Specification for the FIRRTL Language: Version 0.1.0

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

April 6, 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	=	_gender_ id **:** _type_	Bundle Field
gender	=	**female**|**male**	Gender
vptype	=	**read**|**write**|**rdwr**	Known Vector Port-Type
	|	**?**	Unknown Vector Port-Type
width	=	int	Known Integer Width
	|	**?**	Unknown Width
stmt	=	[_info_] **wire** id **:** _type_	Wire Declaration
	|	[_info_] **reg** id **:** _type_	Register Declaration
	|	[_info_] **mem** id **:** _type_	Memory Declaration
	|	[_info_] **inst** id **of** id	Instance Declaration
	|	[_info_] **node** id = _exp_	Node Declaration
	|	[_info_] _vptype_ **accessor** id = _exp_[_exp_]	Accessor Declaration
	|	[_info_] _exp_ **:=** _exp_	Connect
	|	[_info_] **when** _exp_ **:** _stmt_ **else** **:** _stmt_	Conditional
	|	[_info_] (_stmt_*)	Statement Group
	|	[_info_] **skip**	Empty Statement
exp	=	[_info_] **UInt**(int, _width_)	Literal Unsigned Integer
	|	[_info_] **SInt**(int, _width_)	Literal Signed Integer
	|	[_info_] id	Reference
	|	[_info_] _exp_.id	Subfield
	|	[_info_] _exp_.int	Subindex
	|	[_info_] **Register**(_exp_, _exp_)	Register
	|	[_info_] **WritePort**(id , _exp_, _exp_)	Write Port
	|	[_info_] **ReadPort**(id , _exp_, _exp_)	Read Port
	|	[_info_] **ReadWritePort**(id , _exp_, _exp_)	ReadWrite Port

	|	[*info*] *pad!*(*exp*, *width*)	Generic Pad to Width
	|	[*info*] *pad!-u*(*exp*, *width*)	Unsigned Pad to Width
	|	[*info*] *pad!-s*(*exp*, *width*)	Signed Pad to Width
	|	[*info*] *primop*(*exp**, int*)	Primitive Operation
info	=	filename **:** line.col	File Location
	|	**noinfo**	No File Location

primop	=		
		add|add-uu|add-us|add-su|add-ss	Unsigned/Signed Add
	|	**sub|sub-uu|sub-us|sub-su|sub-ss**	Unsigned/Signed Subtract
	|	**mul|mul-uu|mul-us|mul-su|mul-ss**	Unsigned/Signed Multiply
	|	**div|div-uu|div-us|div-su|div-ss**	Unsigned/Signed Divide
	|	**rem|rem-uu|rem-us|rem-su|rem-ss**	Unsigned/Signed Remainder
	|	**quo|quo-uu|quo-us|quo-su|quo-ss**	Unsigned/Signed Quotient
	|	**mod|mod-uu|mod-us|mod-su|mod-ss**	Unsigned/Signed Modulo
	|	**add-wrap|add-wrap-uu|add-wrap-us|add-wrap-su|add-wrap-ss**	Unsigned/Signed Add Wrap
	|	**sub-wrap|sub-wrap-uu|sub-wrap-us|sub-wrap-su|sub-wrap-ss**	Unsigned/Signed Subtract Wrap
	|	**lt|lt-uu|lt-us|lt-su|lt-ss**	Unsigned/Signed Less Than
	|	**leq|leq-uu|leq-us|leq-su|leq-ss**	Unsigned/Signed Less or Equal
	|	**gt|gt-uu|gt-us|gt-su|gt-ss**	Unsigned/Signed Greater Than
	|	**geq|geq-uu|geq-us|geq-su|geq-ss**	Unsigned/Signed Greater or Equal
	|	**equal|equal-uu|equal-ss**	Unsigned/Signed Equal
	|	**mux|mux-uu|mux-ss**	Unsigned/Signed Multiplex
	|	**pad|pad-u|pad-s**	Unsigned/Signed Pad to Length
	|	**asUInt|asUInt-u|asUInt-s**	Unsigned/Signed Reinterpret Bits as UInt
	|	**asSInt|asSInt-u|asSInt-s**	Unsigned/Signed Reinterpret Bits as SInt
	|	**shl|shl-u|shl-s**	Unsigned/Signed Shift Left
	|	**shr|shr-u|shr-s**	Unsigned/Signed Shift Right
	|	**convert|convert-u|convert-s**	Unsigned to Signed Conversion
	|	**and**	Unsigned And
	|	**or**	Unsigned Or
	|	**xor**	Unsigned Xor
	|	**concat**	Unsigned Concatenation
	|	**bit|bits**	Single/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 11.

2 Circuits and Modules

circuit	=	**circuit** toplevel-module **:** (modules*)
module	=	**module** name **:** (ports* body)
	|	**exmodule** name **:** (ports*)
port	=	*dir* id **:** *type*
dir	=	**input**|**output**

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.

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 8.

3 Types

3.1 Ground Types

$$\begin{array}{rcl} \textit{type} & = & \textbf{UInt}(\textit{width}) \\ & | & \textbf{SInt}(\textit{width}) \\ \textit{width} & = & \text{int} \\ & | & \textbf{?} \end{array}$$

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

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

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 and possibly differing 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

$$\begin{array}{rcl} \textit{type} & = & \{\textit{field*}\} \\ \textit{field} & = & \textit{dir}\ \text{name} \ \textbf{:}\ \textit{type} \end{array}$$

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.

$$\{\textbf{output}\ \text{real} \ \textbf{:}\ \textbf{SInt}(10), \textbf{output}\ \text{imag} \ \textbf{:}\ \textbf{SInt}(10)\}$$

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

$$\begin{aligned}\{\textbf{output}\ \text{data} &\mathbf{:}\ \textbf{UInt}(10),\\ \textbf{output}\ \text{valid} &\mathbf{:}\ \textbf{UInt}(1),\\ \textbf{input}\ \text{ready} &\mathbf{:}\ \textbf{UInt}(1)\}\end{aligned}$$

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 an *input* ready signal that must be a 1-bit unsigned integer.

By convention, we specify the directions within a bundle type consistently with how the fields would be defined if they were *output* ports for a module. For this reason, the real and imag fields for the complex number bundle type are both specified to have direction **output**. I.e., if a module were to output a complex number, we would declare it to have an output port real, and an output port imag. 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, an output port, valid, which would indicate when the value is valid, and accept an *input* port, 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. The special field name, *init*, is reserved for register initialization and is not available to the user. Register initialization is covered in section 8.

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.

wire name **:** *type*

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

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 distinguished field, *init*, is used to specify the initialization value for a register, and is described in section 8.

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. 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.

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.

$$\textbf{accessor}\ \text{name}\ \textbf{:}\ \textit{exp}[\text{index}]$$

Given a name, an expression to access, and the index at which to access, 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 unsigned integer.

Note that, though their directions are not explicitly specified, accessors are *not* bidirectional. They 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 they 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 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.

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. 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 Conditional Statement

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

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

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

4.8.1 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.

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

4.8.2 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.

4.9 Statement Groups

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

$$(\textit{stmt*}\ )$$

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.

4.9.1 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.

$$\begin{aligned}&\textbf{wire } w : \textbf{UInt}(\textbf{?})\\ &w := \textbf{UInt}(20, \textbf{?})\\ &w := \textbf{UInt}(42, \textbf{?})\end{aligned}$$

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.

$$\begin{aligned}&\textbf{wire } w : \textbf{UInt}(\textbf{?})\\ &w := \textbf{UInt}(20, \textbf{?})\\ &\textbf{when } \mathit{enable} :\\ &\quad w := \textbf{UInt}(42, \textbf{?})\end{aligned}$$

4.10 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.

4.11 The LetRec Statement

The letrec statement is used for declaring *structural* elements, and is an important part of *lowered form*. It is essentially a netlist representation of a piece of circuitry, but a representation that may exist and that operates correctly together with the other FIRRTL statements.

letrec : *elem** **in :** *stmt*

A letrec statement is defined by a given list of structural elements, which are described in the following section (section 5), and a body which contain statements that may refer to the structural elements.

Unlike in a statement group, the ordering of the structural elements have no meaning. The structural elements can be thought of as being declared *simultaneously*, and any structural element may refer to any other structural element, no matter the order in which they are declared.

[TODO: No combinational loops]

5 Structural Elements

Structural elements appear in letrec statements, and are simply circuit components whose input connections are fully specified upon declaration. They *cannot* be used as inputs, and hence can never be on the left-hand side of a connect statement.

5.1 Structural Register

$$\textbf{reg}\ \text{name}\ \textbf{:}\ \textit{type}\ = \textbf{Register}(\text{value}, \text{enable})$$

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

5.2 Structural Node

$$\textbf{node}\ \text{name}\ \textbf{:}\ \textit{type}\ = \textit{exp}$$

A structural node is specified given a name, a type, and its value. The type must be a ground type, and the given value must match the ground type.

5.3 Structural Memory

$$\textbf{mem}\ \text{name}\ \textbf{:}\ \textit{type}\ = \textbf{Memory}(\textbf{WritePort}(\text{index}, \text{value}, \text{enable})*)$$

A structural memory is specified given a name, a type, and a list of write ports. The given type must be a vector of some ground type. Each write port must specify the index at which to write the value, the value to write, and the enable signal determining when to write the value. The index must be an unsigned integer, the value must match the declared ground type of the memory, and the enable signal must be a 1-bit unsigned integer.

The only method of reading from a structural memory is through the **ReadPort** expression.

5.4 Structural Instance

inst name = module(**Input**(field, *exp*)*)

A structural instance is specified given the instance name, the name of the module, and a list of the expressions to connect to the instance's input ports. An expression must be specified for every one of the instance's input ports, and the expression's type must match the port type.

6 Expressions

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

6.1 Unsigned Integers

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

UInt(value, width)

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

6.2 Signed Integers

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

SInt(value, width)

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

6.3 References

name

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

6.4 Subfields

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

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

1. To refer to the initialization value of a register, using register-name.init.
2. To refer to a specific port of an instance, using instance-name.port-name.
3. To refer to a specific field within a bundle-typed expression.

6.5 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 structural and non-structural memories.

6.6 Structural Read Port

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

The structural read port expression is used to read from declared structural memories. A read port is created given the name of a declared structural memory, and the index at which to read from.

6.7 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 7 will describe the format and semantics of each operation.

7 Primitive Operations

7.1 Add Operation

primop	Resultant Type	Resultant Width
add(*op1*, *op2*)	$UInt \| SInt$	$max(width(op1), width(op2)) + 1$
add-uu(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2)) + 1$
add-us(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
add-su(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
add-ss(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$

FIRRTL supports generic, unsigned, and signed versions of the add operation. The generic version supports adding of any combination of two unsigned or signed integers. The other versions support only a specific permutation of operand types.

If the two operands differ in width, the operand of narrower width is automatically padded to be the same width as the wider operand. The resultant's value is 1-bit larger than the wider of the two operands and has a signed type if either operand is signed (otherwise is unsigned).

7.2 Subtract Operation

primop	Resultant Type	Resultant Width
sub(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
sub-uu(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
sub-us(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
sub-su(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$
sub-ss(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2)) + 1$

The subtraction operation works similarly to the add operation.

7.3 Multiply Operation

primop	Resultant Type	Resultant Width
mul(*op1*, *op2*)	$UInt \| SInt$	$width(op1) + width(op2)$
mul-uu(*op1*, *op2*)	$UInt$	$width(op1) + width(op2)$
mul-us(*op1*, *op2*)	$SInt$	$width(op1) + width(op2)$
mul-su(*op1*, *op2*)	$SInt$	$width(op1) + width(op2)$
mul-ss(*op1*, *op2*)	$SInt$	$width(op1) + width(op2)$

As with add and subtract, there is a generic, unsigned, and signed version of the multiply operation. The resultant value has width equal to the sum of the widths of its two operands.

7.4 Divide Operation

primop	Resultant Type	Resultant Width
div(*op1*, *op2*)	$UInt \| SInt$	$width(op1) \| width(op1) + 1$
div-uu(*op1*, *op2*)	$UInt$	$width(op1)$
div-us(*op1*, *op2*)	$SInt$	$width(op1) + 1$
div-su(*op1*, *op2*)	$SInt$	$width(op1)$
div-ss(*op1*, *op2*)	$SInt$	$width(op1) + 1$

The first argument is the dividend, the second argument is the divisor. The resultant value of a divide operation has width equal to the width of the dividend minus the width of the divisor.

7.5 Modulo Operation

primop	Resultant Type	Resultant Width
mod(*op1*, *op2*)	$UInt$	$width(op1) \| width(op2) - 1$
mod-uu(*op1*, *op2*)	$UInt$	$width(op2)$
mod-us(*op1*, *op2*)	$UInt$	$width(op2) - 1?$
mod-su(*op1*, *op2*)	$SInt$	$width(op2)$
mod-ss(*op1*, *op2*)	$SInt$	$width(op2) - 1?$

7.6 Quotient Operation

primop	Resultant Type	Resultant Width
quo(*op1*, *op2*)	$UInt \| SInt$	$width(?)$
quo-uu(*op1*, *op2*)	$UInt$	$width(?)$
quo-us(*op1*, *op2*)	$SInt$	$width(?)$
quo-su(*op1*, *op2*)	$SInt$	$width(?)$
quo-ss(*op1*, *op2*)	$SInt$	$width(?)$

The first argument is the dividend, the second argument is the divisor. The resultant value of a divide operation has width equal to the width of the dividend minus the width of the divisor.

7.7 Remainder Operation

primop	Resultant Type	Resultant Width
rem(*op1*, *op2*)	$UInt \| SInt$	$width(op2) \| min(width(op1), width(op2))$
rem-uu(*op1*, *op2*)	$UInt$	$min(width(op1), width(op2))$
rem-us(*op1*, *op2*)	$SInt$	$width(op2)$
rem-su(*op1*, *op2*)	$UInt$	$width(op2)$
rem-ss(*op1*, *op2*)	$SInt$	$width(op2)$

The first argument is the dividend, the second argument is the divisor. The resultant value of a modulo operation has the same width as the divisor.

7.8 Add Wrap Operation

primop	Resultant Type	Resultant Width
add-wrap(*op1*, *op2*)	$UInt \| SInt$	$max(width(op1), width(op2))$
add-wrap-uu(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2))$
add-wrap-us(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$
add-wrap-su(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$
add-wrap-ss(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$

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

7.9 Subtract Wrap Operation

primop	Resultant Type	Resultant Width
sub-wrap(*op1*, *op2*)	$UInt \| SInt$	$max(width(op1), width(op2))$
sub-wrap-uu(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2))$
sub-wrap-us(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$
sub-wrap-su(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$
sub-wrap-ss(*op1*, *op2*)	$SInt$	$max(width(op1), width(op2))$

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

7.10 Comparison Operations

primop	Resultant Type	Resultant Width
lt(*op1*, *op2*)	*UInt*	1
lt-uu(*op1*, *op2*)	*UInt*	1
lt-us(*op1*, *op2*)	*UInt*	1
lt-su(*op1*, *op2*)	*UInt*	1
lt-ss(*op1*, *op2*)	*UInt*	1
leq(*op1*, *op2*)	*UInt*	1
leq-uu(*op1*, *op2*)	*UInt*	1
leq-us(*op1*, *op2*)	*UInt*	1
leq-su(*op1*, *op2*)	*UInt*	1
leq-ss(*op1*, *op2*)	*UInt*	1
gt(*op1*, *op2*)	*UInt*	1
gt-uu(*op1*, *op2*)	*UInt*	1
gt-us(*op1*, *op2*)	*UInt*	1
gt-su(*op1*, *op2*)	*UInt*	1
gt-ss(*op1*, *op2*)	*UInt*	1
geq(*op1*, *op2*)	*UInt*	1
geq-uu(*op1*, *op2*)	*UInt*	1
geq-us(*op1*, *op2*)	*UInt*	1
geq-su(*op1*, *op2*)	*UInt*	1
geq-ss(*op1*, *op2*)	*UInt*	1

Generic, unsigned, and signed versions of the less than (**lt**), less than or equal to (**leq**), greater than (**gt**), and greater than or equal to (**geq**) comparison operations are provided. The generic versions of each operation accept either two unsigned integers, or two signed integers, but not a mixture of the two. The resultant value is a single-bit unsigned integer.

7.11 Equality Comparison

primop	Resultant Type	Resultant Width
equal(*op1*, *op2*)	*UInt*	1
equal-u(*op1*, *op2*)	*UInt*	1
equal-s(*op1*, *op2*)	*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.

The two operands may differ in width, in which case the narrower of the two operands is padded to the width of the larger operand.

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-s primop.

7.12 Multiplex

primop	**Resultant Type**	**Resultant Width**
mux(*condition*, *op1*, *op2*)	$UInt\|SInt$	$width(op1)$
mux-u(*condition*, *op1*, *op2*)	$UInt$	$width(op1)$
mux-s(*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.

This operation is the only primop where the two operands may not differ in width. The output is of the same width as the inputs.

7.13 Padding Operation

primop	**Resultant Type**	**Resultant Width**
pad(*op*, num)	$UInt\|SInt$	num
pad-u(*op*, num)	$UInt$	num
pad-s(*op*, num)	$SInt$	num

A generic, unsigned, and signed version of the pad operation is provided which pads some expression to a specified width. The unsigned pad operation zero-extends the given value until the specified width, while the signed pad operation sign-extends the given value. The generic operation performs either zero-extension or sign-extension depending on the type of its operand. The given width must be equal to or greater than the existing width of the expression.

7.14 Reinterpret Bits as UInt

primop	Resultant Type	Resultant Width
asUInt(*op1*)	*UInt*	*width*(*op*1)
asUInt-u(*op1*)	*UInt*	*width*(*op*1)
asUInt-s(*op1*)	*UInt*	*width*(*op*1)

A generic, unsigned and signed version of asUInt is provided. All versions return a UInt with the same width as the operand.

7.15 Reinterpret Bits as SInt

primop	Resultant Type	Resultant Width
asSInt(*op1*)	*SInt*	*width*(*op*1)
asSInt-u(*op1*)	*SInt*	*width*(*op*1)
asSInt-s(*op1*)	*SInt*	*width*(*op*1)

A generic, unsighed and signed version of asSInt is provided. All versions return a SInt with the same width as the operand.

7.16 Shift Left Operation

primop	Resultant Type	Resultant Width
shl	*UInt*|*SInt*	$width(op) + num$
shl-u	*UInt*	$width(op) + num$
shl-s	*SInt*	$width(op) + num$

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

7.17 Shift Right Operation

primop	Resultant Type	Resultant Width
shr	*UInt*|*SInt*	$width(op) - num$
shr-u	*UInt*	$width(op) - num$
shr-s	*SInt*	$width(op) - num$

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

7.18 Convert to Signed

primop	**Resultant Type**	**Resultant Width**
convert	$SInt$	$width(op) \mid width(op)+1$
convert-u	$SInt$	$width(op)+1$
convert-s	$SInt$	$width(op)$

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

7.19 Bitwise Operations

primop	**Resultant Type**	**Resultant Width**
and(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2))$
or(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2))$
xor(*op1*, *op2*)	$UInt$	$max(width(op1), width(op2))$

The above operations correspond to bitwise and, or, and exclusive or respectively. The operands must be unsigned integers, and the resultant width is equal to the width of the wider of the two operands.

7.20 Concatenation

primop	**Resultant Type**	**Resultant Width**
concat(*op1*, *op2*)	$UInt$	$width(op1) + width(op2)$

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

7.21 Bit Extraction Operation

primop	Resultant Type	Resultant Width
bit(op, 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.

7.22 Bit Range Extraction Operation

primop	Resultant Type	Resultant Width
bits(op, high, 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 $n = \text{high} - \text{low} + 1$.

8 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 init field for registers.

8.1 Register Initialization

On circuit reset, registers are automatically set to their *initialization value.* By default, a register's initialization value is set to be equal to the value of the register itself. 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's init field. The following example demonstrates declaring a register, and changing its initialization value to forty two.

$$\begin{aligned}&\textbf{reg}\ r : \textbf{UInt}(10)\\&r.\text{init} := \textbf{UInt}(42, \textbf{?})\end{aligned}$$

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 declared within letrec statements do not have an init field.

8.2 Module Reset Port

Every module has an input reset port declared to be a 1-bit unsigned integer. If the user chooses, the reset port may be explicitly declared. If the user does not explicitly declare a reset port, then FIRRTL will implicitly insert a reset port for the module.

When an instance of a module with an implicit reset is created, the reset signal of the module in which the instance was created is automatically connected to the instance's reset port. Consider the following example:

$$\begin{aligned}&\textbf{module}\ A :\\&\quad \textbf{input}\ x\ \textbf{:}\ \textbf{UInt}(\textbf{?})\\&\quad ...\\&\textbf{module}\ B :\\&\quad \textbf{inst}\ a\ \textbf{of}\ A\\&\quad a.\text{x}\ \textbf{:=}\ \textbf{UInt}(42, \textbf{?})\end{aligned}$$

The module A has an implicit reset port, and the module B creates an instance of A. In this case, there is an implicit connect statement inserted, connecting the instance's reset port to the implicit reset port of B.

$$\begin{aligned}&\textbf{inst}\ a\ \textbf{of}\ A\\&a.\text{reset}\ \textbf{:=}\ \text{reset}\\&\ldots\end{aligned}$$

The user may choose to explicitly override this default behavior by explicitly connecting an instance's reset port to a value.

```
module B :
   inst a of A
   a.reset := UInt(0, ?)
   a.x := UInt(42, ?)
```

The above code explicitly connects the instance's reset port to zero, indicating that its registers are *never* reset.

Should the user choose to explicitly declare the reset port for a module, then the port *must* be appropriately declared as an *input* port with a declared type of unsigned integer with width one. Additionally, if the user explicitly declares the reset port for a module, then all instances of that module must have its reset port connected explicitly. For example, the following code

```
module A :
   input x : UInt(?)
   input reset : UInt(1)
   ...
module B :
   inst a of A
   a.x := UInt(42, ?)
```

is not legal. The module, A, has an explicitly declared reset port, but the reset port for its instance, a, is not explicitly connected to any value.

9 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.

```
module name :
  port...
  letrec :
    elem...
  in :
    connections to output ports...
```

The body of the module must consist of a single letrec statement, in which every component within the module is declared as a structural element. The only statements that may appear in the body of the letrec statement are connect statements for each of the module's output ports. The following restrictions also hold for modules in lowered form.

9.1 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 structural node, and referred to through a reference.

9.2 No Composite Types

No module port 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. Additionally, by virtue of requiring all module components to be declared as structural elements, which support ground types only, there will be no composite types left in lowered form.

9.3 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.

9.4 Explicit Resets

All modules will contain an explicitly declared reset port and the structural instances within a module will explicitly specify the values connected to their reset ports.

10 Inlined Lowered Form

A further 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.

11 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.

```
{output data: UInt(10),
 output valid: UInt(1),
 input 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 of MyModule
accessor myaccessor = e[i]
```

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

```
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
```

Letrec statements are expressed using the `letrec` keyword. The following example demonstrates a letrec statement containing the four structural elements.

```
letrec :
   reg r : UInt(4) = Register(x, e)
   node n : UInt(?) = y
   mem m : UInt(4)[10] = Memory(WritePort(i, x, e), WritePort(j, x, e))
   inst i = MyAdder(Input(data, x), Input(valid, e))
in :
   y := x
```

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(42, 4)
SInt(-42, 4)
UInt(42, ?)
SInt(-42, ?)
```

References are specified with a identifier.

```
x
```

Subfields and subindexes are expressed using the dot operator.

```
x.data
x.10
```

Structural read ports are expressed using the ReadPort constructor.

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

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

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