Home » Tutorials » Lucid Reference

Lucid Reference

2025-02-04 · 40 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

This page is a reference for the Lucid V2 language.

Lucid File Contents

Every Lucid file (.luc extension) can contain module, testbench, and/or global declarations.

Modules

Modules are the core of any Lucid project. They are where you define a block of functionality.

A `module` declaration takes the following form.

```
module module_name #(
    // optional parameter list
)(
    // port list
) {
    // module body
}
```

Parameters

Parameters provide a way for a module to be customized when it is instantiated to improve code reuse.

All parameters must be constant as they are replaced during synthesis (build time).

Parameters are defined in list of comma separated parameter declarations between `#(` and `)`.

It is optional and can be completely omitted.

Each parameter declaration takes the following form.

```
PARAM_NAME = default_value : test_condition
```

or

```
PARAM_NAME ~ test_value : test_condition
```

Here, `PARAM_NAME` is the name of the parameter. Parameter names, like constants, must be made up of only capital letters and underscores.

Everything besides `PARAM_NAME` is optional.

In the first example, `= default_value`, will provide a default value for the parameter. If a default value is provided, when the module is instantiated the parameter can be omitted.

If you want to require a value to be provided when the module in instantiated then you can use the form in the second example of `~ test_value`. With this form, `test_value` is used as an example value to test your module in a stand-alone fashion by Alchitry Labs. However, it won't be used when the module is instantiated.

The `test_value` and `default_value` can be omitted but this will hinder the amount of error checking Alchitry Labs can perform on your module until it is instantiated.

The last piece, `: test_condition` provides a condition to test the parameter against. If it is false (evaluates to 0), then an error is thrown when the module is instantiated.

The `test_condition` can be any expression that evaluates to a number and references only this parameter or any previously declared parameters (ones that appear before this one in the list).

An example full declaration could look like this.

```
#(
    CLK_FREQ ~ 100000000 : CLK_FREQ > 0,              // clock frequency
    BAUD = 1000000 : BAUD > 0 && BAUD <= CLK_FREQ/4 // desired baud rate
)
```

If a parameter has a `default_value` provided, then any value assigned to the module during instantiation, must be compatible with the width of `default_value` .

For a width to be *compatible* all of its dimensions except the outermost dimension must match.

For example, if a parameter was declared as `PARAM = {8d1, 8d2, 8d3}` then any value assigned to `PARAM` must has the width of `[x][8]` where `x` can be anything.

If the parameter is a simple number, or `default_value` is omitted, then the width of the parameter is assumed to be a simple (1D) array.

Ports

Ports are how modules connect to the outside world.

They act similar to signals but have a direction associated with them.

Ports are defined in a comma separated list of port declarations between `(` and `)`.

Note that this differs from the parameter list in that opening symbol is simply `(` instead of `#(`.

Each port declaration takes the following form.

```
signed direction port_name port_size
```

`signed` optionally marks the port to be interpreted as signed.

The `direction` is one of `input`, `output`, or `inout`. The details of these are below.

The `port_name` is the name of the port and must start with a lowercase letter. It can then be followed by letters, numbers, and underscores.
It is convention for it to be `snake_case`.

The port can have an optional `port_size`. This follows the format defined in sizing. If it is omitted, then the port is a single bit wide.

input

Inputs are read-only signals passed into the module.

output

Outputs are write-only signals passed out of module.

Typically, they will have a value of `b0` or `b1`. However, if they connect directly to a top-level output (pin on the FPGA) they can also have the value `bz` meaning high-impedance (not driven).

Signals inside an FPGA don't have a mechanism for realizing `bz` so this can't be used internally.

inout

Inouts provide a way to create a bidirectional signal.

These can't be used internally in the FPGA and are only valid to be connected directly to a top-level `inout` (pin on the FPGA).

When an `inout` is written, the value will dictate if the pin's driver is enabled. If it is anything other than `bz` the driver will be enabled. A value of `bz` will disable the driver and leave the pin floating to be driven externally.

When an `inout` is read, the value at the actual pin is read. The value will never be `bz` .

A module with an `inout` can be instantiated inside another module as long as the `inout` is passed directly to an `inout` of the parent module. It can't be interacted with inside the FPGA.

Module Body

A module body consists of any number of the following statement types between `{` and `}` at the end of a module declaration.

Type Declarations

Inside the module body, local types can be defined.

These can be signals, DFFs, constants, or enums as defined in the types section.

Module Instances

A module can contain sub-modules. When you use a module, that is called *instantiation*.

A module instantiation takes the following form.

```
module_type module_instance_name optional_array_size ( port_and_param_[illegible]e
```

The `module_type` is the name of a previously defined module to be instantiated.

The `module_instance_name` is the name of this particular *instance*. It must start with a lowercase letter. It can then be followed by letters, numbers, and underscores. It is convention for it to be `snake_case` .

The `optional_array_size` follows the format defined in the sizing section for arrays. Structs are not supported for module instances.

If `optional_array_size` is omitted, a single instance of the module is created.

If `optional_array_size` is provided, an instance of the module will be created for every index in the specified size. Each port of every instance is concatenated into a single multidimensional port.

For example, if the module, `module_type` , had an output named `out` that was 1 bit wide and we instantiated 8 copies it with the following, then `module_instance_name.out` would be an 8-bit wide array with each index corresponding to each copy.

```
module_type module_instance_name[8]
```

If `out` was already an array then `module_instance_name.out` would be a 2-D array of size `[8][n]` where `n` is the size of a single `out` .

Finally, `port_and_param_connections` are a comma separated list of connections to ports and parameters.

Port connections take the form `.port_name(port_value)` where `port_name` is the name of the port and `port_value` is the value to connect to it. `port_value` can be an expression of matching width.

Parameter connections take the form `#PARAM_NAME(param_value)` where `PARAM_NAME` is the name of the parameter and `param_value` is the value to assign to it. `param_value` must be a constant expression that can be evaluated during synthesis.

Port and parameter connections can be presented in any order. Convention is to list all parameters first.

Connection Blocks

When declaring many DFFs or modules, you may find yourself specifying the same connection over and over. This is where connection blocks are helpful.

Connection blocks allow you to define a connection to a port or parameter for all instances inside of them. The most common use case for this is connecting the `clk` port of a `dff` to the `clk` signal in the module.

Connection blocks take the following format.

```
connection_list {
        declaration_or_connection_block
}
```

The `connection_list` is a comma separated list of port and parameter connections with the same format used during a typical module instantiation.

Inside the block, you can instantiate modules or DFFs. You can also nest other connection blocks.

A common use case is to have two nested connection blocks. The outer one for the `clk` port and the inner one for the `rst` port.

```
.clk(clk) {
    .rst(rst) {
        dff with_reset
    }
    dff without_reset
}
```

This allows you to easily create `dff` with and without resets.

If the `dff` or module instance is an array, the connections in the connection blocks are connected to each individual instance instead of to the concatenated value.

This can be helpful if you have an array modules and want to use the same parameter value for each one.

```
#PARAM_NAME(10) {
    module_type my_module[8]
}
```

In the above example, all eight instances of `module_type` will have their parameter, `PARAM_NAME` set to `10` .

If we wanted to assign different values to each one, they would need to be assigned inline as an array.

```
module_type my_module[8](#PARAM_NAME({8d0, 8d1, 8d2, 8d3, 8d4, 8d5, 8d6, 8d
```

This does not apply to the `dff` . The `clk` and `rst` inputs are always 1 bit and the `INIT` parameter always applies to the full `dff` .

Always Blocks

Always blocks provide a way to describe complex behavior in a way that resembles traditional programming.

An `always` block takes the following format.

```
always {
    always_statements
}
```

`always_statements` are defined in the block statements section.

The `always` block can contain any number of these separated by new lines or semicolons.

The statements are evaluated top-down. This means lower statements take precedent over higher ones.

For example, if you assigned the signal `led` the value of `0` then immediately assigned it `1` the first assignment would be ignored.

```
always {
    led = 0
    led = 1
}
```

This is *identical* to simply removing the first assignment.

An `always` block describes the desired behavior for part of your design. It is not actually *run* like typical code would be. The tools look at the block and figure out a way to make a circuit that would behave the same way. This means that there are some restrictions.

For example, repeat loops must have a fixed number of iterations. This is because there isn't actually a way to loop with hardware and the loop must be unrolled.

If a signal is written inside an `always` block, it must be written in all possible cases.

For example, this is not allowed.

```
always {
    if (button_pressed) {
        my_sig = 1
    }
}
```

In the case that `button_pressed` is false, `my_sig` won't have a value.

This could be remedied by adding an `else` clause or by assigning a value befor e `if` .

```
always {
    if (button_pressed) {
        my_sig = 1
    } else {
        my_sig = 0
    }
}
```

```
always {
    my_sig = 0 // default value

    if (button_pressed) {
        my_sig = 1
    }
}
```

These two blocks are functionally identical and the choice for each depends on your use case.

If it is a simple assignment like this, you may prefer the first.

If you have a reasonable default with complex logic for when it should deviate, use the second.

An exception to this rule is for DFFs. The `.d` input of a `dff` doesn't need to always be driven. This is because the `.d` input is implicitly assigned the value of `.q` at the start of the `always` block.

If the `.d` input isn't assigned, then the value of the `dff` won't change.

Here is an example where the `dff` will only increment when `button_pressed` is true.

```
always {
   if (button_pressed) {
       my_dff.d = my_dff.q + 1
   }
}
```

At the beginning of the `always` block `my_dff.d = my_dff.q` is implicitly added making this valid.

If an `always` block writes a signal, it is the driver for that signal meaning it can't be driven else where. In other words, a signal can be written in only one `always` block.

Globals

Globals allow you declare are group of constants, structs, and enums that are available anywhere in your project.

A global declaration takes the following form.

```
global GlobalName {
    declarations
}
```

`GlobalName` is the name of the `global` namespace. All the definitions in it are accessed by using `GlobalName.declaration_name`. It must start with a capital letter and contain at least one lowercase letter. Convention is to use `UpperCamelCase`.

`GlobalName` must be unique across the entire project.

The `declarations` are constants, structs, or enums.

Here's an example.

```
global MyGlobal {
    const CLK_FREQ = 100000000
    enum States { IDLE, RUN, STOP }
    struct color_struct { red[8], green[8], blue[8] }
}
```

These can be accessed later with `MyGlobal.CLK_FREQ` , `MyGlobal.States` , and `MyGlobal.color_struct` .

Testbenches

Testbenches look very similar to modules, but they serve as a way to run simulations.

The basic format is as follows.

```
testbench testbench_name {
    testbench_body
}
```

The `testbench_name` follows the same conventions as module names. It must start with a lower case letter and can be followed by letters, numbers, or underscores. It is `snake_case` by convention.

`testbench_body` is basically the same as the module body in the way can instantiate modules and DFFs or declare constants, enums, and structs.

However, instead of `always` blocks you have `test` blocks. You can also declare functions.

Test Blocks

The rules inside a `test` block are similar to those inside of an always block except that things are actually run sequentially like code.

The simulation will run each line in the `test` block line by line. Special test-only functions let you control the flow of the simulation.

A test block takes the following format.

```
test test_name {
    test_statements
}
```

The `test` keyword is followed by the name of the test, `test_name`, which must start with a lowercase letter and can be followed by letters, numbers, or underscores.

The `test_statements` can be any statements described in the statements section.

Comments

Comments can appear anywhere and are ignored by the Lucid parser. They share the same format as C/Java comments.

Single Line Comments

Single line comments take the following form.

```
// my comment
```

The `//` denotes the start of the comment and it continues to the end of the line.

Multi-line Comments

Multi-line comments take the following form.

```
/* inline comment */

/*
    multi
    line
    comment
*/
```

The comment is everything between the two markers, `/*` and `*/` .

They doesn't necessarily have to be on different lines and can be used to place a comment inline with code.

Signals

Basically any named value in Lucid can be thought of as a signal. This includes the `sig` type, ports on a module or `dff` , and even a `const` .

Sizing

When declaring a type, you can specify the width of that type with any number of array dimensions followed by an optional struct type with the following format.

```
[a]...[b]<struct_type>
```

After the array sizes, you can specify a struct type using the syntax `<struct_type>` where `struct_type` is some previously declared struct.

The only exception to this is module instances don't support structs.

All of the components of a signals size are optional and if omitted it defaults to 1 (either a bit or a single module instance).

If you declare something with both, you will have an array of that struct. Here is an example.

```
struct color { r[8], g[8], b[8] }
sig a_few_colors[16]<color>
```

A component of `a_few_colors` could be accessed like `a_few_colors[9].r` . To get the least significant bit of green for index 2 we could use `a_few_colors[2].g[0]`

Arrays

An array is a list consisting of elements of all the same size.

To make an array, you use the `[size]` syntax where `size` is some constant. These can be chained together to form multi-dimensional arrays.

A 1-D array is treated as a binary number. See the expressions section for some examples.

struct

Structs are a way to split a signal into arbitrary named sections.

Unlike an array that requires each element to be identical in size, in a struct, each element can be have any width.

The syntax of a `struct` declaration looks like the following.

```
struct struct_name {
    struct_elements
}
```

The `struct_name` is the name of the struct. It must start with a lowercase letter and can be followed by letters, numbers, or underscores. By convention, it is `snake_case` .

The `struct_elements` are a comma separated list of elements. Each element takes the following format.

```
signed element_name signal_width
```

The `signed` keyword is optional and will mark this element to be treated as a signed value. If the `signal_width` has a struct component to it, this does nothing.

`element_name` is the name of the element. It must start with a lowercase letter and can be followed by letters, numbers, or underscores. By convention, it is `snake_case`. It must be unique inside this struct.

`signal_width` is an optional width for the element as specified in the sizing section.

An example struct for holding a 24 bit color could look like this.

```
struct color { red[8], green[8], blue[8] }
```

Components of the struct are accessed via the `.element_name` syntax.

Signal Selection

When reading or writing a signal, if it isn't a single bit, you may need access only part of it. How you do this depends on the width of the signal.

Array Selection

If the signal is an array, you can use any of the array selectors to access part of it.

There are three main types of array selectors.

Index Selector

This selector is used to select a single index out of an array and takes the form `[idx]`.

The selector, `idx`, doesn't need to be constant.

This is the only selector that can have another array selector following it.

Constant Range Selector

You can select a range of indices using the `[max:min]` syntax.

The values of `max` and `min` must be constant values and `max` must be greater or equal to `min`.

This selector will select all the indices from `min` to `max` including both.

Fixed Width Selector

You will sometimes need to select multiple bits using a dynamic index. This is where the fixed width selector is helpful.

With the selector you specify a starting index then how many bits to include above or below including the start bit.

It takes the form `[start+:width]` to select a total of `width` bits starting at `start` and going up or `[start-:width]` to select `width` bits starting at `start` but going down.

For example, `[4+:3]` would select indices 4, 5, and 6. `[4-:3]` would select indices 4, 3, and 2.

The value used for `width` must be constant. However, the value used for `start` can be dynamic.

The reason for this is so that the resulting selection is always a fixed width.

Negative Indices

The indices for the selectors can also be negative. When an index is negative, it wraps around to the highest value.

For example, if you had an 8 bit wide signal then index `[-1]` would be the same as `[7]`. An index of `[-2]` would be the same as `[6]` and so on. An index of `[-8]` would be out of bounds and invalid.

The actual index used is essentially the negative value plus the width of the signal.

Many times a signal's width is defined by a constant or a parameter to allow it to be easily changed. Negative indices makes it easy to access the MSB of a signal regardless of its width.

Struct Selection

If a signal is a struct then to select an element from it you use the syntax `my_struct_signal.element_name` .

This assumes the signal `my_struct_signal` is of a `struct` type that has an element named `element_name` .

Types

sig

The `sig` type is short for *signal*. These are used as basic connections between parts of your design.

Each `sig` must have a single driver. Something that provides a value at all times as they have no state themselves.

Declaring a `sig` takes the following form.

```
signed sig sig_name sig_size = expression
```

Everything other than the `sig` keyword and `sig_name` are optional.

`signed` optionally marks the `sig` to be interpreted as signed.

`sig_name` is the name of the signal and it must start with a lowercase letter. It can then contain letters, numbers, and underscores. By convention, it is `snake_case` .

`sig_size` is the optional array/struct size of the signal. See sizing for details.

A signal can have an `expression` attached to it. This `expression` is considered to be the driver of the signal and it can't be written elsewhere if provided.

If the `= expression` portion is present, it behaves exactly the same as the following.

```
sig sig_name

always {
    sig_name = expression
}
```

A `sig` can be read and written inside an `always` block. The value that is read is always the last value written.

If a `sig` is read in the same `always` block that it is written, then it must be written before it is read.

Here's an example.

```
sig my_sig[8] // 8-bit wide signal

always {
    if (my_sig == 4) { // ERROR my_sig was read before being written
        my_sig = 2
    }
    my_sig = 3
}
```

A `sig` is often used explicitly internally in an `always` block as a temporary value.

```
sig result[9]

always {
    result = 8hff + 8h05
    if (result[8]) {
        // the addition overflowed!
    }
}
```

As mentioned before, inside an `always` block, the value of a `sig` is always the last value written to it.

```
sig my_sig[8]
always {
    my_sig = 4
    if (my_sig == 5) {
        // never reached
    }
    if (button_pressed) {
        my_sig = 5
    }
    if (my_sig == 5) {
        // only reached if button_pressed is true
    }
    my_sig = 1
}
```

Outside the `always` block that drives the `sig`, only the final value will ever be seen. In the previous example, the final line `my_sig = 1` means that anything reading `my_sig` outside of that `always` block will always see the value `1`.

dff

The `dff` is the building block of any sequential logic. It is the only type to have an internal state.

You can think of the `dff` as a single bit of memory.

The `dff` acts a lot like a module instance in that it has ports and parameters.

It has three inputs, one output, and one parameter.

The `.d` input is the data input. This is used to update the value of the `dff` .

The `.clk` input is the clock input. Whenever this transitions from 0 to 1, a rising edge, the `.d` input is saved.

> Generally, the `.clk` of every `dff` should all connect to the same system clock. You shouldn't drive this signal with other logic. FPGAs have special dedicated clock routing resources to efficiently distribute a clock signal to the entire (or large portions) of the FPGA. Messing with this can cause your design to simulate fine but fail in the real world.

The `.rst` input is the reset input. This is used to force the `dff` into a known state (0 or 1). If a reset isn't needed, this input can be left unconnected. You should only use this when a reset is actually needed as omitting it will reduce the routing complexity of your design.

A complementary `.arst` input can be used as an asynchronous reset. This input works the same as the `.rst` input, but it doesn't wait for a rising edge of the clock to reset the `.q` value. This is typically not desired and should be used with caution as it may lead to timing issues.

Only `.rst` *or* `.arst` can be used at once.

The `.q` output is the current value of the `dff` .

The `#INIT` parameter is used to specify the value that the `dff` will both initialize and reset to. It has a default value of `0` .

FPGAs are fully initialized when programmed regardless if the `dff` has a `rst` signal or not.

The format to declare a `dff` looks like the following.

```
signed dff dff_name dff_size (ports_and_params)
```

`signed` optionally marks the `dff` to be interpreted as signed.

The `dff_name` is the name of the dff and it must start with a lowercase letter. It can then contain letters, numbers, and underscores. By convention, it is `snake_case` .

`dff_size` is the optional array/struct size of the signal. See sizing for details.

The `ports_and_params` portion is a comma separated list of port and parameter connections. See module instances for details.

const

The `const` type provides a way to name constant values. This allows you to set the value in one place but use in in many places. That way if you need to change it later, it is easy.

The form for a `const` declaration looks the the following.

```
const CONST_NAME = const_expr
```

It starts with the `const` keyword followed by `CONST_NAME` , the name of your constant. The name must start with an uppercase letter and be followed by uppercase setters and underscores. By convention, it is `UPPER_SNAKE_CASE` .

The value of the constant is provided by `const_expr` . This can be any expression that evaluates to a constant value.

The width and sign of the `const` is inferred from the `const_expr` .

For example, if you need a constant of an 8-bit number you could use the following.

```
const MY_CONST = 8d120
```

The constant, `MY_CONST` , would be an array of 8 bits wide with the value `120` .

enum

An `enum` provides a way to group a list of constants whose value you don't explicitly care about.

Declaring one takes the following form.

```
enum EnumName { VALUE_1, VALUE_2, ... }
```

It starts with the `enum` keyword followed by its name, `EnumName`. The name must start with an uppercase letter and contain at lease one lowercase letter. It can otherwise contain letters, numbers, and underscores. By convention, it is `UpperCamelCase`.

Following the name is a list of comma separated values. The names of the values follow the same naming convention as constants.

To access the values of the `enum` you use the notation `EnumName.VALUE`.

The `enum` is often paired with a `dff` to store the state of a finite-state machine (FSM).

Here is a common example.

```
enum States { IDLE, RUN, STOP }
dff state[$width(States)](#INIT(States.RUN), .clk(clk))
```

The `$width` function can be used on the `enum` to get the minimum number of bits to store a value.

Expressions

Expressions appear all over Lucid.

An expression is something that can be evaluated to a single value.

The following sections appear in order of precedence. That means that higher up on the list, the earlier the Lucid parser will match that expression.

For example, the multiply and divide section comes before the add and subtract section. That means for something like `5 + 2 * 6` the parser will first evaluate the multiplication before the addition.

Value

The simplest expression is a value.

This can be a literal value, a signal, or a constant.

Signed

When a value is `signed`, it means that it will be interpreted as a 2's complement number and can represent negative and positive numbers instead of simply positive.

An unsigned value can represent values from 0 to 2^b -1 where b is the number of bits.

A signed value can represent values from -2^{b-1} to 2^{b-1} -1 where b is the number of bits.

The `signed` keyword can be applied as a prefix to most types to mark them as `signed`.

The functions `$signed()` and `$unsigned()` can be used to explicitly mark an expression as one or the other.

Expression that operate on numbers assume the values are unsigned binary numbers unless all of the arguments are marked as `signed`. In that case, they are all treated as 2's complement signed numbers.

It's common to see something like `$signed(a) * $signed(5)` to ensure that the multiplication is `signed` .

Group

The group expression takes the form `( expr )` where `expr` is an expression.

It is used to force the order expressions are evaluated. For example, `(5 + 2) * 6` will cause the `5 + 2` to be evaluated before the multiplication.

Concatenation

Concatenation provides a way to merge two or more arrays.

It takes the form `c{ expr1, expr2, ... }` where all `expr` are arrays or bits.

If the values passed into it are multi-dimensional arrays, all of their sub-dimensions must match. For example, an array of width `[2][8]` could be concatenated with an array of width `[3][8]` to form an array of width `[5][8]` .

The order of concatenation is such that the right most element occupies the least significant spot.

Here is an example.

```
c{4b1111, 4b0000} // result is 8b11110000
```

Duplication

Duplication provides a way to concatenate a single value many times with itself.

It takes the form `const_expr x{ expr }` where `const_expr` is a constant expression indicating how many times to duplicate `expr` .

The value `expr` must be an array (or bit).

Duplication works the same way as concatenation. For example, `3x{2b11}` is the same as `c{2b11, 2b11, 2b11}`. Both equal `6b111111`.

It is often used to get a value of all `1` with some given width.

For example, you could get the max value of a signal with `PARAM` bits using `PARAMx{b1}`.

Array Builder

The array builder provides a way to create an array from any number of identically sized expressions.

The syntax is `{ expr1, expr2, ... }` where all `expr` have the same width.

The order of the array is such that the right most `expr` is index 0.

Here is an example.

```
{2d2, 2d1, 2d0} // creates a [3][2] array where index [0] == 2d0
```

Invert

The invert operators allow you to perform a bitwise or logical invert.

Operator	Function
`~expr`	Bitwise invert
`!expr`	Logical invert

The result of a bitwise invert is the same width as `expr` with every bit invertec becomes `1` and `1` becomes `0`).

The result of a logical invert is a single bit. It is `1` if `expr` is equal to `0` and `0` otherwise.

Negate

The negate operator allows you to negate the 2's complement interpretation of an array.

It takes the form `-expr` where `expr` must be a 1-D array or bit. The result is the negative value of `expr`. The width of the result is always 1 bit wider than `expr` to accommodate overflow.

For example, `-4b0001` is equal to `5b11111`.

Using this operator does not mark the value to be considered signed.

Multiply and Divide

The multiply and divide operators do what you would expect, they multiply or divide two expressions.

Operator	Function
`expr * expr`	Multiplication
`expr / expr`	Division

The result from each operation is the minimum number of bits to represent the largest possible value.

The `expr` used in either must be 1-D arrays or bits.

For the computation to be signed, both `expr` must be signed. If either is unsigned, the computation will be unsigned.

FPGAs typically have some number of dedicated multipliers that make multiplication pretty fast.

Division can be very costly to perform. However, it is basically free if your denominator is a power of 2.

A simple trick you can often use is to multiply the numerator by something then divide by a power of 2 to approximate the division. For example dividing by 3 can be efficiently approximated with `n * 85 / 256` .

Addition and Subtract

The addition and subtraction operators allow you to add or subtract two expressions.

Operator	Function
`expr + expr`	Addition
`expr - expr`	Subtraction

The result from each operation is the width of the larger `expr` plus one bit to account for overflow.

The `expr` used in either must be 1-D arrays or bits.

For the computation to be signed, both `expr` must be signed. If either is unsigned, the computation will be unsigned.

Shifting

The shifting operators allow you to shift the bits in a 1-D array or bit left or right.

There are four versions of the shifting operator.

Operator	Function
`expr << amount`	Logical left shift
`expr >> amount`	Logical right shift
`expr <<< amount`	Arithmetic left shift
`expr >>> amount`	Arithmetic right shift

`amount` and `expr` must be 1-D arrays or bits.

Logical shifts always use `0` for the bits shifted in.

Arithmetic right shift will use the sign bit if `expr` is signed and `0` if it is not.

Arithmetic left always uses `0` and is functionally identical to logical left.

The result of right shifts have the width of `expr` .

The result of left shifts have the width of `expr` plus the value of `amount` .

Here are some examples.

```
4b0110 << 1 // 5b01100
4b0110 <<< 1 // 5b01100
4b1100 >> 1 // 4b0110
4b1100 >>> 1 // 4b0110
$signed(4b1100) >> 1 // 4b1110
```

Bitwise

Bitwise operators allow you to perform the boolean operations *and*, *or*, and *xor* on a bit-by-bit basis of two expressions with matching widths.

Operator	Function
`expr & expr`	AND

Operator	Function
`expr | expr`	OR
`expr ^ expr`	XOR

The widths of both `expr` must match exactly. Otherwise, they can be anything.

The result has the same width as `expr` .

Reduction

Reduction operators allow you to perform the boolean operations *and*, *or*, and *xor* on all the bits in an expression with all the other bits.

Operator	Function
`& expr`	AND
`| expr`	OR
`^ expr`	XOR

The result of any reduction operator is a single bit.

The `&` operator is `1` if every bit in `expr` is `1` and `0` otherwise.

The `|` operator is `1` if any bit in `expr` is `1` and `0` otherwise.

The `^` operator is `1` if there are an odd number of `1` bits in `expr` and `0` otherwise.

Comparison

The comparison operators allow you to compare the values of two 1-D arrays or

Operator	Function
`expr < expr`	Less than
`expr > expr`	Greater than
`expr == expr`	Equality
`expr != expr`	Not equal
`expr <= expr`	Less than or equal
`expr >= expr`	Greater than or equal

The result of any comparison is a single bit. The value `1` means the comparison was true and `0` means false.

For a comparison to be signed, both `expr` must be signed. If either is unsigned, the comparison will be unsigned.

Logical

Logical operators allow you to perform the boolean operations *and* and *or* on two logical values.

Operator	Function
`expr && expr`	AND
`expr || expr`	OR

The result from either operator is a single bit.

The `expr` is considered to be *true* if it isn't `0` and *false* only when it equals `0` .

The `&&` operator will produce `1` if both `expr` are *true* and `0` otherwise.

The `||` operator will produce `1` is either `expr` is *true* and `0` otherwise.

Ternary

The ternary operator allows you select between two identically sized expressions using the value of a third expression.

It takes the following form.

```
selector ? true_expr : false_expr
```

The `selector` is considered to be *true* if it isn't `0` and *false* only when it equals `0`.

The result has the same width of `true_expr` and `false_expr`, which must match.

When `selector` is *true* the result is `true_expr` otherwise it is `false_expr`.

Always/Test Block Statements

This section contains the statements that can appear inside always and test blocks.

Assignments

The most common statement is the assignment statement. These allow you to write a value to a signal.

They take the following form.

```
signal = expression
```

Where `signal` is the writable signal to have the value of `expression` written to it.

The `signal` doesn't necessarily have to be the entire width of the signal and signal selectors can be used to assign a value to only part of the signal.

Here is an example.

```
sig my_sig[16]

always {
    my_sig[7:0] = 8haa
    my_sig[15:8] = 8hbb
}
```

If any portion of a signal is driven inside an `always` block then all of the signal must be driven somewhere in the block. In the above example, removing either assignment would result in an error since that `always` block would have a value for only half of the signal.

if

An `if` statement allows you to conditionally consider a block of statements.

They take the following form.

```
if (condition) {
    statements
} else {
    statements
}
```

If `condition` is *true*, meaning non-zero, then the statements in the first `statements` section are considered. If it is *false*, meaning zero, then the statements in the second `statements` section are considered.

The `else` portion is optional.

The brackets can be omitted if there is only one statement in `statements` .

case

The `case` statement provides a way to cleanly write a group of conditional statements that depend on the value of a single expression.

It takes the following form.

```
case (condition) {
    VALUE_1:
        statements
    VALUE_2:
        statements
    ...
    default:
        statements
}
```

The `condition` expression is evaluated and compared against the provided values (`VALUE_1` , `VALUE_2` , etc) inside the `case` block.

If one of the values match `condition` , then its corresponding group of `statements` are considered. If none of the values match, an optional `default` branch's `statements` are considered.

Unlike some programming languages, there is no `break` keyword and the branches don't fall through to the next condition without one.

You often see these used in conjunction with enums where the condition is a dff that holds the current state and each branch is one of the `enum` 's values.

repeat

The `repeat` statement allows for a block of statements to be considered mult
times.

They take the following form.

```
repeat(i, count, start = 0, step = 1) {
    statements
}
```

Here `count` is a constant expression indicating how many times the block of `statements` should be considered.

The arguments `start` and `step` are optional and have default values of `0` and `1` respectively.

The `i` argument is an optional variable name that can be used inside the `repeat` block. It will start with the value `start` and increment by `step` each iteration of the loop to a final value of `start + step * (count - 1)`. It must start with a lowercase letter and can contain letters, numbers, and underscores. By convention, it is `snake_case`.

If `i` is omitted, then only `count` should be provided making it take the form `repeat(count)`.

The `count` value can be an expression depending on outer loops `var` in the case of nested `repeat` blocks.

Here is an example.

```
repeat(i, 3) {
    repeat(j, i+1) {
        $print("(i, j) = (%d, %d)", i, j)
    }
}
```

This would print the following (in a simulation).

```
(i, j) = (0, 0)
(i, j) = (1, 0)
(i, j) = (1, 1)
(i, j) = (2, 0)
(i, j) = (2, 1)
(i, j) = (2, 2)
```

> While it may seem like `repeat` works like `for` loops in many programming languages, it is important to remember that `always` blocks are only a convenient abstraction for describing a circuit's behavior. When your design is synthesized it must be converted to hardware.
>
> This means that all loops must be *unrolled*. A `repeat` block is identical to simply copy-pasting the contents over and over and replacing the loop variable with a different value for each one.

Function call

In test blocks, you can call simulation functions as a statement.

These are used to control the simulator or produce output like `$print()` used in the repeat example.

Literals

Numbers

Numbers can be represented in decimal, binary, or hexadecimal. In each case you can choose to explicitly specify the number of bits.

Format	Radix	Width
`0123456789`	10 (decimal)	Minimum bits required
`d0123456789`	10 (decimal)	Minimum bits required
`b01xz`	2 (binary)	Number of digits
`h0123456789ABCDEFxz`	16 (hex)	Number of digits * 4

Decimal numbers can be written as a stand-alone number. For example, `12` will have the expected value of `12`. They can also be prefixed with a `d` to specify that it is a decimal number like `d12`. In both of these cases, the value will have the minimum number of bits required, 4.

Binary numbers are written with the `b` prefix. After that, they can have the digits `0`, `1`, `x`, and `z`.

Hex numbers are written with the `h` prefix. After that, they can have the digits `0` through `9`, `A` through `F`, `x`, and `z`.

The value of `x` means either *don't care* or *unknown* depending on the context.

It is impossible for hardware to realize the value `x` so outside of simulations these are actually `0` or `1`. You can assign something `x` if you don't care what the value is and this will give the tools the freedom to choose whatever value is most efficient.

The value of `z` means *high-impedance* and is covered in the ports section.

FPGAs don't have the hardware internally to realize `z` values so they can only be used on top-level outputs and inouts.

To explicitly specify the width of a number, you add the number of bits before the radix prefix.

For example, `8d10` will have the value `10` and be 8 bits wide.

If the explicit width is wider than the minimum required width, the value is padded with `0` unless the most-significant (left-most) digit is either `x` or `z`. In that case, it will be padded with `x` or `z` respectively.

For example, `12hx0` has the value `12bxxxxxxxx0000` .

If the explicit width is less than the minimum required width, the value will be truncated and a warning will be shown.

Underscores can be added anywhere in the value portion of a number to help with readability. For example, `100_000_000` or `8b1010_1100` .

Arrays

Literal arrays can be constructed using other literals and the array builder expression.

Here is an example of an array of 4, 8 bit values.

```
{8d4, 8d3, 8d2, 8d1}
```

Strings

Strings are just an easy way of creating an array of 8 bit values that correspond to text.

They take the form of text enclosed by quotation marks like `"this example"` .

The right-most character is index 0 in the resulting array.

For example, `"Hi"` is equal to `{8h48, 8h69}` (`h48` is the code for *H* and `h69` is *i*).

You will often see strings used in conjunction with `$reverse()` to make index 0 be the left-most letter.

Structs

Struct types are covered in the struct section.

To create a literal of a `struct`, you use the following syntax.

```
<struct_type>(.element_name(const_value), ...)
```

Here `struct_type` is a previously defined `struct` type.

The following comma separated list must contain every element in `struct_type`. The `element_name` is the name of element in the `struct_type` and `const_value` is the constant value to assign to the element.

Here is an example.

```
struct color { red[8], green[8], blue[8] }
const ALCHITRY_GOLD = <color>(.red(250), .green(172), .blue(31))
```

Functions

Built-in

In the table below, the argument type of *Value* means a 1-D array or bit. In other words, something that can represent a number.

Function	Argument Type	Purpose
`$width(expr, dim)`	`expr` is an array and `dim` is a constant value	Provides the width of `expr` on dimension `dim`. If `expr` is a 1-D array or bit, `dim` is optional and assumed to be `0`. If `expr` is a multi-dimensional array, then `dim` is required. For example, `$width({4b0,4b0}, 0)` is equal to `2`, `$width({4b0,4b0}, 1)` is equal

Function	Argument Type	Purpose
		to `4` , and `$width(8b0)` is equal to `8` .
`$signed(expr)`	Value	Marks the value to be interpreted as signed without changing the underlying bits.
`$unsigned(expr)`	Value	Marks the value to be interpreted as unsigned without changing the underlying bits.
`$clog2(expr)`	Constant value	Calculates ceiling log base 2 of `expr` .
`$cdiv(numer, denom)`	Constant value	Calculates the ceiling of `numer` / `denom`
`$pow(expr, expo)`	Constant values	Calculates `expr` to the power of `expo` .
`$reverse(expr)`	Constant array	Reverses the indices of the outer most dimension of `expr` .
`$flatten(expr)`	Anything	Returns a 1-D array of all the bits in `expr` . Arrays are concatenated in order and structs are in the order their elements were declared.
`$build(expr, dims...)`	`expr` is a value and `dims` are constant values	Converts a 1-D array into a multi-dimensional array based on the `dims` passed in. Each `dim` corresponds to how many times it should be split. For example, `$build(b111000, 2)` will split it into 2 becoming `{b111, b000}` . More than one `dim` can be supplied to build more dimensions. For example, `$build(b11001001, 2, 2)` becomes `{{b11, b00}, {b10, b01`

Function	Argument Type	Purpose
`$resize(expr, size)`	`expr` is a value and `size` is a constant values	Resizes a value either smaller or wider. If `expr` is signed, it will be sign extended.
`$fixed_point(real, width, fractional)`	`real` is a real number, `width` and `fractional` are constant values	Calculates the nearest fixed-point representation of `real` using a total width of `width` and `fractional` fractional bits. For example, `$fixed_point(3.14, 8, 4)` produces `8d50` .
`$c_fixed_point(real, width, fractional)`	`real` is a real number, `width` and `fractional` are constant values	Calculates the smallest fixed-point representation of `real` that is still larger than it using a total width of `width` and `fractional` fractional bits. For example, `$c_fixed_point(3.14, 8, 4)` produces `8d51` .
`$f_fixed_point(real, width, fractional)`	`real` is a real number, `width` and `fractional` are constant values	Calculates the largest fixed-point representation of `real` that is still smaller than it using a total width of `width` and `fractional` fractional bits. For example, `$f_fixed_point(3.14, 8, 4)` produces `8d50` .
`$is_sim()`	None	Evaluates to `1b1` during interactive simulations and `1b0` otherwise.

Simulation Only

These functions are only available during simulations. In other words, inside tes blocks or test functions.

Function	Argument Type	Purpose
`$tick()`	None	Propagates all signal changes and captures the state.
`$silent_tick()`	None	Propagates all signal changes.
`$assert(expr)`	Any expression, typically a comparison.	Checks that `expr` is non-zero (true). If it is zero the simulation is halted and an error is printed indicating the failed assert.
`$print(expr)`	Any expression	Prints the value of `expr` . If `expr` is a string literal, it prints the string. Otherwise, it prints `expr = value` where `expr` is the text and `value` is the actual value.
`$print(format, exprs...)`	`format` is a string literal and `exprs` is a variable number of expressions depending on the `format`	Prints the string `format` with the values of the provided `exprs` replaced where applicable. Valid format flags are `%d` for decimal, `%h` for hex, `%b` for binary, `%nf` for fractional where `n` is the number of fractional bits.

User Created

Inside testbenches, you can create your own functions using the following syntax.

```
fun function_name(argument_list) {
    function_body
}
```

`function_name` is the name of the function. It must start with a lowercase lett and be followed by letters, numbers, and underscores. By convention, it is `snake_case` .

The `argument_list` is an optional list of arguments. They act the same as read-only signals and have a width of 1 bit if a width isn't provided. They may also be marked as signed using the `signed` keyword.

To call a function, you use the syntax `$function_name(arg, ...)` .

Here is an example function with an argument.

```
fun tick_clock(times[32]) {
    repeat(times) {
        clk = 1
        $tick()
        clk = 0
        $tick()
    }
}
```

This could be called using something like `$tick_clock(20)` .

NEXT »
Alchitry Constraints Reference