signed integer representing the negated numerical value of e.

25.18 Bitwise Complement Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
not	(e)	()	(UInt)	UInt	w_e
			(SInt)	UInt	w_e

The bitwise complement operation performs a logical not on each bit in e.

25.19 Binary Bitwise Operations

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
and,or,xor	(e1, e2)	()	(UInt,UInt) (SInt,SInt)	UInt UInt	$\max(w_{e1},w_{e2})$ $\max(w_{e1},w_{e2})$

The above bitwise operations perform a bitwise and, or, or exclusive or on e1 and e2. The result has the same width as its widest argument, and any narrower arguments are automatically zero-extended or sign-extended to match the width of the result before performing the operation.

25.20 Bitwise Reduction Operations

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
andr,orr,xorr	(e)	()	(UInt) (SInt)	UInt UInt	1 1

The bitwise reduction operations correspond to a bitwise and, or, and exclusive or operation, reduced over every bit in e.

In all cases, the reduction incorporates as an inductive base case the "identity value" associated with each operator. This is defined as the value that preserves the value of the other argument: one for and (as $x \wedge 1 = x$), zero for or (as $x \vee 0 = x$), and zero for xor (as $x \oplus 0 = x$). Note that the logical consequence is that the and-reduction of a zero-width expression returns a one, while the or- and xor-reductions of a zero-width expression both return zero.

25.21 Concatenate Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
cat	(e1,e2)	()	(UInt, UInt) (SInt, SInt)	UInt UInt	$w_{e1}+w_{e2}$ $w_{e1}+w_{e2}$

The result of the concatenate operation is the bits of e1 concatenated to the most significant end of the bits of e2.

25.22 Bit Extraction Operation

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
bits	(e)	(hi,lo)	(UInt) (SInt)	UInt UInt	hi-lo+1 hi-lo+1

The result of the bit extraction operation are the bits of e between lo (inclusive) and hi (inclusive). hi must be greater than or equal to lo. Both hi and lo must be non-negative and strictly less than the bit width of e. The index of the least significant bit is 0 and the index of the most significant bit is one less than the width of the argument.

25.23 Head

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
head	(e)	(n)	(UInt) (SInt)	UInt UInt	n n

The result of the head operation are the n most significant bits of e. n must be non-negative and less than or equal to the bit width of e.

25.24 Tail

Name	Arguments	Parameters	Arg Types	Result Type	Result Width
tail	(e)	(n)	(UInt) (SInt)	UInt UInt	w_e-n w_e-n

The tail operation truncates the n most significant bits from e. n must be non-negative and less than or equal to the bit width of e.

26 Primitive Property Operations

The arguments of all primitive property operations must be expressions with property types. Each specific operation can place additional restrictions on the number and types of their arguments. In general, primitive property operations are named with a prefix indicating the property type on which they operate.

26.1 Integer Arithmetic

Integer arithmetic operations take `Integer` property type expressions as arguments and return an `Integer` property type result.

26.1.1 Integer Add Operation

Name	Arguments	Arg Types	Result Type
integer_add	(e1,e2)	(Integer,Integer)	Integer

The add operation result is the arbitrary precision signed integer arithmetic sum of e1 and e2.

26.1.2 Integer Multiply Operation

Name	Arguments	Arg Types	Result Type
integer_mul	(e1,e2)	(Integer,Integer)	Integer

The multiply operation result is the arbitrary precision signed integer arithmetic product of e1 and e2.

26.1.3 Integer Shift Right Operation

Name	Arguments	Arg Types	Result Type
integer_shr	(e1,e2)	(Integer,Integer)	Integer

The shift right operation result is the arbitrary precision signed integer arithmetic shift right of e1 by e2. e2 sign bits from e1 are shifted into the most significant bits, and the e2 least significant bits of e1 are truncated. e2 must be non-negative.

26.1.4 Integer Shift Left Operation

Name	Arguments	Arg Types	Result Type
integer_shl	(e1,e2)	(Integer,Integer)	Integer

The shift left operation result is the arbitrary precision signed integer arithmetic shift left of e1 by e2. e2 zero bits are shifted into the least significant bits of e1, and the e2 most significant bits of e1 are truncated. e2 must be non-negative.

26.2 List Operations

List operations create `List` property type expressions from other property expressions.

26.2.1 List Construction Operation

Name	Arguments	Arg Types	Result Type
List<t>	(e*)	(t*)	List<t>

The list construction operation constructs a `List` property type expression of a given element type. The `List` constructor is parameterized by element type t, and accepts zero or more property type expressions e of type t.

26.2.2 List Concatenation Operation

Name	Arguments	Arg Types	Result Type
list_concat	(e+)	(t+)	List<t>

The list concatenation operation constructs a **List** property type expression by concatenating one or more lists of the same element type t.

27 Notes on Syntax

FIRRTL's syntax is designed to be human-readable but easily algorithmically parsed.

FIRRTL allows for two types of identifiers:

1. Identifiers
2. Literal Identifiers

Identifiers may only have the following characters: upper and lower case letters, digits, and `_`. Identifiers cannot begin with a digit.

Literal identifiers allow for using an expanded set of characters in an identifier. Such an identifier is encoded using leading and trailing backticks, `` ` ``. A literal identifier has the same restrictions as an identifier, *but it is allowed to start with a digit*. E.g., it is legal to use `` `0` `` as a literal identifier in a Bundle field (or anywhere else an identifier may be used).

A FIRRTL compiler is allowed to change a literal identifier to a legal identifier in the target language (e.g., Verilog) if the literal identifier is not directly representable in the target language.

Comments begin with a semicolon and extend until the end of the line.

In FIRRTL, indentation is significant. Indentation must consist of spaces only—tabs are illegal characters. The number of spaces appearing before a FIRRTL IR statement is used to establish its *indent level*. Statements with the same indent level have the same context. The indent level of the **circuit** declaration must be zero.

Certain constructs (**circuit**, **module**, **when**, and **else**) create a new sub-context. The indent used on the first line of the sub-context establishes the indent level. The indent level of a sub-context is one higher than the parent. All statements in the sub-context must be indented by the same number of spaces. To end the sub-context, a line must return to the indent level of the parent.

Since conditional statements (**when** and **else**) may be nested, it is possible to create a hierarchy of indent levels, each with its own number of preceding spaces that must be larger than its parent's and consistent among all direct child statements (those that are not children of an even deeper conditional statement).

As a concrete guide, a few consequences of these rules are summarized below:

- The `circuit` keyword must not be indented.
- All `module` keywords must be indented by the same number of spaces.
- In a module, all port declarations and all statements (that are not children of other statements) must be indented by the same number of spaces.
- The number of spaces comprising the indent level of a module is specific to each module.
- The statements comprising a conditional statement's branch must be indented by the same number of spaces.
- The statements of nested conditional statements establish their own, deeper indent level.
- Each `when` and each `else` context may have a different number of non-zero spaces in its indent level.

As an example illustrating some of these points, the following is a legal FIRRTL circuit:

```
circuit Foo :
    public module Foo :
      skip
    module Bar :
     input a: UInt<1>
     output b: UInt<1>
     when a:
         connect b, a
     else:
       connect b, not(a)
```

All circuits, modules, ports and statements can optionally be followed with the info token `@[fileinfo]` where fileinfo is a string containing the source file information from where it was generated. The following characters need to be escaped with a leading '\': '`\n`' (new line), '`\t`' (tab), '`]`' and '\' itself.

The following example shows the info tokens included:

```
circuit Top : @[myfile.txt 14:8]
  public module Top : @[myfile.txt 15:2]
    output out: UInt<3> @[myfile.txt 16:3]
    input b: UInt<32> @[myfile.txt 17:3]
    input c: UInt<1> @[myfile.txt 18:3]
    input d: UInt<16> @[myfile.txt 19:3]
    wire a: UInt<2> @[myfile.txt 21:8]
    when c : @[myfile.txt 24:8]
      connect a, b @[myfile.txt 27:16]
    else :
      connect a, d @[myfile.txt 29:17]
    connect out, add(a,a) @[myfile.txt 34:4]
```

27.1 Literals

FIRRTL has both integer, string, and raw string literals.

An integer literal is a signed or unsigned decimal integer. The following are examples of integer literals:

```
42
-9000
```

A string literal is a sequence of characters with a leading " and a trailing ". The following is an example of a string literal:

```
"hello"
```

A raw string literal is a sequence of characters with a leading ' and a trailing '. The following is an example of a raw string literal:

```
'world'
```

A radix-specified integer literal is a special integer literal with one of the following leading characters to indicate the numerical encoding:

- `0b` – for representing binary numbers
- `0o` – for representing octal numbers
- `0d` – for representing decimal numbers
- `0h` – for representing hexadecimal numbers

Signed radix-specified integer literals have their sign before the leading encoding character.

The following string-encoded integer literals all have the value `42`:

```
0b101010
0o52
0d42
0h2a
```

The following string-encoded integer literals all have the value `-42`:

```
-0b101010
-0o52
-0d42
-0h2a
```

Radix-specified integer literals are only usable when constructing hardware integer literals. Any use in place of an integer is disallowed.

28 Grammar

```
(* Circuit Definition *)
circuit =
  version , newline ,
  "circuit" , id , ":" , [ annotations ] , [ info ] , newline , indent ,
    { decl } ,
  dedent ;

(* Top-level Declarations *)
decl =
    decl_module
  | decl_extmodule
  | decl_layer
  | decl_formal
  | decl_type_alias ;

decl_module =
  [ "public" ], "module" , id , { enablelayer } , ":" , [ info ] ,
    newline , indent ,
    { port , newline } ,
    { statement , newline } ,
  dedent ;

decl_extmodule =
  "extmodule" , id , ":" , [ info ] , newline , indent ,
    { port , newline } ,
    [ "defname" , "=" , id , newline ] ,
    { "parameter" , id , "=" , type_param , newline } ,
  dedent ;

decl_layer =
  "layer" , id , string , ":" , [ info ] , newline , indent ,
  { decl_layer , newline } ,
  dedent ;

decl_formal =
  "formal" , id , "of" , id , ":" , [ info ] , newline , indent ,
    { id , "=" , decl_formal_param , newline } ,
  dedent ;
decl_formal_param =
    int
  | string_dq
  | string_sq
  | "[" , [ decl_formal_param , { "," , decl_formal_param } ] , "]"
```

```
  | "{" , [ id , "=" , decl_formal_param , { "," , id , "=" , decl_formal_param } ] , "}

decl_type_alias = "type", id, "=", type ;

port = ( "input" | "output" ) , id , ":" , (type | type_property) , [ info ] ;
type_param = int | string_dq | string_sq ;
type_property = "Integer" | "List" , "<" , type_property , ">";

(* Statements *)
statement =
    circuit_component
  | connectlike
  | conditional
  | command
  | layerblock
  | skip ;

(* Circuit Components *)
circuit_component =
    circuit_component_node
  | circuit_component_wire
  | circuit_component_reg
  | circuit_component_inst
  | circuit_component_mem ;

circuit_component_node = "node" , id , "=" , expr , [ info ] ;
circuit_component_wire = "wire" , id , ":" , type , [ info ] ;
circuit_component_inst = "inst" , id , "of" , id , [ info ] ;

circuit_component_reg =
    "reg" , id , ":" , type , "," , expr , [ info ]
  | "regreset" , id , ":" , type , "," , expr , "," , expr , "," , expr , [info] ;

circuit_component_mem =
  "mem" , id , ":" , [ info ] , newline , indent ,
    "data-type" , "=>" , type , newline ,
    "depth" , "=>" , int , newline ,
    "read-latency" , "=>" , int , newline ,
    "write-latency" , "=>" , int , newline ,
    "read-under-write" , "=>" , read_under_write , newline ,
    { "reader" , "=>" , id , newline } ,
    { "writer" , "=>" , id , newline } ,
    { "readwriter" , "=>" , id , newline } ,
  dedent ;
```

```
read_under_write =  "old" | "new" | "undefined" ;

(* Connect-like Statements *)
connectlike =
    "connect" , reference , "," , expr , [ info ]
  | "invalidate" , reference , [ info ]
  | "attach" , "(" , reference , { "," ,  reference } , ")" , [ info ]
  | "define" , reference_static , "=" , expr_probe , [ info ]
  | "propassign" , reference_static , "," , property_expr , [ info ] ;

(* Conditional Statements *)
conditional =
    conditional_when
  | conditional_match ;

conditional_when =
  "when" , expr , ":" [ info ] , newline ,
    indent , statement, { statement } , dedent ,
  [ "else" , ":" , indent , statement, { statement } , dedent ] ;

conditional_match =
  "match" , expr , ":" , [ info ] , newline ,
  [ indent , { conditional_match_branch } , dedent ] ;

conditional_match_branch =
  id , [ "(" , id , ")" ] , ":" , newline ,
  [ indent , { statement } , dedent ] ;

(* Command Statements *)
command =
    "stop" , "(" , expr , "," , expr , "," , int , ")" , [ info ]
  | "force" , "(" , expr , "," , expr , "," , expr_probe , "," , expr , ")"
  | "force_initial" , "(" , expr_probe , "," , expr , ")"
  | "release" , "(" , expr , "," , expr , "," , expr_probe , ")"
  | "release_initial" , "(" , expr_probe , ")"
  | expr_intrinsic , [ info ]
  | "printf" , "(" ,
        expr , "," ,
        expr , "," ,
        string_dq ,
        { "," , expr }
    , ")" ,
    [ ":" , id ] , [ info ]
  | "assert" , "(" ,
        expr , "," ,
```

```
        expr , "," ,
        expr , "," ,
        string_dq ,
        { "," , expr }
    , ")" ,
    [ ":" , id ] , [ info ]
  | "assume" , "(" ,
        expr , "," ,
        expr , "," ,
        expr , "," ,
        string_dq ,
        { "," , expr }
    , ")" ,
    [ ":" , id ] , [ info ]
  | "cover" , "(" ,
        expr , "," ,
        expr , "," ,
        expr , "," ,
        string_dq
    , ")" ,
    [ ":" , id ] , [ info ] ;

(* Layer Block Statement *)
layerblock =
  "layerblock" , id , "of" , id , ":" , [ info ] , newline , indent ,
    { port , newline } ,
    { statement , newline } ,
  dedent ;

(* Skip Statement *)
skip = "skip" , [ info ] ;

(* References *)
reference =
    reference_static
  | reference_dynamic ;

reference_static =
    id
  | reference_static , "." , id
  | reference_static , "[" , int , "]" ;

reference_dynamic =
    reference , "[" , expr , "]" ;
```

```
(* Expressions *)
expr =
    expr_reference
  | expr_lit
  | expr_enum
  | expr_mux
  | expr_read
  | expr_primop
  | expr_intrinsic ;

expr_reference = reference ;
expr_lit = ( "UInt" | "SInt" ) , [ width ] , "(" , ( int | rint ) , ")" ;
expr_enum = type_enum , "(" , id , [ "," , expr ] , ")" ;
expr_mux = "mux" , "(" , expr , "," , expr , "," , expr , ")" ;
expr_read = "read" , "(" , expr_probe , ")" ;

expr_probe =
  "probe" , "(" , reference_static , ")"
  "rwprobe" , "(" , reference_static , ")"
  | reference_static ;

property_literal_expr = "Integer", "(", int, ")" ;
property_expr = reference_static | property_literal_expr | property_expr_primop ;
property_expr_primop = property_primop_2expr | property_primop_varexpr;
expr_primop = primop_2expr | primop_1expr | primop_1expr1int | primop_1expr2int ;

expr_intrinsic = "intrinsic", "(" , id ,
  [ "<"    , id , "=" , ( int | string_dq ) ,
    { "," , id , "=" , ( int | string_dq ) } , ">" ] ,
  [ ":" , type ] ,
  { "," , expr } , ")"

(* Types *)
type = ( [ "const" ] , type_hardware ) | type_probe ;

type_hardware =
    type_ground
  | type_bundle
  | type_vec
  | type_enum
  | id ;

(* Ground Types *)
type_ground =  type_ground_nowidth | type_ground_width ;
```

```
type_ground_nowidth =
    "Clock"
  | "Reset"
  | "AsyncReset" ;

type_ground_width =
    "UInt" , [ width ]
  | "SInt" , [ width ]
  | "Analog" , [ width ] ;

width = "<" , int , ">" ;

(* Bundle Types *)
type_bundle = "{" , type_bundle_field , { type_bundle_field } , "}" ;
type_bundle_field = [ "flip" ] , id , ":" , type ;

(* Vec Types *)
type_vec = type , "[" , int , "]" ;

(* Enum Types *)
type_enum = "{|" , { type_enum_alt } , "|}" ;
type_enum_alt = id, [ ":" , type_constable ] ;

(* Probe Types *)
type_probe = ( "Probe" | "RWProbe" ) , "<", type , [ "," , id ] ">" ;

(* Primitive Operations *)
primop_2expr     = primop_2expr_keyword , "(" , expr , "," , expr ")" ;
primop_1expr     = primop_1expr_keyword , "(" , expr , ")" ;
primop_1expr1int = primop_1expr1int_keyword , "(", expr , "," , int , ")" ;
primop_1expr2int = primop_1expr2int_keyword , "(" , expr , "," , int , "," , int , ")" ;

(* Primitive Property Operations *)
property_primop_2expr = property_primop_2expr_keyword ,
                        "(" , property_expr , "," , property_expr ")" ;
property_primop_varexpr = property_primop_varexpr_keyword ,
                          "(" , { property_expr } , ")" ;

(* Enable Layers *)
enablelayer = "enablelayer" , id , { "." , id } ;

(* Tokens: Annotations *)
annotations = "%" , "[" , json_array , "]" ;

(* Tokens: Version *)
```

```
sem_ver = int , "."  , int , "." , int ;
version = "FIRRTL" , "version" , sem_ver ;

(* Tokens: Whitespace *)
indent = " " , { " " } ;
dedent = ? remove one level of indentation ? ;
newline = ? a newline character ? ;

(* Tokens: Integer Literals *)
int = [ "-" ] , digit_dec , { digit_dec } ;
digit_bin = "0" | "1" ;
digit_oct = digit_bin | "2" | "3" | "4" | "5" | "6" | "7" ;
digit_dec = digit_oct | "8" | "9" ;
digit_hex = digit_dec
          | "A" | "B" | "C" | "D" | "E" | "F"
          | "a" | "b" | "c" | "d" | "e" | "f" ;

(* Tokens: Radix-specified Integer Literals *)
rint =
    [ "-" ] , "0b" , digit_bin , { digit_bin }
  | [ "-" ] , "0o" , digit_oct , { digit_oct }
  | [ "-" ] , "0d" , digit_oct , { digit_dec }
  | [ "-" ] , "0h" , digit_hex , { digit_hex } ;

(* Tokens: String Literals *)
string = ? a string ? ;
string_dq = '"' , string , '"' ;
string_sq = "'" , string , "'" ;

(* Tokens: Identifiers *)
id = ( "_" | letter ) , { "_" | letter | digit_dec } | literal_id ;
literal_id = "`" , ( "_" | letter | digit_dec ), { "_" | letter | digit_dec } , "`" ;
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z"
       | "a" | "b" | "c" | "d" | "e" | "f" | "g"
       | "h" | "i" | "j" | "k" | "l" | "m" | "n"
       | "o" | "p" | "q" | "r" | "s" | "t" | "u"
       | "v" | "w" | "x" | "y" | "z" ;

(* Tokens: Fileinfo *)
info = "@" , "[" , lineinfo, { ",", lineinfo }, "]" ;
lineinfo = string, " ", linecol ;
linecol = digit_dec , { digit_dec } , ":" , digit_dec , { digit_dec } ;
```

```
(* Tokens: PrimOp Keywords *)
primop_1expr_keyword =
    "asUInt" | "asSInt" | "asClock" | "asAsyncReset" | "cvt"
  | "neg"    | "not"
  | "andr"   | "orr"    | "xorr" ;

primop_2expr_keyword =
    "add"  | "sub" | "mul" | "div" | "rem"
  | "lt"   | "leq" | "gt"  | "geq" | "eq" | "neq"
  | "dshl" | "dshr"
  | "and"  | "or"  | "xor" | "cat" ;

primop_1expr1int_keyword =
    "pad" | "shl" | "shr" | "head" | "tail" ;

primop_1expr2int_keyword = "bits" ;

property_primop_2expr_keyword =
    "integer_add" | "integer_mul" | "integer_shr" | "integer_shl" ;

property_primop_varexpr_keyword =
    "List" , "<" , type_property , ">" | "list_concat" ;
```

29 Versioning Scheme of this Document

This is the versioning scheme that applies to version 1.0.0 and later.

The versioning scheme complies with Semantic Versioning 2.0.0.

Specifically,

The PATCH digit is bumped upon release which only includes non-functional changes, such as grammar edits, further examples, and clarifications.

The MINOR digit is bumped for feature additions to the spec.

The MAJOR digit is bumped for backwards-incompatible changes such as features being removed from the spec, changing their interpretation, or new required features being added to the specification.

In other words, any `.fir` file that was compliant with `x.y.z` will be compliant with `x.Y.Z`, where `Y >= y`, `z` and `Z` can be any number.