Specification for the FIRRTL Language: Version 0.1.1

Patrick S. Li, Adam M. Izraelevitz, Jonathan Bachrach

June 5, 2015

1 FIRRTL Language Definition

1.1 Abstract Syntax Tree

circuit	=	**circuit** id **:** (_module*_)	Circuit
module	=	[_info_] **module** id **:** (_port* stmt_)	Module
	|	[_info_] **exmodule** id **:** (_port*_)	External Module
port	=	[_info_] _dir_ id **:** _type_	Port
dir	=	**input**|**output**	Input/Output
type	=	**UInt** < _width_ >	Unsigned Integer
	|	**SInt** < _width_ >	Signed Integer
	|	{_field*_ }	Bundle
	|	_type_[int]	Vector
field	=	_orientation_ id **:** _type_	Bundle Field
orientation	=	**default**|**reverse**	Orientation
width	=	int	Known Integer Width
	|	**?**	Unknown Width
atype	=	**read**|**write**|**unknown**	Accessor Type
stmt	=	[_info_] **wire** id **:** _type_	Wire Declaration
	|	[_info_] **reg** id **:** _type_	Register Declaration
	|	[_info_] **mem** id **:** _type_	Memory Declaration
	|	[_info_] **inst** id **:** id	Instance Declaration
	|	[_info_] **node** id = _exp_	Node Declaration
	|	[_info_] _atype_ **accessor** id = _exp_[_exp_]	Accessor Declaration
	|	[_info_] _exp_ **:=** _exp_	Connect
	|	[_info_] **on-reset** _exp_ **:=** _exp_	On Reset
	|	[_info_] **when** _exp_ **:** _stmt_ **else** **:** _stmt_	Conditional
	|	[_info_] (_stmt*_)	Statement Group
	|	[_info_] **skip**	Empty Statement
exp	=	[_info_] **UInt** < _width_ > (_ints_)	Literal Unsigned Integer
	|	[_info_] **SInt** < _width_ > (_ints_)	Literal Signed Integer
	|	[_info_] id	Reference
	|	[_info_] _exp_.id	Subfield
	|	[_info_] _exp_[int]	Subindex
	|	[_info_] **Register**(_exp_, _exp_)	Structural Register
	|	[_info_] **WritePort**(id , _exp_, _exp_)	Write Port
	|	[_info_] **ReadPort**(id , _exp_, _exp_)	Read Port

	|	[*info*] *primop*(*exp**, int*)	Primitive Operation
info	=	filename **:** line.col	File Location
	|	**noinfo**	No File Location

primop	=		
		add	Unsigned/Signed Add
	|	**sub**	Unsigned/Signed Subtract
	|	**mul**	Unsigned/Signed Multiply
	|	**div**	Unsigned/Signed Divide
	|	**rem**	Unsigned/Signed Remainder
	|	**quo**	Unsigned/Signed Quotient
	|	**mod**	Unsigned/Signed Modulo
	|	**add-wrap**	Unsigned/Signed Add Wrap
	|	**sub-wrap**	Unsigned/Signed Subtract Wrap
	|	**lt**	Unsigned/Signed Less Than
	|	**leq**	Unsigned/Signed Less or Equal
	|	**gt**	Unsigned/Signed Greater Than
	|	**geq**	Unsigned/Signed Greater or Equal
	|	**eq**	Unsigned/Signed Equal
	|	**neq**	Unsigned/Signed Not-Equal
	|	**mux**	Unsigned/Signed Multiplex
	|	**pad**	Unsigned/Signed Pad to Length
	|	**asUInt**	Unsigned/Signed Reinterpret Bits as UInt
	|	**asSInt**	Unsigned/Signed Reinterpret Bits as SInt
	|	**shl**	Unsigned/Signed Shift Left
	|	**shr**	Unsigned/Signed Shift Right
	|	**toSInt**	Unsigned/Signed to Signed Conversion
	|	**neg**	Unsigned/Signed Negate
	|	**bit-not**	Unsigned Not
	|	**bit-and**	Unsigned And
	|	**bit-or**	Unsigned Or
	|	**bit-xor**	Unsigned Xor
	|	**bit-and-reduce**	Unsigned And
	|	**bit-or-reduce**	Unsigned Or
	|	**bit-xor-reduce**	Unsigned Xor
	|	**cat**	Unsigned Concatenation
	|	**bit**	Single Bit Extraction
	|	**bits**	Multiple Bit Extraction

1.2 Notation

The above definition specifies the structure of the abstract syntax tree corresponding to a FIRRTL circuit. Nodes in the abstract syntax tree are *italicized*. Keywords are shown in **bold**. The special productions, id and int, indicates an identifier and an integer literal respectively. Tokens followed by an asterisk, *e.g. field**, indicates a list formed from repeated occurences of the token.

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

2 Circuits and Modules

$$\begin{array}{rcl}
\textit{circuit} & = & \textbf{circuit} \text{ toplevel-module } \textbf{:} \text{ (modules*)} \\
\textit{module} & = & \textbf{module} \text{ name } \textbf{:} \text{ (ports* body)} \\
 & | & \textbf{exmodule} \text{ name } \textbf{:} \text{ (ports*)} \\
\textit{port} & = & \textit{dir} \text{ id } \textbf{:} \ \textit{type} \\
\textit{dir} & = & \textbf{input}|\textbf{output}
\end{array}$$

All FIRRTL circuits are comprised of a flat list of modules, each representing one hardware block. Each module has a given name, a list of ports, and a statement representing the circuit connections within the module. Externally defined modules consist of a given name, and a list of ports, whose types must match the types defined in the associated Verilog. Module names exist in their own namespace, and all modules must have a unique name. The name of the top-level module must be specified for a circuit.

A module port is specified by a direction, which may be input or output, a name, and the data type for the port. The port names exist in the identifier namespace for the module, and must be unique. In addition, all references within a module must be unique.

The special port name, *reset*, is used to carry the module reset signal for circuit initialization, and has special meaning. Circuit initialization is described in section 7.

3 Types

3.1 Ground Types

$$\begin{aligned} \textit{type} &= \textbf{UInt} < \textit{width} > \\ &\mid \textbf{SInt} < \textit{width} > \\ \textit{width} &= \text{int} \\ &\mid \textbf{?} \end{aligned}$$

There are only two ground types in FIRRTL, an unsigned and a signed integer type. Both of these types require a given bitwidth, which may be some known integer width, which must be non-negative, or an unknown width. Unknown widths are a declaration for the width to be computed by the FIRRTL width inferencer, instead of manually given by the programmer.

3.2 Vector Types

$$\textit{type} = \textit{type}[\text{int}]$$

Vector types in FIRRTL indicate a structure consisting of multiple elements of some given type. This is akin to array types in the C programming language. Note that the number of elements must be known, and non-negative.

As an example, the type **UInt** $< 16 >$ [10] indicates a ten element vector of 16-bit unsigned integers. The type **UInt** $<$ **?** $>$ [10] indicates a ten element vector of unsigned integers, with unknown but the same bitwidths.

Vector types may be nested ad infinitum. The type **UInt** $< 16 >$ [10][5] indicates a five element vector *of* ten element vectors of 16-bit unsigned integers.

3.3 Bundle Types

$$\begin{aligned} \textit{type} &= \{\textit{field*}\} \\ \textit{field} &= \textit{orientation}\ \text{name} : \textit{type} \\ \textit{orientation} &= \textbf{default}|\textbf{reverse} \end{aligned}$$

Bundle types in FIRRTL are composite types formed from an ordered sequence of named, nested types. All fields in a bundle must have a given direction, name, and type. The following is an example of a possible type for representing a complex number.

$$\{\textbf{default}\ \text{real} : \textbf{SInt} < 10 >, \textbf{default}\ \text{imag} : \textbf{SInt} < 10 >\}$$

It has two fields, real, and imag, both 10-bit signed integers. Here is an example of a possible type for a decoupled port.

$$\begin{gathered}\{\textbf{default}\ \text{data} : \textbf{UInt} < 10 >,\\ \textbf{default}\ \text{valid} : \textbf{UInt} < 1 >,\\ \textbf{reverse}\ \text{ready} : \textbf{UInt} < 1 >\}\end{gathered}$$

It has a data field that is specified to be a 10-bit unsigned integer, a valid signal that must be a 1-bit unsigned integer, and a flipped ready signal that must be a 1-bit unsigned integer.

By convention, we specify the directions within a bundle type with their relative orientation. For this reason, the real and imag fields for the complex number bundle type are both specified to be *default*. Similarly, if a module were to output a value using a decoupled protocol, we would declare the module to have an output port, data, which would contain the value itself, a non-flipped field, valid, which would indicate when the value is valid, and accept an *reverse* field, ready, from the receiving component, which would indicate when the component is ready to receive the value.

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

As in the case of vector types, bundle types may also be nested ad infinitum. I.e., the types of the fields themselves may also be bundle types, which will in turn contain more fields, etc.

4 Statements

FIRRTL circuit components are instantiated and connected together using *statements*.

4.1 Wires

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

$$\textbf{wire}\ \text{name} : \textit{type}$$

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

4.2 Registers

A register is a named stateful circuit element. A register with a given name and type can be instantiated with the following statement.

reg name **:** *type*

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

The statement *on-reset* is used to specify the initialization value for a register, and is described in section 7.

4.3 Memories

A memory is a stateful circuit element containing multiple elements. Unlike registers, memories can *only* be read from or written to through *accessors*. A memory with a given name and type can be instantiated with the following statement.

mem name **:** *type*

Note that, by definition, memories contain multiple elements, and hence *must* be declared with a vector type. It is an error to specify any other type for a memory. However, the internal type to the vector type may be a non-ground type, with the caveat that the internal type, if a bundle type, cannot contain any reverse fields. Additionally, the type for a memory must be completely specified and cannot contain any unknown widths.

4.4 Nodes

A node is simply a named intermediate value in a circuit. A node with a given name and value can be instantiated with the following statement.

node name $=$ *exp*

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

4.5 Accessors

Accessors are used for either connecting to or from a vector-typed expression, from some *variable* index. An accessor can be instantiated with the following statement.

$$\textit{atype}\ \textbf{accessor}\ \text{name} = \textit{exp}[\text{index}]$$
$$\textit{atype} \ = \ \textbf{read}|\textbf{write}|\textbf{unknown}$$

Given a name, an expression to access, the index at which to access, and optionally the accessor type, the above statement creates an accessor that may be used for connecting to or from the expression. The expression must have a vector type, and the index must be an variable with a UInt type.

An untyped (unknown) accessor or a read accessor may be used as outputs, by being on the right-hand side of a connect statement, in which case the accessor acts as a reader from the given expression at the given index. Or, an untyped accessor or a write accessor may be used as inputs, by being on the left-hand side of a connect statement, in which case the accessor acts as a writer to the given expression at the given index. An accessor must consistently be used either as an input, or as an output, but not as both.

The following example demonstrates using untyped accessors to read and write to a memory. The accessor, reader, acts as a memory read port that reads from the index specified by the wire i. The accessor, writer, acts as a memory write port that writes 42 to the index specified by wire j.

$$\begin{aligned}
&\textbf{wire}\ i : \textbf{UInt} < 5 > \\
&\textbf{wire}\ j : \textbf{UInt} < 5 > \\
&\textbf{mem}\ m : \textbf{UInt} < 10 > [10] \\
&\textbf{accessor}\ reader = m[i] \\
&\textbf{accessor}\ writer = m[j] \\
&writer := \textbf{UInt} <? > (42) \\
&\textbf{node}\ temp = reader
\end{aligned}$$

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

4.6 Instances

An instance refers to a particular instantiation of a FIRRTL module. An instance with some given name, of a given module can be created using the following statement.

inst name **of** module

The ports of an instance may be accessed using the subfield expression. The output ports of an instance may only be used in output positions, e.g. the right-hand side of a connect statement, and the input ports of an instance may only be used in input positions, e.g. the left-hand side of a connect statement.

An instance may be directly connected to a value. However, it can only be used as an input, or on the right side of a connect.

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

4.7 The Connect Statement

The connect statement is used to specify a physical wired connection between one hardware component to another, and is the most important statement in FIRRTL. The following statement is used to connect the output of some component, to the input of another component.

input := output

For a connection to be legal, the types of the two expressions must match exactly, including all field flips if the wires contain bundle types. The component on the right-hand side must be able to be used as an output, and the component on the left-hand side must be able to be used as an input.

4.8 The On Reset Statement

The on-reset statement is used to specify the default value for a register. Its semantics are described in Section 7.

$$\textbf{on-reset}\ \text{reg}\ \textbf{:=}\ \text{output}$$

For a connection to be legal, the types of the two expressions must match exactly, including all field flips if the wires contain bundle types. The component on the right-hand side must be able to be used as an output, and the component on the left-hand side must be able to be used as an input, and must be a reg.

4.9 The Conditional Statement

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

$$\textbf{when}\ \text{condition}\ \textbf{:}\ \text{conseq}\ \textbf{else}\ \textbf{:}\ \text{alt}$$

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

4.9.1 Initialization Coverage

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

$$\begin{array}{l}\textbf{wire}\ w : \textbf{UInt} < \textbf{?} > \\ \textbf{when}\ \mathit{enable} : \\ \quad w := \textbf{UInt} < \textbf{?} > (42)\end{array}$$

4.9.2 Scoping

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

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

4.10 Statement Groups

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

$$(stmt^*)$$

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

4.10.1 Last Connect Semantics

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

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

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

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

4.11 The Empty Statement

The empty statement is specified using the following.

skip

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

5 Expressions

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

5.1 Unsigned Integers

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

UInt < width > (value)

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

5.2 Signed Integers

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

SInt < width > (value)

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

5.3 References

$$\text{name}$$

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

5.4 Subfields

$$\textit{exp}.\text{name}$$

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

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

5.5 Subindex

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

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

5.6 Structural Register

$$\textbf{Register}(\text{value}, \text{enable})$$

A structural register is an unnamed register specified by the input value for the register and the enable signal for the register. The type of the input must be a ground type and the enable signal must be a 1-bit unsigned integer.

5.7 WritePort

$$\textbf{WritePort}(\text{mem}, \text{index}, \text{enable})$$

A write port is specified given the memory it accesses, the index into the memory, and the enable signal determining when to write the value. The index must be an expression with an unsigned integer type and the enable signal must be a 1-bit unsigned integer. The type of the WritePort is the inside type of the memory's vector type. A WritePort can only be used as an output (on the left side of a connect statement).

5.8 ReadPort

$$\textbf{ReadPort}(\text{mem}, \text{index}, \text{enable})$$

A read port is specified given the memory it accesses, the index into the memory, and the enable signal determining when to read the value. The index must be an expression with an unsigned integer type and the enable signal must be a 1-bit unsigned integer. The type of the ReadPort is the inside type of the memory's vector type. A ReadPort can only be used as an input (on the right side of a connect statement).

5.9 Primitive Operation

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

There are a number of different primitive operations supported by FIRRTL. Each operation takes some number of expressions, along with some number of integer literals. Section 6 will describe the format and semantics of each operation.

6 Primitive Operations

All primitive operations expression operands must be ground types. In addition, some allow all permutations of operand ground types, while others on allow subsets. When well defined, input arguments are allowed to be differing widths.

FAQ: Why have generic operations instead of explicit types (e.g. add-uu, where the two inputs are unsigned) FAQ: Why allow operations to allow inputs of differing widths?

6.1 Add Operation

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

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

6.2 Subtract Operation

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

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

6.3 Multiply Operation

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

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

6.4 Divide Operation

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

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

6.5 Modulus Operation

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

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

6.6 Quotient Operation

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

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

6.7 Remainder Operation

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

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

6.8 Add Wrap Operation

primop	Resultant Type	Resultant Width
add-wrap$(op1 : UInt, op2 : UInt)$	$UInt$	$max(width(op1), width(op2))$
add-wrap$(op1 : UInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$
add-wrap$(op1 : SInt, op2 : UInt)$	$SInt$	$max(width(op1), width(op2))$
add-wrap$(op1 : SInt, op2 : SInt)$	$SInt$	$max(width(op1), width(op2))$

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

6.9 Subtract Wrap Operation

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

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

6.10 Comparison Operations

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

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

6.11 Equality Comparison

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

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

If an arithmetic equals between a signed and unsigned integer is desired, one must first use convert on the unsigned integer, then use the equal primop.

6.12 Not-Equality Comparison

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

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

If an arithmetic not-equals between a signed and unsigned integer is desired, one must first use convert on the unsigned integer, then use the not-equal primop.

6.13 Multiplex

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

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

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

6.14 Padding Operation

primop	**Resultant Type**	**Resultant Width**
pad($op : UInt$, num)	$UInt$	*num*
pad($op : SInt$, num)	$SInt$	*num*

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

6.15 Reinterpret Bits as UInt

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

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

6.16 Reinterpret Bits as SInt

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

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

6.17 Shift Left Operation

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

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

6.18 Shift Right Operation

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

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

6.19 Dynamic Shift Left Operation

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

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

6.20 Dynamic Shift Right Operation

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

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

6.21 Convert to Signed

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

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

6.22 Negate

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

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

6.23 Bitwise Operations

primop	Resultant Type	Resultant Width
bit-not(*op1:UInt*)	$UInt$	$width(op1)$
bit-and(*op1:UInt*, *op2:UInt*)	$UInt$	$max(width(op1), width(op2))$
bit-or(*op1:UInt*, *op2:UInt*)	$UInt$	$max(width(op1), width(op2))$
bit-xor(*op1:UInt*, *op2:UInt*)	$UInt$	$max(width(op1), width(op2))$

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

6.24 Reduce Bitwise Operations

primop	Resultant Type	Resultant Width
bit-and-reduce(*op:UInt**)	$UInt$	$max(width(op)*)$
bit-or-reduce(*op:UInt**)	$UInt$	$max(width(op)*)$
bit-xor-reduce(*op:UInt**)	$UInt$	$max(width(op)*)$

The above operations correspond to bitwise not, and, or, and exclusive or respectively, reduced over a list of unsigned integers. The resultant width is equal to the width of the widest operand.

6.25 Concatenation

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

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

6.26 Bit Extraction Operation

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

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

6.27 Bit Range Extraction Operation

primop	Resultant Type	Resultant Width
$\textbf{bits}(op : UInt, \text{high}, \text{low})$	$UInt$	$high - low$
$\textbf{bits}(op : SInt, \text{high}, \text{low})$	$UInt$	$high - low$

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

7 Circuit Initialization

This section describes FIRRTL's facilities for expressing circuit initialization, and how it is customized through the reset port for modules, and the on-reset construct for registers.

7.1 Module Initialization

As stated before, all modules must have a port named reset defined.

7.2 Register Initialization

By default, a register will not have a initialization value and will not be enabled during reset. Hence, the default reset behavior for registers is to maintain their current value.

However, an explicit initialization value can be provided for a register by connecting an expression to the register via the on-reset construct. The following example demonstrates declaring a register, and changing its initialization value to forty two.

$$\begin{aligned}&\textbf{reg}\ r : \textbf{UInt} < 10 >\\ &on - resetr := \textbf{UInt} < \textbf{?} > (42)\end{aligned}$$

The type of the initialization value must match the declared type of the register. In the above example, the register, r, will be set to forty two when the circuit is reset. Note that structural registers cannot be assigned an initial value because they can only be used on the right side of a connect statement.

8 Lowered Form

To simplify the writing of transformation passes, FIRRTL provides a *lowering* pass, which rewrites any FIRRTL circuit into an equivalent *lowered form*. The lowered form has the advantage of not having any high-level constructs or composite types, and hence is a minimal representation that is convenient for low-level transforms.

In lowered form, every module has exactly the following form.

$$\begin{aligned}&\textbf{module}\ \text{name} :\\ &\quad port\ldots\\ &\quad \textbf{wire}\ \ldots\\ &\quad \text{connections to output ports}\ldots\end{aligned}$$

The body of the module must consist of a list of wire declarations, followed by a series of connect statements. The following restrictions also hold for modules in lowered form.

8.1 No Nested Expressions

In the declaration of the structural elements, the only nested expressions allowed are references, and unsigned and signed literals. All other nested expressions must be lifted to a named node, and referred to through a reference.

8.2 No Composite Types

No module port or wire may be declared with a bundle or vector type. The lowering pass will recursively expand ports into its constituent elements until all ports are declared with ground types.

8.3 Single Connect

Every wire declared can only be assigned to once within a module.

8.4 No Unknown Widths

No port or structural element may be declared with unknown width. The FIRRTL width inferencer will compute the widths for all structural elements and ports. If a width for some element cannot be calculated, then the lowering pass will fail with an error.

9 Inlined Lowered Form

A further (and optional) pass provided by FIRRTL is the inlining pass, which recursively inlines all instances in the top-level module until the top-level module is the only remaining module in the circuit. Inlined lowered form is essentially a flat netlist which specifies every component used in a circuit and their input connections.

10 Concrete Syntax

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

General Principles

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

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

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

Circuits and Modules

A circuit is specified the following way.

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

Or by taking advantage of indentation structuring:

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

A module is specified the following way.

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

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

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

The following shows an example of a simple module.

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

Types

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

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

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

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

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

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

Statements

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

```
wire mywire : UInt<10>
reg myreg : UInt<10>
mem mymem : UInt<10>[16]
inst myinst : MyModule
accessor myaccessor = e[i]
```

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

```
x := y
```

The on-reset statement is specified using the on-reset keyword and the `:=` operator.

```
on-reset x := y
```

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

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

Or by using indentation structuring:

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

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

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

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

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

Expressions

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

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

References are specified with a identifier.

```
x
```

Subfields are expressed using the dot operator.

```
x.data
```

Subindicies are expressed using the `[]` operator.

```
x[10]
```

Read ports are expressed using the ReadPort constructor.

```
ReadPort(m, index, enable)
```

Write ports are expressed using the WritePort constructor.

```
WritePort(m, index, enable)
```

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

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