Specification for the FIRRTL Language

Patrick S. Li
psli@eecs.berkeley.edu

Adam M. Izraelevitz
adamiz@eecs.berkeley.edu

Jonathan Bachrach
jrb@eecs.berkeley.edu

March 1, 2022

Contents

1 Introduction **4**
1.1 Background 4
1.2 Design Philosophy 5

2 Acknowledgments **5**

3 Circuits and Modules **6**
3.1 Circuits 6
3.2 Modules 6
3.3 Externally Defined Modules 7

4 Types **7**
4.1 Ground Types 7
4.1.1 Integer Types 7
4.1.2 Fixed-Point Number Type 8
4.1.3 Clock Type 9
4.1.4 Analog Type 9
4.2 Vector Types 9
4.3 Bundle Types 10
4.4 Passive Types 11
4.5 Type Equivalence 11
4.6 Weak Type Equivalence 11
4.6.1 Oriented Types 11
4.6.2 Conversion to Oriented Types 12
4.6.3 Oriented Type Equivalence 12

5 Statements **12**

5.1 Connects . . . 12
5.1.1 The Connection Algorithm . . . 13
5.2 Partial Connects . . . 13
5.2.1 The Partial Connection Algorithm . . . 14
5.3 Statement Groups . . . 15
5.3.1 Last Connect Semantics . . . 15
5.4 Empty . . . 16
5.5 Wires . . . 16
5.6 Registers . . . 17
5.7 Invalidates . . . 17
5.7.1 The Invalidate Algorithm . . . 18
5.8 Attaches . . . 18
5.9 Nodes . . . 18
5.10 Conditionals . . . 19
5.10.1 Syntactic Shorthands . . . 19
5.10.2 Nested Declarations . . . 21
5.10.3 Initialization Coverage . . . 22
5.10.4 Scoping . . . 22
5.10.5 Conditional Last Connect Semantics . . . 22
5.11 Memories . . . 24
5.11.1 Read Ports . . . 25
5.11.2 Write Ports . . . 25
5.11.3 Readwrite Ports . . . 26
5.11.4 Read Under Write Behaviour . . . 26
5.11.5 Write Under Write Behaviour . . . 27
5.12 Instances . . . 27
5.13 Stops . . . 27
5.14 Formatted Prints . . . 28
5.14.1 Format Strings . . . 28
5.15 Verification . . . 29
5.15.1 Assert . . . 29
5.15.2 Assume . . . 30
5.15.3 Cover . . . 30

6 Expressions 30
6.1 Unsigned Integers . . . 30
6.2 Unsigned Integers from Literal Bits . . . 31
6.3 Signed Integers . . . 31
6.4 Signed Integers from Literal Bits . . . 31
6.5 References . . . 32
6.6 Sub-fields . . . 32
6.7 Sub-indices . . . 32
6.8 Sub-accesses . . . 33
6.9 Multiplexers . . . 35
6.10 Conditionally Valids . . . 35

6.11 Primitive Operations 36

7 Primitive Operations 36
7.1 Add Operation 36
7.2 Subtract Operation 36
7.3 Multiply Operation 37
7.4 Divide Operation 37
7.5 Modulus Operation 37
7.6 Comparison Operations 38
7.7 Padding Operations 38
7.8 Interpret As UInt 38
7.9 Interpret As SInt 38
7.10 Interpret As Fixed-Point Number 39
7.11 Interpret as Clock 39
7.12 Shift Left Operation 39
7.13 Shift Right Operation 39
7.14 Dynamic Shift Left Operation 40
7.15 Dynamic Shift Right Operation 40
7.16 Arithmetic Convert to Signed Operation 40
7.17 Negate Operation 41
7.18 Bitwise Complement Operation 41
7.19 Binary Bitwise Operations 41
7.20 Bitwise Reduction Operations 41
7.21 Concatenate Operation 42
7.22 Bit Extraction Operation 42
7.23 Head 42
7.24 Tail 42
7.25 Fixed-Point Precision Modification Operations 43

8 Flows 43

9 Width Inference 43

10 Fixed-Point Math 44

11 Namespaces 45
11.1 Name Expansion Algorithm 45
11.2 Prefix Uniqueness 45

12 The Lowered FIRRTL Forms 46
12.1 MidFIRRTL 46
12.2 LoFIRRTL 46

13 Details about Syntax 47

14 FIRRTL Language Definition 50

1 Introduction

1.1 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.

1.2 Design Philosophy

FIRRTL represents the standardized elaborated circuit that the Chisel HDL produces. FIRRTL represents the circuit immediately after Chisel's elaboration but before any circuit simplification. 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, partial connects, and modules. These high-level constructs are then gradually removed by a sequence of *lowering* transformations. During each lowering transformation, the circuit is rewritten into an equivalent circuit using simpler, lower-level constructs. Eventually the circuit is simplified to its most restricted form, resembling a structured netlist, which allows for easy translation to an output language (e.g. Verilog). This form is given the name *lowered FIRRTL* (LoFIRRTL) and is a strict subset of the full FIRRTL language.

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

2 Acknowledgments

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

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

Adam Izraelevitz then reworked the design and re-implemented FIRRTL, and after many discussions with Patrick Li and the Chisel group, refined the design to its present version.

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

- Andrew Waterman: for his many contributions to the design of FIRRTL's constructs, for his work on Chisel 3.0, and for porting architecture research infrastructure
- Richard Lin: for improving the Chisel 3.0 code base for release quality
- Jack Koenig: for implementing the FIRRTL parser in Scala

- Henry Cook: for porting and cleaning up many aspects of Chisel 3.0, including the testing infrastructure and the parameterization library
- Chick Markley: for creating the new testing harness and porting the Chisel tutorial
- Stephen Twigg: for his expertise in hardware intermediate representations and for providing many corner cases to consider
- Palmer Dabbelt, Eric Love, Martin Maas, Christopher Celio, and Scott Beamer: for their feedback on previous drafts of the FIRRTL specification

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

This research was partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

3 Circuits and Modules

3.1 Circuits

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

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

3.2 Modules

Each module has a given name, a list of ports, and a statement representing the circuit connections within the module. A module port is specified by its , 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 5.1 for details on the connect statement.

```
module MyModule :
   input foo: UInt
   output bar: UInt
   bar <= foo
```

Note that a module definition does *not* indicate that the module will be physically present in the final circuit. Refer to the description of the instance statement for details on how to instantiate a module (Section 5.12).

3.3 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 and 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.

An example of an externally defined module is:

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

The widths of all externally defined module ports must be specified. Width inference, described in Section 9, is not supported for module ports.

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.

4 Types

Types are used to specify the structure of the data held by each circuit component. All types in FIRRTL are either one of the fundamental ground types or are built up from aggregating other types.

4.1 Ground Types

There are five ground types in FIRRTL: an unsigned integer type, a signed integer type, a fixed-point number type, a clock type, and an analog type.

4.1.1 Integer Types

Both unsigned and signed integer types may optionally be given a known non-negative integer bit width.

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

Alternatively, if the bit width is omitted, it will be automatically inferred by FIRRTL's width inferencer, as detailed in Section 9.

```
UInt
SInt
```

4.1.2 Fixed-Point Number Type

In general, a fixed-point binary number type represents a range of values corresponding with the range of some integral type scaled by a fixed power of two. In the FIRRTL language, the number represented by a signal of fixed-point type may expressed in terms of a base integer *value* term and a *binary point*, which represents an inverse power of two.

The range of the value term is governed by a *width* an a manner analogous to integral types, with the additional restriction that all fixed-point number types are inherently signed in FIRRTL. Whenever an operation such as a `cat` operates on the "bits" of a fixed-point number, it operates on the string of bits that is the signed representation of the integer value term. The *width* of a fixed-point typed signal is the width of this string of bits.

$$\begin{aligned}
\text{fixed-point quantity} &= (\text{integer value}) \times 2^{-(\text{binary point})} \\
\text{integer value} &\in \left[-2^{(\text{width})-1}, 2^{(\text{width})-1}\right) \\
\text{binary point} &\in \mathbb{Z}
\end{aligned}$$

In the above equation, the range of possible fixed-point quantities is governed by two parameters beyond a the particular "value" assigned to a signal: the width and the binary point. Note that when the binary point is positive, it is equivalent to the number of bits that would fall after the binary point. Just as width is a parameter of integer types in FIRRTL, width and binary point are both parameters of the fixed-point type.

When declaring a component with fixed-point number type, it is possible to leave the width and/or the binary point unspecified. The unspecified parameters will be inferred to be sufficient to hold the results of all expressions that may drive the component. Similar to how width inference for integer types depends on width-propagation rules for each FIRRTL expression and each kind of primitive operator, fixed-point parameter inference depends on a set of rules outlined throughout this spec.

Included below are examples of the syntax for all possible combinations of specified and inferred fixed-point type parameters.

```
Fixed<3><<2>>     ; 3-bit width, 2 bits after binary point
Fixed<10>         ; 10-bit width, inferred binary point
Fixed<<-4>>       ; Inferred width, binary point of -4
Fixed             ; Inferred width and binary point
```

4.1.3 Clock Type

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

The clock type is specified as follows:

```
Clock
```

4.1.4 Analog Type

The analog type specifies that a wire or port can be attached to multiple drivers. `Analog` cannot be used as the type of a node or register, nor can it be used as 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; however, the partial connection statement will `attach` corresponding analog fields of its operands according to the partial connection algorithm described in Section 5.2.1.

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
Analog     ; analog type with inferred width
```

4.2 Vector Types

A vector type is used to express an ordered sequence of elements of a given type. The length of the sequence must be non-negative and known.

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

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

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

```
UInt[10]
```

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

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

4.3 Bundle Types

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

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

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

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

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

In a connection between circuit components with bundle types, the data carried by the flipped fields flow in the opposite direction as the data carried by the non-flipped fields.

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

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

In a connection to the `a` port, the data carried by the `word` and `valid` sub-fields will flow out of the module, while data carried by the `ready` sub-field will flow into the module. More details about how the bundle field orientation affects connections are explained in Section 5.1.

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

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

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

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

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

4.4 Passive Types

It is inappropriate for some circuit components to be declared with a type that allows for data to flow in both directions. For example, all sub-elements in a memory should flow in the same direction. These components are restricted to only have a passive type.

Intuitively, a passive type is a type where all data flows in the same direction, and is defined to be a type that recursively contains no fields with flipped orientations. Thus all ground types are passive types. Vector types are passive if their element type is passive. And bundle types are passive if no fields are flipped and if all field types are passive.

4.5 Type Equivalence

The type equivalence relation is used to determine whether a connection between two components is legal. See Section 5.1 for further details about connect statements.

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

A fixed-point number type is always equivalent to another fixed-point number type, regardless of width or binary point. It is not equivalent to any other type.

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

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

Two bundle types are equivalent 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}}`.

4.6 Weak Type Equivalence

The weak type equivalence relation is used to determine whether a partial connection between two components is legal. See Section 5.2 for further details about partial connect statements.

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

4.6.1 Oriented Types

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

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

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

4.6.2 Conversion to Oriented Types

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

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

To convert a bundle field to an oriented field, convert its type to an oriented type, apply the field orientation, and combine this with the original field's name to create the oriented field. To convert a bundle type to an oriented bundle type, convert each field to an oriented field.

4.6.3 Oriented Type Equivalence

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

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

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

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

5 Statements

Statements are used to describe the components within a module and how they interact.

5.1 Connects

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

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

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

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

1. The types of the left-hand and right-hand side expressions must be equivalent (see Section 4.5 for details).
2. The bit widths of the two expressions must allow for data to always flow from a smaller bit width to an equal size or larger bit width.
3. The flow of the left-hand side expression must be sink or duplex (see Section 8 for an explanation of flow).
4. Either the flow of the right-hand side expression is source or duplex, or the right-hand side expression has a passive type.

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 behaviour of connect statements between two circuit components with aggregate types is defined by the connection algorithm in Section 5.1.1.

5.1.1 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.

5.2 Partial Connects

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

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

1. The types of the left-hand and right-hand side expressions must be weakly equivalent (see Section 4.6 for details).
2. The flow of the left-hand side expression must be sink or duplex (see Section 8 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.

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

Partial connect statements from a wider ground type component to a narrower ground type component will have its value automatically truncated to fit the smaller bit width.

Intuitively, bundle fields with matching names will be connected appropriately, while bundle fields not present in both types will be ignored. Similarly, vectors with mismatched lengths will be connected up to the shorter length, and the remaining sub-elements are ignored. The full algorithm is detailed in Section 5.2.1.

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

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

The above example is equivalent to the following:

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

For details on the syntax and semantics of the sub-field expression, sub-index expression, and statement groups, see Sections 6.6, 6.7, 5.3.

5.2.1 The Partial Connection Algorithm

A partial connect statement between two non-analog ground type components connects the right-hand side expression to the left-hand side expression. Conversely, a *reverse* partial connect statement between two non-analog ground type components connects the left-hand side expression to the right-hand side expression. A partial connect statement between two analog-typed components performs an attach between the two signals.

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

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

5.3 Statement Groups

An ordered sequence of one or more statements can be grouped into a single statement, called a statement group. The following example demonstrates a statement group composed of three connect statements.

```
module MyModule :
   input a: UInt
   input b: UInt
   output myport1: UInt
   output myport2: UInt
   myport1 <= a
   myport1 <= b
   myport2 <= a
```

5.3.1 Last Connect Semantics

Ordering of statements is significant in a statement group. Intuitively, during elaboration, statements execute in order, and the effects of later statements take precedence over earlier ones. In the previous example, in the resultant circuit, port `b` will be connected to `myport1`, and port `a` will be connected to `myport2`.

Note that connect and partial connect statements have equal priority, and later connect or partial connect statements always take priority over earlier connect or partial connect statements. Conditional statements are also affected by last connect semantics, and for details see Section 5.10.5.

In the case where a connection to a circuit component with an aggregate type is followed by a connection to a sub-element of that component, only the connection to the sub-element is overwritten. Connections to the other sub-elements remain unaffected. In the following example, in the resultant circuit, 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`.

```
module MyModule :
   input portx: {b: UInt, c: UInt}
   input porty: UInt
   output myport: {b: UInt, c: UInt}
   myport <= portx
   myport.b <= porty
```

The above circuit can be rewritten equivalently as follows.

```
module MyModule :
   input portx: {b: UInt, c: UInt}
   input porty: UInt
   output myport: {b: UInt, c: UInt}
   myport.b <= porty
   myport.c <= portx.c
```

In the case where a connection to a sub-element of an aggregate circuit component is followed by a connection to the entire circuit component, the later connection overwrites the earlier connections completely.

```
module MyModule :
   input portx: {b: UInt, c: UInt}
   input porty: UInt
   output myport: {b: UInt, c: UInt}
   myport.b <= porty
   myport <= portx
```

The above circuit can be rewritten equivalently as follows.

```
module MyModule :
   input portx: {b: UInt, c: UInt}
   input porty: UInt
   output myport: {b: UInt, c: UInt}
   myport <= portx
```

See Section 6.6 for more details about sub-field expressions.

5.4 Empty

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:

```
a <= b
skip
c <= d
```

can be equivalently expressed as:

```
a <= b
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 5.10 for details on the conditional statement.

5.5 Wires

A wire is a named combinational circuit component that can be connected to and from using connect and partial connect statements.

The following example demonstrates instantiating a wire with the given name `mywire` and type `UInt`.

```
wire mywire: UInt
```

5.6 Registers

A register is a named stateful circuit component.

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
; ...
```

Optionally, for the purposes of circuit initialization, a register can be declared with a reset signal and value. 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
reg myreg: SInt, myclock with: (reset => (myreset, myinit))
; ...
```

Note that the clock signal for a register must be of type `clock`, the reset signal must be a single bit `UInt`, and the type of initialization value must match the declared type of the register.

5.7 Invalidates

An invalidate statement is used to indicate that a circuit component contains indeterminate values. It is specified as follows:

```
wire w: UInt
w is invalid
```

Invalidate statements can be applied to any circuit component of any type. However, if the circuit component cannot be connected to, then the statement has no effect on the component. This allows the invalidate statement to be applied to any component, to explicitly ignore initialization coverage errors.

The following example demonstrates the effect of invalidating a variety of circuit components with aggregate types. See Section 5.7.1 for details on the algorithm for determining what is invalidated.

```
module MyModule :
   input in: {flip a: UInt, b: UInt}
   output out: {flip a: UInt, b: UInt}
   wire w: {flip a: UInt, b: UInt}
   in is invalid
   out is invalid
   w is invalid
```

is equivalent to the following:

```
module MyModule :
   input in: {flip a: UInt, b: UInt}
   output out: {flip a: UInt, b: UInt}
   wire w: {flip a: UInt, b: UInt}
   in.a is invalid
   out.b is invalid
   w.a is invalid
   w.b is invalid
```

For the purposes of simulation, invalidated components are initialized to random values, and operations involving indeterminate values produce undefined behaviour. This is useful for early detection of errors in simulation.

5.7.1 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). 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.

5.8 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
```

When signals of aggregate types that contain analog-typed fields are used as operators of a partial connection, corresponding fields of analog type are attached, rather than connected.

5.9 Nodes

A node is simply a named intermediate value in a circuit. The node must be initialized to a value with a passive type and cannot be connected to. Nodes are often used to split a complicated compound expression into named sub-expressions.

The following example demonstrates instantiating a node with the given name mynode initialized with the output of a multiplexer (see Section 6.9).

```
wire pred: UInt<1>
wire a: SInt
wire b: SInt
node mynode = mux(pred, a, b)
```

5.10 Conditionals

Connections within a conditional statement that connect to previously declared components hold only when the given condition is high. The condition must have a 1-bit unsigned integer type.

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.

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      x <= a
   else :
      x <= b
```

5.10.1 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:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      x <= a
```

can be equivalently expressed as:

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   wire x: UInt
   when en :
      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:

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

can be equivalently written as:

```
module MyModule :
   input a: UInt
   input b: UInt
   input c: UInt
   input d: UInt
   input c1: UInt<1>
   input c2: UInt<1>
   input c3: UInt<1>
   wire x: UInt
   when c1 :
      x <= a
   else when c2 :
      x <= b
   else when c3 :
      x <= c
   else :
      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 :
   a <= b
else :
   e <= f
```

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

```
when c : a <= b else :
  e <= f
```

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

```
when c : a <= b else : e <= f
```

5.10.2 Nested Declarations

If a component is declared within a conditional statement, connections to the component are unaffected by the condition. In the following example, register `myreg1` is always connected to `a`, and register `myreg2` is always connected to `b`.

```
module MyModule :
   input a: UInt
   input b: UInt
   input en: UInt<1>
   input clk : Clock
   when en :
      reg myreg1 : UInt, clk
      myreg1 <= a
   else :
      reg myreg2 : UInt, clk
      myreg2 <= b
```

Intuitively, a line can be drawn between a connection (or partial connection) to a component and that component's declaration. All conditional statements that are crossed by the line apply to that connection (or partial connection).

5.10.3 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.

```
module MyModule :
   input en: UInt<1>
   input a: UInt
   wire w: UInt
   when en :
      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.

5.10.4 Scoping

The conditional statement creates a new *scope* within each of its `when` and `else` branches. It is an error to refer to any component declared within a branch after the branch has ended. As mention in Section 11, circuit component declarations in a module must be unique within the module's flat namespace; this means that shadowing a component in an enclosing scope with a component of the same name inside a conditional statement is not allowed.

5.10.5 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 6.9.

The following example:

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

can be rewritten equivalently using a multiplexer as follows:

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

In the case where an invalid statement is followed by a conditional statement containing a connect to the invalidated component, the resulting connection to the component can be expressed using a conditionally valid expression. See Section 6.10 for more details about the conditionally valid expression.

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
w is invalid
when c :
   w <= a
```

can be rewritten equivalently as follows:

```
wire a: UInt
wire c: UInt<1>
wire w: UInt
w <= validif(c, a)
```

The behaviour 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 Sections 5.1.1, 5.2.1 for the connection and partial connection algorithms) 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}
w <= x
when c :
   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}
w.a <= mux(c, y.a, x.a)
w.b <= mux(c, y.b, x.b)
```

Similar to the behavior of aggregate types under last connect semantics (see Section 5.3.1), 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}
w <= x
when c :
   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}
w.a <= mux(c, y, x.a)
w.b <= x.b
```

5.11 Memories

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

1. A passive type representing the type of each element in the memory.
2. A positive integer 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 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 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 behaviour when a memory location is written to while a read to that location is in progress.

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.

5.11.1 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.

5.11.2 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.

5.11.3 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.

5.11.4 Read Under Write Behaviour

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.

5.11.5 Write Under Write Behaviour

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

5.12 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
      b <= a
   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.

5.13 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
```

5.14 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 5.13.

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.

5.14.1 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

5.15 Verification

To facilitate simulation, model checking and formal methods, there are three non-synthesizable verification statements available: assert, assume and cover. Each type of verification statement requires a clock signal, a predicate signal, an enable signal and an explanatory message string literal. The predicate and enable signals must have single bit unsigned integer type. When an assert 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 5.13.

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.

5.15.1 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.

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

5.15.2 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.

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

5.15.3 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 enable implies predicate true at some time step.

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

6 Expressions

FIRRTL expressions are used for creating literal unsigned and signed integers, for referring to a declared circuit component, for statically and dynamically accessing a nested element within a component, for creating multiplexers and conditionally valid signals, and for performing primitive operations.

6.1 Unsigned Integers

A literal unsigned integer can be created given a non-negative integer value and an optional positive bit width. The following example creates a 10-bit unsigned integer representing the number 42.

```
UInt<10>(42)
```

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.

```
UInt(42)
```

6.2 Unsigned Integers from Literal Bits

A literal unsigned integer can alternatively be created given a string representing its bit representation and an optional bit width.

The following radices are supported:

1. `b` : For representing binary numbers.
2. `o` : For representing octal numbers.
3. `h` : For representing hexadecimal numbers.

If a bit width is not given, the number of bits in the bit representation is directly represented by the string. The following examples create a 8-bit integer representing the number 13.

```
UInt("b00001101")
UInt("h0D")
```

If the provided bit width is larger than the number of bits required to represent the string's value, then the resulting value is equivalent to the string zero-extended up to the provided bit width. If the provided bit width is smaller than the number of bits represented by the string, then the resulting value is equivalent to the string truncated down to the provided bit width. All truncated bits must be zero.

The following examples create a 7-bit integer representing the number 13.

```
UInt<7>("b00001101")
UInt<7>("o015")
UInt<7>("hD")
```

6.3 Signed Integers

Similar to unsigned integers, a literal signed integer can be created given an integer value and an optional positive bit width. The following example creates a 10-bit unsigned integer representing the number -42.

```
SInt<10>(-42)
```

Note that it is an error to supply a bit width that is not large enough to fit the given value using two's complement representation. If the bit width is omitted, then the minimum number of bits necessary to fit the given value will be inferred.

```
SInt(-42)
```

6.4 Signed Integers from Literal Bits

Similar to unsigned integers, a literal signed integer can alternatively be created given a string representing its bit representation and an optional bit width.

The bit representation contains a binary, octal or hex indicator, followed by an optional sign, followed by the value.

If a bit width is not given, the number of bits in the bit representation is the minimal bitwidth to represent the value represented by the string. The following examples create a 8-bit integer representing the number -13. For all bases, a negative sign acts as if it were a unary negation; in other words, a negative literal produces the additive inverse of the unsigned interpretation of the digit pattern.

```
SInt("b-1101")
SInt("h-d")
```

If the provided bit width is larger than the number of bits represented by the string, then the resulting value is unchanged. It is an error to provide a bit width smaller than the number of bits required to represent the string's value.

6.5 References

A reference is simply a name that refers to a previously declared circuit component. It may refer to a module port, node, wire, register, instance, or memory.

The following example connects a reference expression `in`, referring to the previously declared port `in`, to the reference expression `out`, referring to the previously declared port `out`.

```
module MyModule :
   input in: UInt
   output out: UInt
   out <= in
```

In the rest of the document, for brevity, the names of components will be used to refer to a reference expression to that component. Thus, the above example will be rewritten as "the port `in` is connected to the port `out`".

6.6 Sub-fields

The sub-field expression refers to a sub-element of an expression with a bundle type.

The following example connects the `in` port to the `a` sub-element of the `out` port.

```
module MyModule :
   input in: UInt
   output out: {a: UInt, b: UInt}
   out.a <= in
```

6.7 Sub-indices

The sub-index expression statically refers, by index, to a sub-element of an expression with a vector type. The index must be a non-negative integer and cannot be equal to or exceed the length of the vector it indexes.

The following example connects the `in` port to the fifth sub-element of the `out` port.

```
module MyModule :
   input in: UInt
   output out: UInt[10]
   out[4] <= in
```

6.8 Sub-accesses

The sub-access expression dynamically refers to a sub-element of a vector-typed expression using a calculated index. The index must be an expression with an unsigned integer type.

The following example connects the n'th sub-element of the `in` port to the `out` port.

```
module MyModule :
   input in: UInt[3]
   input n: UInt<2>
   output out: UInt
   out <= in[n]
```

A connection from a sub-access expression can be modeled by conditionally connecting from every sub-element in the vector, where the condition holds when the dynamic index is equal to the sub-element's static index.

```
module MyModule :
   input in: UInt[3]
   input n: UInt<2>
   output out: UInt
   when eq(n, UInt(0)) :
      out <= in[0]
   else when eq(n, UInt(1)) :
      out <= in[1]
   else when eq(n, UInt(2)) :
      out <= in[2]
   else :
      out is invalid
```

The following example connects the `in` port to the n'th sub-element of the `out` port. All other sub-elements of the `out` port are connected from the corresponding sub-elements of the `default` port.

```
module MyModule :
   input in: UInt
   input default: UInt[3]
   input n: UInt<2>
   output out: UInt[3]
   out <= default
   out[n] <= in
```

A connection to a sub-access expression can be modeled by conditionally connecting to every sub-element in the vector, where the condition holds when the dynamic index is equal to the sub-element's static index.

```
module MyModule :
   input in: UInt
   input default: UInt[3]
   input n: UInt<2>
   output out: UInt[3]
   out <= default
   when eq(n, UInt(0)) :
      out[0] <= in
   else when eq(n, UInt(1)) :
      out[1] <= in
   else when eq(n, UInt(2)) :
      out[2] <= in
```

The following example connects the `in` port to the m'th `UInt` sub-element of the n'th vector-typed sub-element of the `out` port. All other sub-elements of the `out` port are connected from the corresponding sub-elements of the `default` port.

```
module MyModule :
   input in: UInt
   input default: UInt[2][2]
   input n: UInt<1>
   input m: UInt<1>
   output out: UInt[2][2]
   out <= default
   out[n][m] <= in
```

A connection to an expression containing multiple nested sub-access expressions can also be modeled by conditionally connecting to every sub-element in the expression. However the condition holds only when all dynamic indices are equal to all of the sub-element's static indices.

```
module MyModule :
   input in: UInt
   input default: UInt[2][2]
   input n: UInt<1>
   input m: UInt<1>
   output out: UInt[2][2]
   out <= default
   when and(eq(n, UInt(0)), eq(m, UInt(0))) :
      out[0][0] <= in
   else when and(eq(n, UInt(0)), eq(m, UInt(1))) :
      out[0][1] <= in
   else when and(eq(n, UInt(1)), eq(m, UInt(0))) :
      out[1][0] <= in
   else when and(eq(n, UInt(1)), eq(m, UInt(1))) :
      out[1][1] <= in
```

6.9 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.

```
module MyModule :
   input a: UInt
   input b: UInt
   input sel: UInt<1>
   output c: UInt
   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. One-bit
    2. Unspecified, but will infer to one-bit
3. The types of the two input expressions are equivalent.
4. The types of the two input expressions are passive (see Section 4.4).

6.10 Conditionally Valids

A conditionally valid expression is expressed as an input expression guarded with an unsigned single bit valid signal. It outputs the input expression when the valid signal is high, otherwise the result is undefined.

The following example connects the `a` port to the `c` port when the `valid` signal is high. Otherwise, the value of the `c` port is undefined.

```
module MyModule :
   input a: UInt
   input valid: UInt<1>
   output c: UInt
   c <= validif(valid, a)
```

A conditionally valid expression is legal only if the following holds.

1. The type of the valid signal is a single bit unsigned integer.
2. The type of the input expression is passive (see Section 4.4).

Conditional statements can be equivalently expressed as multiplexers and conditionally valid expressions. See Section 5.10 for details.

6.11 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 static 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 6.11 will describe the format and semantics of each primitive operation.

7 Primitive Operations

The arguments of all primitive operations must be expressions with ground types, while their parameters are static integer literals. Each specific operation can place additional restrictions on the number and types of their arguments and parameters.

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

7.1 Add Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
add	(e1,e2)	()	(UInt,UInt)	UInt	$\max(w_{e1},w_{e2})+1$
			(SInt,SInt)	SInt	$\max(w_{e1},w_{e2})+1$
			(Fixed,Fixed)	Fixed	see Section 10

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

7.2 Subtract Operation

Name	Arguments	Parmaeters	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
			(Fixed,Fixed)	Fixed	see Section 10

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

7.3 Multiply Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
mul	(e1,e2)	()	(UInt,UInt)	UInt	w_{e1}+w_{e2}
			(SInt,SInt)	SInt	w_{e1}+w_{e2}
			(Fixed,Fixed)	Fixed	see Section 10

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

7.4 Divide Operation

Name	Arguments	Parmaeters	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.

7.5 Modulus Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
rem	(num,den)	()	(UInt,UInt)	UInt	min(w_{num},w_{den})
			(SInt,SInt)	SInt	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))}
```

7.6 Comparison Operations

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
lt,leq			(UInt,UInt)	UInt	1
gt,geq	(e1,e2)	()	(SInt,SInt)	UInt	1
eq,neq			(Fixed,Fixed)	UInt	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.

7.7 Padding Operations

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
pad	(e)	(n)	(UInt)	UInt	max(w_e,n)
			(SInt)	SInt	max(w_e,n)
			(Fixed)	Fixed	see Section 10

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. The binary point of fixed-point values is not affected by padding.

7.8 Interpret As UInt

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
asUInt	(e)	()	(UInt)	UInt	w_e
			(SInt)	UInt	w_e
			(Fixed)	UInt	w_e
			(Clock)	UInt	1

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

7.9 Interpret As SInt

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
asSInt	(e)	()	(UInt)	SInt	w_e
			(SInt)	SInt	w_e
			(Fixed)	SInt	w_e
			(Clock)	SInt	1

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

7.10 Interpret As Fixed-Point Number

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width	Result Binary Point
asFixed	(e)	(p)	(UInt)	Fixed	w_e	p
			(SInt)	Fixed	w_e	p
			(Fixed)	Fixed	w_e	p
			(Clock)	Fixed	1	p

The interpret as fixed-point operation reinterprets e's bits as a fixed-point number of identical width. Since all fixed-point number in FIRRTL are signed, the bits are taken to mean a signed value according to two's complement representation. They are scaled by the provided binary point p, and the result type has binary point p.

7.11 Interpret as Clock

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
asClock	(e)	()	(UInt)	Clock	n/a
			(SInt)	Clock	n/a
			(Fixed)	Clock	n/a
			(Clock)	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.

7.12 Shift Left Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
shl	(e)	(n)	(UInt)	UInt	w_e+n
			(SInt)	SInt	w_e+n
			(Fixed)	Fixed	see Section 10

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

7.13 Shift Right Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
shr	(e)	(n)	(UInt)	UInt	max(w_e-n, 1)
			(SInt)	SInt	max(w_e-n, 1)
			(Fixed)	Fixed	see Section 10

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.

7.14 Dynamic Shift Left Operation

Name	Arguments	Parmaeters	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
			(Fixed, UInt)	Fixed	see Section 10

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.

7.15 Dynamic Shift Right Operation

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
dshr	(e1, e2)	()	(UInt, UInt)	UInt	w_{e1}
			(SInt, UInt)	SInt	w_{e1}
			(Fixed, UInt)	Fixed	see Section 10

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.

7.16 Arithmetic Convert to Signed Operation

Name	Arguments	Parmaeters	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.

7.17 Negate Operation

Name	Arguments	Parmaeters	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.

7.18 Bitwise Complement Operation

Name	Arguments	Parmaeters	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.

7.19 Binary Bitwise Operations

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

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.

7.20 Bitwise Reduction Operations

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

7.21 Concatenate Operation

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

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

7.22 Bit Extraction Operation

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

7.23 Head

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
head	(e)	(n)	(UInt)	UInt	n
			(SInt)	UInt	n
			(Fixed)	UInt	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.

7.24 Tail

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

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
			(SInt)	UInt	w_e-n
			(Fixed)	UInt	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.

7.25 Fixed-Point Precision Modification Operations

Name	Arguments	Parmaeters	Arg Types	Result Type	Result Width
incp, decp, setp	(e)	(n)	(Fixed)	Fixed	

The increase precision, decrease precision, and set precision operations are used to alter the number of bits that appear after the binary point in a fixed-point number. This will cause the binary point and consequently the total width of the fixed-point result type to differ from those of the fixed-point argument type. See Section 10 for more detail.

8 Flows

An expression's flow partially determines the legality of connecting to and from the expression. Every expression is classified as either *source*, *sink*, or *duplex*. For details on connection rules refer back to Sections 5.1, 5.2.

The flow of a reference to a declared circuit component depends on the kind of circuit component. A reference to an input port, an instance, a memory, and a node, is a source. A reference to an output port is a sink. A reference to a wire or register is duplex.

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.

9 Width Inference

For all circuit components declared with unspecified widths, the FIRRTL compiler will infer the minimum possible width that maintains the legality of all its incoming connections. If a component has no incoming connections, and the width is unspecified, then an error is thrown to indicate that the width could not be inferred.

For module input ports with unspecified widths, the inferred width is the minimum possible width that maintains the legality of all incoming connections to all instantiations of the module.

The width of a ground-typed multiplexer expression is the maximum of its two corresponding input widths. For multiplexing aggregate-typed expressions, the resulting widths of each leaf sub-element is the maximum of its corresponding two input leaf sub-element widths.

The width of a conditionally valid expression is the width of its input expression.

The width of each primitive operation is detailed in Section 6.11.

The width of the integer literal expressions is detailed in their respective sections.

10 Fixed-Point Math

Table 26: Propagation rules for binary primitive operators that operate on two fixed-point numbers. Here, w_{e1} and p_{e1} are used to indicate the width and binary point of the first operand, while w_{e2} and p_{e2} are used to indicate the width and binary point of the second operand.

Operator	Result Width	Result Binary Point
add(e1, e2)	$\max(w_{e1}\text{-}p_{e1}, w_{e2}\text{-}p_{e2}) + \max(p_{e1}, p_{e2}) + 1$	$\max(p_{e1}, p_{e2})$
mul(e1, e2)	$w_1 + w_2$	$p_1 + p_2$

Table 27: Propagation rules for binary primitive operators that modify the width and/or precision of a single fixed-point number using a constant integer literal parameter. Here, w_e and p_e are used to indicate the width and binary point of the fixed-point operand, while `n` is used to represent the value of the constant parameter.

Operator	Result Width	Result Binary Point
pad(e, n)	$\max(w_e, n)$	p_e
shl(e, n)	$w_e + n$	p_e
shr(e, n)	$\max(w_e - n, \max(1, p_e))$	p_e
incp(e, n)	$w_e + n$	$p_e + n$
decp(e, n)	$w_e - n$	$p_e - n$
setp(e, n)	$w_e - p_e + n$	n

Table 28: Propagation rules for dynamic shifts on fixed-point numbers. These take a fixed-point argument and an UInt argument. Here, w_{e1} and p_{e1} are used to indicate the width and binary point of the fixed-point operand, while w_{e1} is used to represent the width of the unsigned integer operand. Note that the actual shift amount will be the dynamic value of the `e2` argument.

Operator	Result Width	Result Binary Point
dshl(e1, e1)	w_{e1} + 2^w_{e2} - 1	p_e
dshr(e1, e2)	w_{e1}	p_e

11 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. Furthermore, the set of component declarations within a module must be *prefix unique*. Please see Section 11.2 for the definition of prefix uniqueness.

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

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

During the lowering transformation, all circuit component declarations with aggregate types are rewritten as a group of component declarations, each with a ground type. The name expansion algorithm in Section 11.1 calculates the names of all replacement components derived from the original aggregate-typed component.

After the lowering transformation, the names of the lowered circuit components are guaranteed by the name expansion algorithm and thus can be reliably referenced by users to pair meta-data or other annotations with named circuit components.

11.1 Name Expansion Algorithm

Given a component with a ground type, the name of the component is returned.

Given a component with a vector type, the suffix `$`i is appended to the expanded names of each sub-element, where i is the index of each sub-element.

Given a component with a bundle type, the suffix `$`f is appended to the expanded names of each sub-element, where f is the field name of each sub-element.

11.2 Prefix Uniqueness

The *symbol sequence* of a name is the ordered list of strings that results from splitting the name at each occurrence of the '$' character.

A symbol sequence a is a *prefix* of another symbol sequence b if the strings in a occur in the beginning of b.

A set of names are defined to be *prefix unique* if there exists no two names such that the symbol sequence of one is a prefix of the symbol sequence of the other.

As an example `firetruck$y$z` shares a prefix with `firetruck$y` and `firetruck`, but does not share a prefix with `fire`.

12 The Lowered FIRRTL Forms

The lowered FIRRTL forms, MidFIRRTL and LoFIRRTL, are increasingly restrictive subsets of the FIRRTL language that omit many of the higher level constructs. All conforming FIRRTL compilers must provide a *lowering transformation* that transforms arbitrary FIRRTL circuits into equivalent LoFIRRTL circuits. However, there are no additional requirements related to accepting or producing MidFIRRTL, as the LoFIRRTL output of the lowering transformation will already be a legal subset of MidFIRRTL.

12.1 MidFIRRTL

A FIRRTL circuit is defined to be a valid MidFIRRTL circuit if it obeys the following restrictions:

- All widths must be explicitly defined.
- The conditional statement is not used.
- The dynamic sub-access expression is not used.
- All components are connected to exactly once.

12.2 LoFIRRTL

A FIRRTL circuit is defined to be a valid LoFIRRTL circuit if it obeys the following restrictions:

- All widths must be explicitly defined.
- The conditional statement is not used.
- All components are connected to exactly once.
- All components must be declared with a ground type.
- The partial connect statement is not used.

The first three restrictions follow from the fact that any LoFIRRTL circuit is also a legal MidFIRRTL circuit. The additional restrictions give LoFIRRTL a direct correspondence to a circuit netlist.

Low level circuit transformations can be conveniently written by first lowering a circuit to its LoFIRRTL form, then operating on the restricted (and thus simpler) subset of constructs. Note that circuit transformations are still free to generate high level constructs as they can simply be lowered again.

The following module:

```
module MyModule :
   input in: {a: UInt<1>, b: UInt<2>[3]}
   input clk: Clock
   output out: UInt
   wire c: UInt
   c <= in.a
   reg r: UInt[3], clk
   r <= in.b
   when c :
      r[1] <= in.a
   out <= r[0]
```

is rewritten as the following equivalent LoFIRRTL circuit by the lowering transform.

```
module MyModule :
   input in$a: UInt<1>
   input in$b$0: UInt<2>
   input in$b$1: UInt<2>
   input in$b$2: UInt<2>
   input clk: Clock
   output out: UInt<2>
   wire c: UInt<1>
   c <= in$a
   reg r$0: UInt<2>, clk
   reg r$1: UInt<2>, clk
   reg r$2: UInt<2>, clk
   r$0 <= in$b$0
   r$1 <= mux(c, in$a, in$b$1)
   r$2 <= in$b$2
   out <= r$0
```

13 Details about Syntax

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

The following characters are allowed in identifiers: upper and lower case letters, digits, and `_`. Identifiers cannot begin with a digit.

An integer literal in FIRRTL begins with one of the following, where '#' represents a digit between 0 and 9.

- 'h' : For indicating a hexadecimal number, followed by an optional sign. The rest of the literal must consist of either digits or a letter between 'A' and 'F'.
- 'o' : For indicating an octal number, followed by an optional sign. The rest of the literal must consist of digits between 0 and 7.
- 'b' : For indicating a binary number, followed by an optional sign. The rest of the literal must consist of digits that are either 0 or 1.
- '-#' : For indicating a negative decimal number. The rest of the literal must consist of digits between 0 and 9.
- '#' : For indicating a positive decimal number. The rest of the literal must consist of digits between 0 and 9.

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

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 :
    module Foo :
      skip
    module Bar :
     input a: UInt<1>
     output b: UInt<1>
     when a:
         b <= a
     else:
       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]
   module Top : @[myfile.txt 15:2]
     output out: UInt @[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 @[myfile.txt 21:8]
     when c : @[myfile.txt 24:8]
       a <= b @[myfile.txt 27:16]
     else :
       a <= d @[myfile.txt 29:17]
     out <= add(a,a) @[myfile.txt 34:4]
```

14 FIRRTL Language Definition

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

(* Integer literal definitions  *)
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" ;
(* An integer *)
int = '"' , "b" , [ "-" ] , { digit_bin } , '"'
    | '"' , "o" , [ "-" ] , { digit_oct } , '"'
    | '"' , "h" , [ "-" ] , { digit_hex } , '"'
    |             [ "-" ] , { digit_bin } ;

(* Identifiers define legal FIRRTL or Verilog names *)
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" ;
id = ( "_" | letter ) , { "_" | letter | digit_dec } ;

(* Fileinfo communicates Chisel source file and line/column info *)
linecol = digit_dec , { digit_dec } , ":" , digit_dec , { digit_dec } ;
info = "@" , "[" , { string , " " , linecol } , "]" ;

(* Type definitions *)
width = "<" , int , ">" ;
binarypoint = "<<" , int , ">>" ;
type_ground = "Clock"
            | ( "UInt" | "SInt" | "Analog" ) , [ width ]
            | "Fixed" , [ width ] , [ binarypoint ] ;
type_aggregate = "{" , field , { field } , "}"
               | type , "[" , int , "]" ;
field = [ "flip" ] , id , ":" , type ;
type = type_ground | type_aggregate ;
```

```
(* Primitive operations *)
primop_2expr_keyword =
    "add"   | "sub" | "mul" | "div" | "mod"
  | "lt"    | "leq" | "gt"  | "geq" | "eq" | "neq"
  | "dshl" | "dshr"
  | "and"   | "or"  | "xor" | "cat" ;
primop_2expr =
    primop_2expr_keyword , "(" , expr , "," , expr ")" ;
primop_1expr_keyword =
    "asUInt" | "asSInt" | "asClock" | "cvt"
  | "neg"    | "not"
  | "andr"   | "orr"    | "xorr"
  | "head"   | "tail" ;
primop_1expr =
    primop_1expr_keyword , "(" , expr , ")" ;
primop_1expr1int_keyword =
    "pad" | "shl" | "shr" ;
primop_1expr1int =
    primop_1exrp1int_keywork , "(", expr , "," , int , ")" ;
primop_1expr2int_keyword =
    "bits" ;
primop_1expr2int =
    primop_1expr2int_keywork , "(" , expr , "," , int , "," , int , ")" ;
primop = primop_2expr | primop_1expr | primop_1expr1int | primop_1expr2int ;

(* Expression definitions *)
expr =
    ( "UInt" | "SInt" ) , [ width ] , "(" , ( int ) , ")"
  | reference
  | "mux" , "(" , expr , "," , expr , "," , expr , ")"
  | "validif" , "(" , expr , "," , expr , ")"
  | primop ;
reference = id
          | reference , "." , id
          | reference , "[" , int , "]"
          | reference , "[" , expr , "]" ;

(* Memory *)
ruw = ( "old" | "new" | "undefined" ) ;
memory = "mem" , id , ":" , [ info ] , newline , indent ,
           "data-type" , "=>" , type , newline ,
           "depth" , "=>" , int , newline ,
           "read-latency" , "=>" , int , newline ,
           "write-latency" , "=>" , int , newline ,
           "read-under-write" , "=>" , ruw , newline ,
           { "reader" , "=>" , id , newline } ,
           { "writer" , "=>" , id , newline } ,
           { "readwriter" , "=>" , id , newline } ,
         dedent ;

(* Statements *)
statement = "wire" , id , ":" , type , [ info ]
          | "reg" , id , ":" , type , expr ,
            [ "(with: {reset => (" , expr , "," , expr ")})" ] , [ info ]
          | memory
          | "inst" , id , "of" , id , [ info ]
          | "node" , id , "=" , expr , [ info ]
          | reference , "<=" , expr , [ info ]
          | reference , "<-" , expr , [ info ]
          | reference , "is invalid" , [ info ]
          | "attach(" , { reference } , ")" , [ info ]
          | "when" , expr , ":" [ info ] , newline , indent ,
              { statement } ,
            dedent , [ "else" , ":" , indent , { statement } , dedent ]
          | "stop(" , expr , "," , expr , "," , int , ")" , [ info ]
          | "printf(" , expr , "," , expr , "," , string ,
            { expr } , ")" , [ ":" , id ] , [ info ]
          | "skip" , [ info ] ;

(* Module definitions *)
port = ( "input" | "output" ) , id , ":": , type , [ info ] ;
module = "module" , id , ":" , [ info ] , newline , indent ,
           { port , newline } ,
           { statement , newline } ,
         dedent ;
extmodule = "extmodule" , id , ":" , [ info ] , newline , indent ,
              { port , newline } ,
              [ "defname" , "=" , id , newline ] ,
              { "parameter" , "=" , ( string | int ) , newline } ,
            dedent ;

(* Circuit definition *)
circuit = "circuit" , id , ":" , [ info ] , newline , indent ,
            { module | extmodule } ,
          dedent ;
```