CHAPTER 13
Lucid Reference

This chapter contains information on Lucid and can be used as a reference. Although a lot of Lucid's syntax and features are covered throughout the book, this chapter explains them in a more comprehensive way. This chapter is roughly organized from an outside-in approach, meaning we start from a full module declaration and work our way down to individual expressions and constants.

The templates are written using syntax similar to ANTLR grammar. Things in single quotes are string literals, meaning they must appear as is. A question mark after something means that it is optional. An asterisk following something means zero or more of that thing may be present. A plus following something means one or more of that thing may be present. Parentheses are used to group multiple tokens.

The following page is intended as a quick reference guide. It covers the most commonly used features of Lucid but is not an exhaustive reference. Refer to the rest of the chapter for more details on any specific feature. You may find it helpful to print the reference guide.

Lucid Quick Reference

Modules

```
module name #(
    PARAM = DEFAULT_VALUE : CONSTRAINT
  )(
    port_type name
  ) {
  SIGNAL_AND_MODULE_INSTANCES
  always {
    ALWAYS_STATEMENTS
  }
}
```

Assignment Blocks

```
.port_name(EXPR), #PARAM_NAME(CONST_EXPR) {
  SIGNAL_AND_MODULE_INSTANCES
}
```

Types

`input`	Port input, read-only
`output`	Port output, write-only
`inout`	Bidirectional port with subsignals: *read*, *write*, *enable*
`sig`	Acts as a way to connect expressions
`var`	Used in loops
`dff`	Stores data, subsignals *clk*, *rst*, *d*, *q*
`fsm`	Similar to a *dff* but with built-in constants
`const`	Constant value

Instantiation

DFF: `dff name (.clk(clk_sig));`

FSM: `fsm name = { STATE, STATE, ... };`

Module: `module name (#PARAM(CONST_EXPR), .port(EXPR));`

Always Statements

Assignment: `signal = EXPR;`

If: `if (EXPR) { ... } else { ... }`

Case:
```
case (EXPR) {
  CONST: ...
  default: ...
}
```

For: `for (INIT; CONDITION; STEP) { ... }`

Bit Selectors

Single index: `[EXPR]`

Constant range: `[MAX_CONST:MIN_CONST]`

Fixed-width increment: `[START+:WIDTH_CONST]`

Fixed-width decrement: `[START-:WIDTH_CONST]`

Functions

`$clog2(a)`	Performs ceiling of $\log_2$(a)
`$pow(a,b)`	Raises *a* to the *b* power
`$reverse(a)`	Reverses the indices of the outermost dimension of *a*
`$flatten(a)`	Collapses *a* into a 1D array
`$signed(a)`	Forces *a* to be interpreted as signed
`$unsigned(a)`	Forces *a* to be interpreted as unsigned

Numbers

Integers: `0-9`

Decimal: `Wd0-9`

Hexadecimal: `Wh0-9 A-F x z`

Binary: `Wb0 1 x z`

Strings: `"TEXT"`

Arrays

Array builder: `{ EXPR, EXPR, ... }`

Concatenation: `c{ EXPR, EXPR, ... }`

Duplication: `NUM x{ EXPR }`

Bitwise Operators

NOT: `~EXPR`

AND: `EXPR & EXPR`

OR: `EXPR | EXPR`

XOR: `EXPR ^ EXPR`

XNOR: `EXPR ~^ EXPR`

Logical Operators

NOT: `!EXPR`

AND: `EXPR && EXPR`

OR: `EXPR || EXPR`

Ternary: `EXPR ? EXPR : EXPR`

Comparison Operators

Less than: `EXPR < EXPR`

Greater than: `EXPR > EXPR`

Less than or equal: `EXPR <= EXPR`

Greater than or equal: `EXPR >= EXPR`

Equal: `EXPR == EXPR`

Not equal: `EXPR != EXPR`

Reduction Operators

OR: `|EXPR`

NOR: `~|EXPR`

AND: `&EXPR`

NAND: `~&EXPR`

XOR: `^EXPR`

XNOR: `~^EXPR`

Math Operators

Negate: `-EXPR`

Add: `EXPR + EXPR`

Subtract: `EXPR - EXPR`

Multiply: `EXPR * EXPR`

Divide: `EXPR / EXPR`

Shift right: `EXPR >> NUM_BITS`

Signed shift right: `EXPR >>> NUM_BITS`

Shift left: `EXPR << NUM_BITS`

Signed shift left: `EXPR <<< NUM_BITS`

Modules

Module declarations take place at the root of a file. Each file can have at most one module declaration, and it is convention for the module's name to match the file's name.

A module declaration takes the following form:

```
'module' name param_list? port_list module_body
```

The declaration starts with the `module` keyword followed by the name of this module. The `param_list` is optional and is followed by the `port_list` and the `module_body`.

A module's name must start with a lowercase letter. The rest of the name can contain lower- or uppercase letters, numbers, and underscores.

Here is an example of a bare-bones, but complete, module:

```
module my_module (
    input clk,  // clock
    input rst,  // reset
    output out
  ) {

  always {
    out = 0;
  }
}
```

Parameter Lists

The purpose of a *parameter list* is to specify parameters that are used to make your module more flexible. The values of the parameters can be changed for each instance of the module, but they are always constant values.

A `param_list` takes the following form:

```
'#(' param_dec (',' param_dec)* ')'
```

That is, a series of one or more `param_dec` separated by commas and enclosed in parentheses prefixed by a hash symbol.

Each `param_dec` takes the following form:

```
name ('=' expr)? (':' param_constraint)?
```

Here is an example of a `param_dec`:

```
SIZE = 8 : SIZE > 0
```

It starts with the parameter's name, which must start with a capital letter followed by only capital letters, numbers, and underscores. This is followed by an optional default value, which is any constant expression, and an optional `param_constraint` that is prefixed with a colon.

The `param_constraint` is a constant expression that can use the parameter being declared and any parameters declared before this one. This expression is evaluated with the parameter values set when the module is instantiated. If they fail (have a value of 0), an error will be thrown.

Because the default value and constraint are optional, the most minimal version is simply a name:

```
SIZE
```

Here is an example of a full `param_list` from the Counter component:

```
#(
  SIZE = 8 : SIZE > 0, // Width of the output
  DIV = 0  : DIV >= 0, // number of bits to use as divisor
  TOP = 0  : TOP >= 0, // max value, 0 = none

  // direction to count, use 1 for up and 0 for down
  UP = 1 : UP == 1 || UP == 0
)
```

Port Lists

Every module has a *port list*. This is where you list the inputs, outputs, and inouts of the module. It defines all the external connections.

The `port_list` takes the following form:

```
'(' port_dec (',' port_dec)* ')'
```

It is a list of `port_dec` separated by commas and enclosed in parentheses.

The `port_dec` takes the following form:

```
'signed'? ('input' | 'output' | 'inout') struct_type? name array_size?
```

Here is an example of a few `port_dec` statements:

```
output<Memory.master> memOut
inout sda
signed output value [8]
```

The `port_dec` starts with an optional signed type modifier. This will make the value be interpreted as two's complement. This is followed by the type: `input`, `output`, or `inout`. Each port can then be a struct specified by an optional `struct_type` and an array of potentially multiple dimensions specified by the optional array size. The

name of a port is the port's name and must start with a lowercase letter followed by upper- or lowercase letters, numbers, or underscores.

Each `port_dec` can be for an `input`, `output`, or `inout`. Inputs and outputs are fairly straightforward. Inputs can be read and get their value only from an external connection. Outputs can only be written. The `inout` type is for bidirectional signals and can be used only by the top-level module of your design or must be routed directly to an `inout` in the top-level module. The `inout` can be read and written to.

Here is an example of a full `port_list` from the Memory Arbiter component:

```
(
  input clk,                                    // clock
  input rst,                                    // reset
  input<Memory.slave> memIn,                    // memory inputs
  output<Memory.master> memOut,                 // memory outputs
  input<Memory.master> devIn [DEVICES],         // devices inputs
  output<Memory.slave> devOut [DEVICES]         // devices outputs
)
```

Note that you can use a module's parameters in the array sizes of the ports. In the preceding example, the parameter `DEVICES` is used to size `devIn` and `devOut`.

Module Body

A *module's body* consists of a list of statements enclosed in curly braces:

```
'{' statement* '}'
```

A *statement* can be any one of the following.

Constant declaration

Constant declarations are used to define a constant that can be used later. Wherever the constant is used, it will be replaced by the value it represents. The declaration takes the following form:

```
'const' name '=' expr ';'
```

Here's an example:

```
const MY_CONST = 12;
```

The *`name`* of a constant must start with a capital letter and can be followed by capital letters, numbers, and underscores. It can't contain lowercase letters.

Here, *`expr`* is an expression. The expression must have a constant value—that is, one that can be determined at synthesis time.

Variable declaration

Variable declarations are used to define one or more variables that can be used later. Variables are used to store temporary integer values that you won't see in your actual design. The most common use for them is as the index in a `for` statement.

The declaration takes the following form:

```
'var' name array_size? (',' name array_size?)* ';'
```

Here's an example:

```
var i, j, k;
```

Each variable declaration can define one or more variables separated by commas. Each variable requires a *name* that starts with a lowercase letter followed by lower- or uppercase letters, numbers, and underscores. The name is then followed by an optional array size.

Unlike other signals, a `var` can hold integers without being an array. An array of `vars` would act like an array of `ints` in a programming language.

Signal declaration

Signal declarations are used to define one or more signals. Signals should be thought of as wires in your design; they are used to carry a value from one expression to another. Inside an `always` block, they can be read and written. Their value will be whatever was last written in the `always` block, and they must be written before being read.

The declaration takes the following form:

```
'signed'? 'sig' struct_type? name array_size? (',' name array_size?)*
```

Here's an example:

```
sig mySig [16];
```

The type `sig` behaves similarly to other types. Without the optional `struct_type` or `array_size`, it will be a single bit wide and can hold a value of 0 or 1. If `struct_type` is present, it is used for all `sigs` declared in this line.

DFF declaration

DFF declarations are used to define one or more DFFs. A DFF acts as a basic unit of memory and is commonly used to split up combinational logic.

The declaration takes the following form:

```
'signed'? 'dff' struct_type? name array_size? inst_cons?
(',' name array_size? inst_cons?)*
```

Here are some examples:

```
dff ctr [32] (#INIT(32d15), .clk(clk));
signed dff value [8] (.clk(clk), .rst(rst));
dff<color> rgbValue;
```

Like `input`, `output`, and `inout` types, the `dff` type has an optional `struct_type` and `array_size` that will specify the size of the `dff`. However, unlike other simpler types, the `dff` acts more like a module instance. The `dff` has four ports: `clk`, `rst`, `d`, and `q`. The `clk` port is used as the clock and must be connected when the `dff` is declared. The `rst` port can optionally be connected. If it isn't connected, the DFF won't have a reset signal. Typically, if you don't need a reset, don't use it. The ports `d` and `q` are used to input a new value and read the current value, respectively. These are typically used inside an `always` block.

The `dff` type also has a few optional parameters. The `INIT` parameter can be used to set the initial value of the DFF when the FPGA starts up or the reset signal is asserted. The `IOB` parameter accepts a value of 0 or 1, and when 1, the DFF will be marked to be packed into an I/O buffer in the FPGA. This is useful when the DFF output goes directly out of the FPGA and you want it have minimal delay. If the FPGA being targeted doesn't support this, it will be ignored.

The optional `inst_cons` takes the following form:

```
'(' (param_con | port_con) (',' (param_con | port_con))* ')'
```

Here's an example:

```
(#INIT(32d15), .clk(clk), .rst(rst))
```

This is a list of parameter and port connections enclosed by parentheses and separated by commas. The details of the connections are in the following section.

Instance connections. The connections made to the ports of a DFF declaration, FSM declaration, or module instance are made using the following form:

```
'.' name '(' expr ')'
```

Here are some examples:

```
.dataIn(myData)
.value(8d12)
```

Here *`name`* is the name of the port to be connected to *`expr`*, and *`expr`* is an expression. The name must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Parameter assignments follow a similar form but replace the period with a hash:

```
'#' name '(' expr ')'
```

Here's an example:

```
#WIDTH(16)
```

Another difference is that *expr* must be known at synthesis time. That means it can contain only constants or parameters.

FSM declaration

FSM declarations are used to define a single FSM. The `fsm` type is similar to the `dff` type. The only real difference is that you specify a list of different constant states that can be assigned to it. The size of the `fsm` will be automatically set to be able to contain any of the states you declared. An array of `fsms` is able to store multiple states.

The declaration takes the following form:

```
'fsm' name array_size? inst_cons? '=' '{' name (',' name)* '}'
```

Here's an example:

```
fsm state (.clk(clk), .rst(rst)) =
  {IDLE, START, WAIT_CMD, READ, WRITE, STOP, WAIT} ;
```

The first *name* is the name of the `fsm`. This must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

The list of *names* is the list of states. These must start with an uppercase letter and can be followed by uppercase letters, numbers, and underscores.

Like DFF declarations, `fsms` have the `INIT` parameter. However, if present, it can be assigned to only one of the states. If it isn't present, the first state in the list is used by default as the initial state.

Again like `dffs`, there are four ports: `clk`, `rst`, `d`, and `q`, which behave exactly the same way.

To access the state constants of an `fsm`, you prefix the name with the name of the `fsm` followed by a period. For example, if we had an `fsm` called `current_state` and it had a state named `RUNNING`, we could access this constant with `current_state.RUNNING`. This allows you to use the same state names for different `fsms` without collision.

Struct declaration

Struct declarations allow you to define a new `struct`. The `struct` can then be used to size other signals later in your design. A `struct` allows you to bundle a collection of names into a single type. It is similar to an array, but the elements are accessed with names instead of numbers and each element can have a unique width.

The declaration takes the following form:

```
'struct' name
  '{' name struct_type? array_size? (',' name struct_type? array_size?)* '}'
```

Here's an example from the Memory Arbiter component:

```
// simple structure to hold pending commands
struct command {
  valid,                          // valid flag (1 = valid)
  write,                          // write/read flag (1 = write)
  addr[23],                       // address to read/write
  data[32]                        // data for writes
}
```

The *name* of the `struct` must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Following the *name* is a list of the `struct's` members. Each member can simply be a name, which follows the same naming convention as the `struct`. However, the members can also be arrays and/or `structs` allowing you to nest `structs`.

Here is an example of using nested `structs`:

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

struct video_data {
  valid,
  pixel<color>
}

sig mySig<video_data>;

always {
  mySig.pixel.red = 129;
}
```

If two data types are declared with the same `struct` type, they can be directly assigned to each other. This is equivalent to assigning each member individually.

Module instance

Module instantiation is similar to using DFF declaration, except a module in your project serves as the *type*. An instantiation takes the following form:

```
name name array_size? inst_cons?
```

Here are some examples:

```
reset_conditioner reset_cond;
pipeline pipe [8] (#DEPTH(16));
```

The first *name* is the name of the module you want to instantiate, and the second *name* is the name of this instance. This must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Similar to using DFF declarations, you can connect inputs and parameters to the module with the optional instance connections.

Inputs that weren't assigned at instantiation and outputs can be accessed using the name of the instance followed by a period and the name of the port—for example, `counter.value`.

You can also specify an array size for the instance. If you do this, that module will be duplicated in your design for each element in the array. Any connections or parameters specified at instantiation will be connected to each instance separately. For example, if your module has a single-bit input `clk` and you connect a signal to it, that single-bit signal will be duplicated and connected to each module. However, ports not connected at instantiation are packed into arrays. Single-bit signals become arrays the same size as your module array, and arrays become multidimensional arrays. The module indices become the first dimensions in the array. In other words, you select the module first and then the bits in the port.

Assignment block

An *assignment block* is a way to connect a signal or value to a port or parameters of multiple modules. It is most commonly used to connect the clock and reset signals as they exist on most modules.

The assignment block takes the following form:

```
con_list '{' ((dff_dec | fsm_dec | module_inst) ';' | assign_block)* '}'
```

Here `con_list` is a comma-separated list of instance connections. These are the parameter and port connections that will be applied to all the DFF declarations, FSM declarations, and module instances. All instances in the block must have ports and parameters with the same names as the ones in the list.

Assignment blocks can be nested.

Here is an example of a common use case:

```
.clk(clk) {
  dff noResetDff;
  dff anotherDff;

  .rst(rst) {
```

```
    dff dffWithReset;

    myModule instOfMyModule;
  }
}
```

always block

`always` blocks are where all a module's logic resides. This is where you can assign values to signals, perform calculations, and evaluate expressions.

An `always` block takes the following form:

```
'always' block
```

It simply is the `always` keyword followed by a *`block`*. A *`block`* is a single `always` statement, or multiple `always` statements enclosed in curly brackets.

Statements are evaluated from top to bottom, with lower statements having precedence over previous statements. In this regard, it can feel a bit like programming, where statements are evaluated sequentially. However, they aren't really evaluated sequentially, but simply interpreted sequentially. You should think of everything in an `always` block as being evaluated continuously.

The statements in an `always` block can be any of the following.

Assignment. Assignments are the most basic `always` statement and are perhaps the most useful. They take the following form:

```
signal '=' expr ';'
```

Here are some examples:

```
data_out = data.q;
scl_reg.d = 0;
scl_reg.d = c{2b11, (scl_reg.WIDTH-2)x{1b0}} + 1;
state.d = state.WRITE;
```

Here *`signal`* is anything that can be assigned a value, and *`expr`* is an expression.

As you might expect, this will assign the value of *`expr`* to *`signal`*.

If statement. `If` statements provide a simple method for making decisions. They allow you to make a set of `always` statements to be conditionally evaluated.

An `if` statement takes the following form:

```
'if' '(' expr ')' block ('else' block)?
```

Here's an example:

```
if (!&ctr.q)         // if counter isn't full
  ctr.d = ctr.q + 1; // increment it
```

```
else
  ctr.d = newValue;
```

If the value of *expr*, which is an expression, is nonzero, then the first *block* is evaluated. If the value of *expr* is zero and the optional `else` clause is present, then the second *block* is evaluated.

A *block* is a single `always` statement, or multiple `always` statements enclosed in curly brackets.

Case statement. Case statements provide a compact way to select a set of `always` statements based on the value of an expression.

They take the following form:

```
'case' '(' expr ')' '{' case_elem+ '}'
```

Here's an example:

```
case (ctr.q) {
  0: out = 15;
  1: out = 6;
  2: out = 24;
  default: out = 0;
}
```

Here *expr* is the expression to evaluate and is an expression.

Each `case_elem` takes the following form:

```
(expr | 'default') ':' always_statements+
```

Here's an example:

```
state.INIT:
  value = 1;
  ctr.d = 0;
  state.d = state.NEXT;
```

This *expr* is also an expression, but it must have a constant value known at synthesis time. The `default` keyword is used to catch all values not otherwise stated.

If the *expr* being evaluated by the `case` statement matches the `case_elem's` *expr* value, then the corresponding set of `always` statements are evaluated.

The same effect could be accomplished using a set of `if-else` statements, but `case` statements are more concise. Unlike programming, there is no performance improvement generally associated with using a case statement over a series of `if-else` statements.

for statement. `for` statements provide a compact way to write something that is otherwise repetitive. It takes the following form:

```
'for' '(' assign_stat ';' expr ';' var_assign ')' block
```

Here's an example from the Encoder component:

```
for (i = 0; i < WIDTH; i++) {
  if (in[i]) // if the bit is set
    out = i; // the output is the index of that bit
}
```

A `for` loop has four main components. The *`assign_stat`* is an assignment that is evaluated once before everything else. The *`expr`*, an expression, is evaluated before each loop iteration. If it is nonzero, the loop continues. Otherwise, the loop exits. The *`var_assign`* is evaluated at the end of each iteration. The *`var_assign`* is identical to an assignment, except that it also supports the *increment* and *decrement* shorthand for variables.

The form for variable increment and decrement is as follows:

```
signal ('++'|'--')
```

Here's an example:

```
i++
```

Here, *`signal`* is a variable. If it is followed by `++`, the value of the variable is incremented by 1. If it is followed by `--`, it is decremented by 1.

Each iteration of the loop, the statements in *`block`* are evaluated with a different value of the loop variable. The loop will behave exactly the same as if you manually copied and pasted the block and replaced any instances of the loop variable with each loop's value. `For` loops must have a constant number of iterations, because if the loop can't be unrolled, it can't be realized in hardware.

Expressions

Expressions are common throughout Lucid. Although there are many operators, they all boil down to a single value. The following is a list of the possible forms for an expression.

Signals and Constants

The most basic form of an expression is a *signal* or *constant*. This can be the name of any readable signal such as inputs, module instance outputs, sigs, vars, parameters, or constants. It can also be a literal value.

Functions

Functions perform an operation on a set of arguments. Many of them can be used only with constant values and are evaluated at synthesis. Others simply change how values are interpreted.

$clog2(arg)

This function performs the operation *ceiling(log_2(arg))*—that is, the ceiling of the log base 2 of its argument. This is useful when needing to determine the number of bits that are needed to store a certain number, as *`$clog2(num + 1)`* will return the minimum numbers of bits to store the value *`num`* with the exception of *`num`* being 0. This function can be used only with constant values and is evaluated at synthesis time.

$pow(arg1, arg2)

This function performs the operation $arg1^{arg2}$. That is, it raises *`arg1`* to the power of *`arg2`*. This function can be used only with constant values and is evaluated at synthesis time.

$reverse(arg)

This function takes the outermost index of the array *`arg`* and reverses it. The most common use is for reversing the indices of a string to make the leftmost letter index 0. This function can be used only with constant values and is evaluated at synthesis time.

$flatten(arg)

This function takes the array *`arg`* and reduces it to a single-dimensional array. For example, an 8 x 2 array would become a 16-bit array. It would be arranged with the 2 LSBs being the 2 bits indexed by 0 for the outermost dimension. This can be used on any signals.

$signed(arg)

This function causes *`arg`* to be interpreted as a signed (two's complement) value. This can be used on any signals.

$unsigned(arg)

This function causes *`arg`* to be interpreted as an unsigned value. This can be used on any signals.

Groups

Groups are indicated by parentheses; we group an expression so that it is evaluated before things outside the parentheses. It takes the following form:

```
'(' expr ')'
```

Array Concatenation

Array concatenation allows you to concatenate two arrays as long as all their dimensions match, excluding the outermost one. For example, you can concatenate a 3 x 8 array with a 4 x 8 array to make a 7 x 8 array. It takes the following form:

```
'c{' expr (',' expr)* '}'
```

Array Duplication

Array duplication allows you to duplicate the outermost dimension of an array. It has the same effect as concatenating an array with itself multiple times. It takes the following form:

```
expr 'x{' expr '}'
```

The first *`expr`* specifies the number of times the outermost dimension of the array, the second *`expr`*, should be duplicated. The first *`expr`* must be a constant expression that can be evaluated at synthesis time.

Array Builder

The *array builder* is used to pack multiple values into an array. Each value must have the same dimensions. It takes the following form:

```
'{' expr (',' expr)* '}'
```

The dimensions of the new array will be one higher than the dimension of any element.

Bitwise Invert

The *bitwise inverting operator* is used to flip the values of all the bits in an expression. In other words, all 0s become 1s, and all 1s become zeros. It takes the following form:

```
'~' expr
```

Logical Invert

The *logical inverting operator* is used to negate the logical value of an expression. If the expression is nonzero, it becomes 0. If it is 0, it becomes 1. It takes the following form:

```
'!' expr
```

Multidimensional arrays are considered to be true if any element is nonzero.

Negate

The *negation operator* performs a two's complement negation on the expression. It takes the following form:

```
'-' expr
```

Multiply

The *multiplication operator* multiplies two values and takes the following form:

```
expr '*' expr
```

Divide

The *division operator* divides two values and takes the following form:

```
expr '/' expr
```

While the division operator *can* be used on signals, it is highly recommended to use this operator only when the result is constant and can be evaluated at synthesis time. If you use it with a dynamic value, the resulting circuit is expensive. Instead, if you need to perform division, you should use Xilinx's CoreGen tool to generate a division module with optimal parameters for your design.

Add and Subtract

Addition and subtraction operators add or subtract two numbers, respectively. They take the following form:

```
expr ('+'|'-') expr
```

When the + symbol is used, the expressions are added together. When the - symbol is used, the second expression is subtracted from the first.

Bit Shifting

Bit shifting allows you to shift the bits in a single-dimensional array either left or right. It takes the following form:

```
expr ('>>'|'<<'|'>>>'|'<<<') expr
```

The first *expr* is shifted by the value of the second *expr*.

If the `>>` operator is used, the array is shifted right and padded with 0s.

If the `<<` operator is used, the array is shifted left and padded with 0s.

If the `>>>` operator is used, the array is shifted right and padded with 0s if the value being shifted is unsigned. If the value is signed, it is padded with the value of the MSB.

If the `<<<` operator is used, the array is shifted left and padded with 0s.

Bitwise Operators

The *bitwise operators* perform a Boolean operation on two arrays bit by bit. The two arrays must have matching dimensions.

It takes the following form:

```
expr ('|'|'&'|'^'|'~^') expr
```

The `|` operator performs a bitwise OR.

The `&` operator performs a bitwise AND.

The `^` operator performs a bitwise XOR.

The `~^` operator performs a bitwise XNOR.

The resulting array has the same dimensions as both inputs.

Reduction Operators

Reduction operators take in only one argument and produce a single bit. They simply perform a Boolean operation on all the bits in an array.

It takes the following form:

```
('&'|'~&'|'|'|'~|'|'^'|'~^') expr
```

The `|` operator ORs all the bits together.

The `~|` operator NORs all the bits together.

The `&` operator ANDs all the bits together.

The ~& operator NANDs all the bits together.

The ^ operator XORs all the bits together.

The ~^ operator XNORs all the bits together.

Comparison Operators

The *comparison operators* allow you to compare two values and create a Boolean value (1 or 0).

It takes the following form:

```
expr ('<'|'>'|'=='|'!='|'<='|'>=') expr
```

The < operator is true when the first *expr* is less than the second *expr*.

The > operator is true when the first *expr* is greater than the second *expr*.

The == operator is true when the first *expr* is equal to the second *expr*.

The == operator is true when the first *expr* is not equal to the second *expr*.

The <= operator is true when the first *expr* is less than or equal to the second *expr*.

The >= operator is true when the first *expr* is greater than or equal to the second *expr*.

Logical AND and OR

The *logical operators* are used to combine multiple logical expressions. True is considered anything nonzero, and false is 0.

They take the following form:

```
expr ('||'|'&&') expr
```

The || operator will return 1 when either *expr* is nonzero, and 0 otherwise.

The && operator will return 1 only when both *expr* are nonzero, and 0 otherwise.

Ternary Operator

The *ternary operator* is basically a compact `if` statement. It allows you to select one of two values based on the logical value of another.

It takes the following form:

```
expr '?' expr ':' expr
```

If the first *expr* is nonzero, the expression takes the value of the second *expr*. However, if it is 0, the expression takes the value of the third *expr*.

The second and third *expr* must be the same size because the result of the expression must be a fixed size.

Literal Values

Literal values can be expressed using numbers in binary, decimal, or hexadecimal as well as strings.

Numbers

Numbers can be specified in binary, decimal, or hexadecimal. The width used to represent the number can be explicitly or implicitly stated.

The most basic way to represent a value is to type it, such as `12`. This defaults to decimal with an implicit width. When the width is implicit, the value will have the minimum number of bits needed to represent the value.

To specify a radix, you prefix the value with `b` for binary, `d` for decimal, or `h` for hexadecimal. For example, `hF2` or `b110110`.

To specify a width, you must specify the radix first and then prefix the whole thing with the number of bits to use. For example, `8h2C` or `12d8`.

When using binary or hexadecimal, you can also use the characters `x` for *don't care* and `z` for *high impedance*.

Strings

Strings are an easy way to get the ASCII values for characters. A *string* consists of double quotes enclosing a set of characters. A single character string, such as `"B"`, is an 8-bit array. However, strings with more than one character become two-dimensional arrays, with the first dimension being the letter index, and the second dimension is the bits of each letter. It is therefore an *N* x 8 array, where *N* is the number of letters in the string. This makes it easy to access the individual letters.

The first letter in the string is the rightmost one. This is opposite of how we read them, but consistent with the right being the least significant value. Because of this, it is common to use `$reverse(`*`arg`*`)` to make the leftmost letter index 0.

global Blocks

`global` blocks give you a way to declare constants and structs that can be used anywhere in your design. They are especially helpful because they allow you to use structs in the ports of your modules. A file can contain any number of `global` blocks.

Each block is named, and that name is used when accessing the constants. The `global` block's name must be unique throughout your design.

They take the following form:

```
'global' name '{' (struct_dec | const_dec)* '}'
```

The *name* of a `global` block must begin with an uppercase letter and can be followed by lower- or uppercase letters, numbers, and underscores. It must also contain at least one lowercase letter.

See "Constant declaration" on page 187 and "Struct declaration" on page 190 for `const_dec` and `struct_dec`, respectively.

To access a globally declared struct or constant, start with the `global` block's name followed by a period and then the name of the constant or struct. See the following example from the OV2640 Interface component:

```
global Camera {
  // structure for storing the image data
  struct image_data {
    end_frame,        // end of frame reached (active high)
    end_line,         // end of line reached (active high)
    new_pixel,        // new pixel (active high)
    pixel [16]        // pixel data (valid when new_pixel = 1)
  }
}

...

    output<Camera.image_data> image // image data
```

Array Size and Bit Selection

Array sizes are used when declaring many types in Lucid to turn them into arrays. *Bit selection* is used to index the various elements inside an array.

Array Size

Array sizes are pretty simple and take the following form:

```
('[' expr ']')*
```

Here *expr* is an expression, but it must have a value that can be determined at synthesis time.

If no sizes are specified, the type is assumed to be 1 bit or one instance. For each size specified, the array will get another dimension. For example, something declared with `[4][8]` would be considered a 4 x 8 two-dimensional array.

Bit Selectors

Bit selectors are used to index into arrays. The most basic kind, simple indices, select a single element from the array. The more complex constant bit and fixed-width bit selectors allow selection of multiple elements.

When indexing into a multidimensional array, only the last index can be a constant bit selector or a fixed-width bit selectors. Every other index must be a simple array index.

Array index

The array index is the simplest form of indexing into an array. It selects a single element and takes the following form:

```
'[' expr ']'
```

Here *expr* is an expression. When indexing into a multidimensional array, the first index specified will index into the leftmost dimension when the array was declared. For example, `sig mySig[4][7]` can be indexed later with `mySig[3]`. This will have a width of 7. To index to a single bit, you could use `mySig[3][6]`.

Constant bit selector

Constant bit selectors allow you to select everything between a fixed start and stop indices. It takes the following form:

```
'[' expr ':' expr ']'
```

Both *expr* elements are expressions, but they both must be constant values that can be determined at synthesis time.

The first *expr* must be greater or equal to the second *expr* in value. These specify an inclusive range to select from the array.

Fixed-width bit selector

The fixed-width bit selector allows you to specify a start index and the number of elements to select above or below that index. The benefit of this over the constant index selector is that the start index doesn't need to be a constant value.

It takes the following form:

```
'[' expr ('+'|'-') ':' expr ']'
```

Again, both *expr* elements are expressions. The first *expr* is the starting index. The second *expr* is the number of elements to select. If the selector uses the `+` symbol, it selects that many elements above the starting index. If it is a `-` symbol, it selects that many elements below the starting index. The starting index itself is always included.

Comments

Comments give you a way to annotate your designs. The tools will ignore everything in comments. Lucid uses C/Java style comments, with single-line and block comments available.

Single-line comments start after `//` and end at the end of the line. Block comments start with `/*` and end with `*/`. These can span multiple lines or begin and end in the same line.