Silice

A language for hardcoding Algorithms into FPGA hardware

October 27, 2020

Silice makes it possible to write algorithms for FPGAs in the same way we write them for processors: defining sequences of operations, subroutines that can be called, and using control flow statements such as `while` and `break` (even good old `goto`s are available!). At the same time, Silice lets you fully exploit the parallelism and niceties of FPGA architectures, describing operations and algorithms that run in parallel and are always active, as well as pipelines. Silice *remains close to the hardware*. When writing an algorithm you are in control of what happens at which clock cycle, with predictable rules for flow control. Clock domains are exposed.

Silice is only a thin layer above the hardware: nothing gets obfuscated away. In fact, Silice compiles to and inter-operates with Verilog: you can directly instantiate and bind with existing modules. When Verilog makes more sense, use it!

Silice is reminiscent of high performance programming in the late 90s (in the demo scene in particular): the then considered high-level C language was commonly interfaced with time-critical ASM routines. This enabled a best-of-both-worlds situation, with C being used for the overall program flow and ASM used only on carefully optimized hardware dependent routines. Silice aims to do the same, providing a thin programmer friendly layer on top of Verilog, while allowing to call low level Verilog modules whenever needed. Silice also favors and exposes parallelism, so as to fully utilize the FPGA architecture.

Oh, and it features a powerful LUA-based pre-processor.

While I developed Silice for my own needs, I hope you'll find it useful for your projects!

GitHub repository: `https://github.com/sylefeb/Silice/`

1 A first example

This first example assumes two signals: a 'button' input (high when pressed) and a 'led' output, each one bit.

The *main* algorithm – the one expected as top level – simply asks the led to turn on when the button is pressed. We will consider several versions – all working – to demonstrate basic features of Silice.

Perhaps the most natural version for a programmer not used to FPGAs would be:

```
algorithm main(input uint1 button,output uint1 led) {
  while (1) {
    led = button;
  }
}
```

This sets 'led' to the value of 'button' every clock cycle. However, we could instead specify a *continuous assignment* between led and button:

```
algorithm main(input uint1 button,output uint1 led) {
  led := button;
}
```

This makes 'led' constantly track the value of 'button'. A very convenient feature is that the continuous assignment can be overridden at some clock steps, for instance:

```
algorithm main(input uint1 button,output uint1 led) {
  led := 0;
  while (1) {
    if (button == 1) {
      led = 1;
    }
  }
}
```

Of course this last version is needlessly complicated (the previous one was minimal), but it shows the principle: 'led' is continuously assigned 0, but this will be overridden whenever the button is pressed. This is useful to maintain an output to a value that must change only on some specific event (e.g. producing a pulse).

Contents

2 Terminology

Some terminology we use next:

- **VIO**: a Variable, Input or Output.
- **Combinational chain**: A chain of dependent operations implemented as combinational hardware.
- **Combinational loop**: A combinational chain where a cyclic dependency exists. These lead to unstable hardware synthesis and have to be avoided (but in a few rare cases).
- **Host hardware framework**: The Verilog glue to the hardware meant to run the design. This can also be a glue to Icarus[1] or Verilator[2], both frameworks are provided.

3 Basic language constructs

3.1 Types

Silice supports signed and unsigned integers with a specified bit width:

- `int`N with N the bit-width, e.g. `int8`: signed integer.
- `uint`N with N the bit-width, e.g. `uint8`: unsigned integer.

[1] http://iverilog.icarus.com/
[2] https://www.veripool.org/wiki/verilator

3.2 Constants

Constants may be given directly as decimal based numbers (eg. `1234`), or can be given with a specified bit width and base:

- `3b0101`, 3 bits wide value 5.
- `32hffff`, 32 bits wide value 65535.
- `4d10`, 4 bits wide value 10.

Supported base identifiers are: `b` for binary, `h` for hexadecimal, `d` for decimal. If the value does not fit the bit width, it is clamped.

3.3 Variables

Variables are declared with the following pattern:
`TYPE ID = VALUE;`
where `TYPE` is a type definition (Section 3.1), `ID` a variable identifier (starting with a letter followed by alphanumeric or underscores) and `VALUE` a constant (Section 3.2).

The initializer is mandatory, and is always a simple constant (no expressions), or the special value `uninitialized`. The later indicates that the initialization can be skipped, reducing design size. This is particularly interesting on brams/broms and register arrays.

3.4 Tables

`intN tbl[M] = {...}`
Example: `int6 tbl[4] = {0,0,0,0};`

Table sizes have to be constant at compile time. The initializer is mandatory and can be a string, in which case each letter becomes its ASCII value, and the string is null terminated.

The table size `M` is optional (eg. `int4 tbl[]={1,2,3};`) in which case the size is automatically derived from the initializer (strings have one additional implicit character: the null terminator).

If the table size is specified and the initializer is a shorter string, the table is automatically padded with zeros. A longer string results in an error.

If the table size is specified and the initializer – not a string – has a different number of elements, an error occurs. However, if a smaller number of elements is given the last element can be `pad(value)` in which case the remainder of the table is filled with `value`.
Example: `int6 tbl[256] = {0,0,0,0,pad(255)};`
Example: `int6 tbl[256] = {pad(0)};`

The keyword `uninitialized` may be used to explicitly skip table initialization, in which case the initial table state is unknown: `int6 tbl[4] = uninitialized;`.

Similarly, the padding can be uninitialized: `int6 tbl[256] = {0,0,0,0,pad(uninitialized)};`. In this case, only the first part of the table will be initialized, the state of the other values will be unknown.

3.5 Block RAMs / ROMs

`bram intN tbl[M] = {...}`
Block RAMs are declared in a way similar to tables. Block RAMs map to special FPGA blocks and avoid using FPGA LUTs to store data. However, accessing a block RAM typically requires a one cycle latency.

A block RAM variable has four members accessible with the 'dot' syntax:

1. `addr` the address being accessed,
2. `wenable` set to 1 if writing, set to 0 if reading,

3. `rdata` result of read,
4. `wdata` data to be written.

Here is an example of using a block RAM in Silice:

```
bram int8 table[4] = {42,43,44,45};
int8 a = 0;

table.wenable = 0; // read
table.addr    = 2; // third entry
++:                // wait on clock
a = table.rdata;   // now a == 44
```

Finally, an advanced option lets you indicate that the BRAM inputs need not be latched:

```
bram int8 table<input!>[4] = uninitialized;
```

This is used to minimize a design size ; however the caller has to ensure values are all properly present at the right cycle when reading/writing.
The rules for initializers of BRAMs are the same as for tables.

Block ROMs ROMs use a similar syntax, using `brom` instead of `bram`.

Dual-port RAMs A dual-port block RAM has eight members accessible with the 'dot' syntax:

1. `addr0` the address being accessed on port 0,
2. `wenable0` write enable on port 0,
3. `rdata0` result of read on port 0,
4. `wdata0` data to be written on port 0,
5. `addr1` the address being accessed on port 1
6. `wenable1` write enable on port 1,
7. `rdata1` result of read on port 1,
8. `wdata1` data to be written on port1.

The dual-port BRAM also has optional clock parameters for port0 and port1.
Here is an example of using a block RAM in Silice with different clocks:

```
dualport_bram int8 table<@clock0,@clock1>[4] = {42,43,44,45};
```

3.6 Operators

3.6.1 Swizzling

```
int6 a = 0;
int1 b = a[1,1]; // second bit
int2 c = a[1,2]; // int2 with second and third bits
int3 d = a[2,3]; // int3 with third, fourth and fifth bits
```

The first entry may be an expression, the second has to be a constant.

3.6.2 Arithmetic, comparison, bit-wise, reduction, shift

All standard verilog operators are supported, binary and unary.

3.6.3 Concatenation

Concatenation allows to combine expressions to form expressions having larger bit-width. The syntax is to form a comma separated list enclosed by braces. Example:

```
  int6  i = 6b111000;
  int2  j = 2b10;
  int10 k = 0;
  k := {j,i,2b11};
```

Here c is obtained by concatenating a,b and the constant 2b11 to form a ten bits wide integer. Bits can also be replicated with a syntax similar to Verilog:

```
  uint2  a = 2b10;
  uint9  c = 0;
  c := { 1b0, {8{a[1,1]}} };
```

Here the last eight bits of variable c are set to the second bit of a.

3.6.4 Bindings

Silice defines operators for bindings the input/output of algorithms and modules:

- `<:` binds right to left
- `:>` binds left to right
- `<:>` binds both ways

The bound sides have to be VIO identifiers. To bind expressions you can use expression tracking (see Section 3.6.6).

Bound VIOs are connected and immediately track each others values. A good way to think of this is as a physical wire between IC pins, where each VIO is a pin. Bindings are specified when instantiating algorithms and modules.

The bidirectional binding is reserved for two use cases:

1. binding `inout` variables,
2. binding groups (see Section 3.7) and interfaces (see Section 3.8).

Advanced There are two specialized versions of the binding operators:

- `<::` binds right to left, using value at prior clock rising edge,
- `<::>` binds an IO group, using value at prior clock rising edge for inputs.

Normally, the bound inputs are tracking the value of the bound VIO as they are changing during the current clock cycle. Therefore, the tracking algorithm/module immediately gets new values (in other words, there is a combinatorial connection). However, in specific cases one needs to bind the value as it was at the last (positive) clock edge, ignoring ongoing changes. This may, for instance, be necessary to avoid a combinatorial cycle.

3.6.5 Always assign

Silice defines operators for continuous assignment of VIOs. A continuous assignment is performed at each clock rising edge, regardless of the execution state of the algorithm. The left side of the assignment has to be a VIO identifier, while the right side may be an expression.

- `:=` assign right to left at each rising clock.
- `::=` assign right to left with a one clock cycle delay (two stages flip-flop for e.g. clock domain crossing).

Continuous assignments are specified just after variable declarations and algorithms/modules instantiations. A typical use case is to make something pulse high, for instance:

```
algorithm writer(
  output uint1 write, // pulse high to write
  output uint8 data,  // byte to write
  // ...
) {

  write := 0; // maintain low with always assign
  // ...
  if (do_write) {
    data  = ...;
    write = 1; // pulses high on next rising clock
  }

}
```

Important: the assignment is performed immediately after the clock rising edge. If the value of the expression in the right hand side is later changing (during the clock cycle), this will not change the value of the left hand side. To achieve this use *expression tracking* (see next).

3.6.6 Expression tracking

Variables can be defined to constantly track the value of an expression. This is done during the variable declaration, using either the *:=* or *::=* operators. Example:

```
algorithm adder(
  output uint9 o,
  // ...
) {
  uint8 a = 1;
```

```
6     uint8 b = 2;
7     uint9 a_plus_b := a + b;
8
9     a = 15;
10    b = 3;
11    o = a_plus_b;
12  }
```

In this case *o* gets assigned 15+3 on line 11, as it tracks immediate changes to the expression *a+b*. Note that the variable *a_plus_b* becomes read only (in Verilog terms, this is now a wire). We call *o* an expression tracker.

The second operator, *::=* tracks the expression using the values of the variables as they where at the previous clock rising edge. If used in this example, o would be assigned 1+2.

Expression trackers can refer to other trackers, however trackers using *:=* and *::=* cannot be mixed.

3.7 Groups

Often, we want to pass around variables that form conceptual groups, like the interface to a controller. Silice has a specific mechanism to help with that. A group is declared as:

```
1   // SDRAM interface
2   group sdram_32b_io
3   {
4     uint24 addr       = 0,
5     uint1  rw         = 0,
6     uint32 data_in    = 0,
7     uint32 data_out   = 0,
8     uint1  busy       = 1,
9     uint1  in_valid   = 0,
10    uint1  out_valid  = 0
11  }
```

A group variable can be declared directly:

```
1   sdram_32b_io sd; // init values are defined in the group declaration
```

To pass a group variable to an algorithm, we need to define an interface (see also Section 3.8). This will further specify which group members are input and outputs:

```
1   algorithm sdramctrl(
2     // ..
3     // interface
4     sdram_provider sd {
5       input   addr,       // address to read/write
6       input   rw,         // 1 = write, 0 = read
7       input   data_in,    // data from a read
8       output  data_out,   // data for a write
9       output  busy,       // controller is busy when high
10      input   in_valid,   // pulse high to initiate a read/write
```

```
    output  out_valid   // pulses high when data from read is
}
) {
  // ..
```

Binding to the algorithm is then a single line ; in the parent:

```
sdramctrl memory(
  sd <:> sd,
  // ..
);
```

Note: A group cannot contain tables.

3.8 Interfaces

Interfaces are ways to describe what inputs and outputs an algorithm expects, without knowing in advance the exact specification of these fields (e.g. their widths). Besides making algorithm IO description more compact, this provides genericity. Interfaces are meant to be bound to groups (Section 3.7).

Reusing the example of Section 3.7 we can define an interface ahead of time:

```
interface sdram_provider {
  input   addr,       // address to read/write
  input   rw,         // 1 = write, 0 = read
  input   data_in,    // data from a read
  output  data_out,   // data for a write
  output  busy,       // controller is busy when high
  input   in_valid,   // pulse high to initiate a read/write
  output  out_valid   // pulses high when data from read is
}
```

And then the sdram controller algorithm declaration simply becomes:

```
  algorithm sdramctrl(
    // interface
    sdram_provider sd,
    // ..
  ) {
    // ..
```

Note than the algorithm is now using a generic interface, it does not know in advance the width of the signals in the interface. This is determined at binding time, when a group is bound to the interface. Interfaces can be bound to other interfaces, the information is propagated when a group is bound at the top level.

For instance, we may define another group as follows, which uses 8 bits instead of 32 bits:

```
  // SDRAM interface
  group sdram_8b_io
  {
    uint24 addr       = 0,
    uint1  rw         = 0,
    uint8  data_in    = 0,
    uint8  data_out   = 0,
    uint1  busy       = 1,
    uint1  in_valid   = 0,
    uint1  out_valid  = 0
  }
```

This group can be bound to the `sdramctrl` controller, which now will receive 8 bits signals for `data_in`/`data_out`.

The `sdramctrl` controller can adjust to this change, using `sameas` and `widthof`. The first, `sameas`, allows to declare a variable the same as another:

```
  sameas(sd.data_in) tmp;
```

The second, `widthof`, returns the width of a signal:

```
  while (i < (widthof(sd.data_in) >> 3)) {
    // ...
  }
```

Note: In the future Silice will provide compile time checks, for instance verifying the width of a signal is a specifc value.

3.9 Bitfields

There are many cases where we pack multiple data in a larger variable, for instance:

```
  uint32 data = 32hffff1234;
  // ..
  left  = data[ 0,16];
  right = data[16,16];
```

The hard coded values make this type of code quite hard to read and difficult to maintain. To cope with that, Silice defines bitfields. A bitfield is first declared:

```
bitfield Node {
  uint16 right,
  uint16 left
}
```

Note that the first entry in the list is mapped to the higher bits of the field.

This can then be used during access, to unpack the data:

```
left  = Node(data).left;
right = Node(data).right;
```

There is no overhead to the mechanism, and different bitfield can be used on a same variable depending on the context (e.g. instruction decoding).

The bitfield can also be used to initialize the wider variable:

```
uint32 data = Node(left=16hffff,right=16h1234);
```

3.10 Intrinsics

Silice has convenient intrinsics:

- `__signed(exp)` indicates the expression is signed (Verilog `$signed`)
- `__unsigned(exp)` indicates the expression is unsigned (Verilog `$unsigned`)
- `__display(format_string,value0,value1,...)` maps to Verilog `$display` allowing text output during simulation.

4 Algorithms

Algorithms are the main elements of a Silice design. An algorithm is a specification of a circuit implementation. Thus, like modules in Verilog, algorithms have to be instanced before being used. Each instance becomes an actual physical layout on the final design. An algorithm can instance other algorithms and Verilog modules.

Instanced algorithms *always run in parallel*. However they can be called synchronously (implying a wait state in the caller). They may run forever and start automatically. Each may be driven from a specific clock and reset signal.

main (entry point). The top level algorithm is called *main* and has to be defined. It is automatically instanced by the *host hardware framework*, see Section 8.

4.1 Declaration

An algorithm is declared as follows:

```
algorithm ID (
input TYPE ID
...
output TYPE ID
...
output! TYPE ID
...
) <MODS> {
  DECLARATIONS
  SUBROUTINES
  ALWAYS_ASSIGNMENTS
  ALWAYS_BLOCK
  INSTRUCTIONS
}
```

Most elements are optional: the number of inputs and outputs can vary, the modifiers may be empty (in which case the `'<>'` is not necessary) and declarations, bindings and instructions may all be empty.

Here is a simple example:

```
algorithm adder(intput uint8 a,intput uint8 b,output uint8 v)
{
  v = a + b;
}
```

Let us now discuss each element of the declaration.

Inputs and outputs. Inputs and outputs may be declared in any order, however the order matters when calling the algorithms (parameters are given in the order of inputs, results are read back in the order of outputs). Input and outputs can be single variables or tables. A third type `inout` exists for compatibility with Verilog modules, however these can only be passed and bound to imported modules (i.e. they cannot be used in expressions and instructions for now).

Note that there are two types of output: `output` and `output!`. These distinguish between a standard and a combinational output (exclamation mark). A standard output sees its value updated at the next clock cycle. A combinational output ensures that callers sees the results within the same combinational chain. This is for instance important when implementing a video driver; e.g. a HDMI module generates a pixel coordinate on its output and expects to see the corresponding color on its input within the same clock step. Thus, the algorithm outputting the color to the HDMI driver will input the coordinate and use `output!` for the color. However, using only `output!` will in some cases result in combinational loops, leading to unstable circuits.

Declarations. Variables, instanced algorithms and instanced modules have to be declared first (in any order). A simple example:

```
algorithm main(output uint8 led)
{
  uint8 r = 0;
  adder ad1;

  // ... btw this is a comment
}
```

4.2 Instantiation

Algorithms and modules can be instanced from within a parent algorithm. Optionally parameters can be *bound* to the parent algorithm variables. In terms of hardware, these are the wires connecting instanced and parent algorithms. Instantiation uses the following syntax:

```
MOD_ALG ID<@CLOCK,!RESET> (
BINDINGS
(<:auto:>)
);
```

where `MOD_ALG` is the name of the module/algorithm, `ID` an identifier for the instance, and `BINDINGS` a comma separated list of bindings between the instance inputs/outputs and the parent algorithm

variables. The bindings are optional and may be partial. `@CLOCK` optionally specifies a clock signal, `!RESET` a reset signal, where `CLOCK` and `RESET` have to be `uint1` variables in the parent algorithm. Both are optional, and if none is specified the brackets `<>` can be skipped. When none are specified the parent `clock` and `reset` are used for the instance.

Each binding is defined as: `ID_left OP ID_right` where `ID_left` is the identifier of an instance input or output, `OP` a binding operator (Section 3.6.4) and `ID_right` a variable identifier.

Note that only identifiers are allowed in bindings: access to tables and swizzling are not allowed. This can be alleviated using expression tracking, see Section 3.6.6.

Combined with `autorun` such bindings allow to instantiate and immediately run an algorithm to drive some of the parent algorithm variables. Here is an example:

```
algorithm blink(output int1 b_fast,output int1 b_slow) <autorun>
{
  uint20 counter = 0;
  while (1) {
    counter = counter + 1;
      if (counter[0,8] == 0) {
        b_fast = !b_fast;
      }
      if (counter == 0) {
        b_slow = !b_slow;
      }
  }
}

algorithm main(output int8 led)
{
  int1 a = 0;
  int1 b = 0;
  blink b0(
    b_slow :> a,
    b_fast :> b
  );

  led := 0;       // turn off all eight LEDs
  led[0,1] := a; // first LED tracks slow blink
  led[1,1] := b; // second LED tracks fast blink

  while (1) { } // inifinite loop, blink runs in parallel
}
```

This example reveals some interesting possibilities, and a constraint. As algorithm `blink` is instanced as `b0`, it starts running immediately (due to the `autorun` modifier in the declaration of `blink`). Hence, the variables `a` and `b` are immediately tracking the `b_slow` and `b_fast` outputs of `b0`. The always assignment to `led` then use `a` and `b` to output to the `led` 8-bit variable, which we assume is physically connected to LEDs. The first assignment sets `led` to zero, and then its two first bits to `a` and `b`. As the always assignments are order dependent, this behaves properly with the two first bits always assigned at each clock cycle. Since all updates in `main` are done through bindings and alway assignments, and because *algorithms run in parallel*, there is nothing else to do, and `main` enters an infinite loop (runs as long as there is power).

The constraint, however, is that `a` and `b` are necessary. This is due to the fact that we cannot directly bind `led[0,1]` and `led[1,1]` to the instance of `blink`. Only identifiers can be specified during bindings. This can be alleviated by using expression tracking (see Section 3.6.6).

Automatic binding. The optional `<:auto:>` tag allows to automatically bind matching identifiers: the compiler finds all valid left/right identifier pairs having the same name and binds them directly. While this is convenient when many bindings are repeated and passed around, it is recommended to use groups for brevity and make all bindings explicit.

Direct access to inputs and outputs. The inputs and outputs of instanced algorithms, when not bound, can be directly accessed using a 'dot' notation. Outputs can be read and inputs written to. Example:

```
algorithm algo(input uint8 i,output uint o)
{
  o = i;
}

algorithm main(output uint8 led)
{
  algo a;
  a.i = 131;
  () <- a <- (); // this is a call without explicit inputs/outputs
  led = a.o;
}
```

4.3 Call

Algorithms may be called synchronously or asynchronously, with or without parameters. Only instanced algorithms may be called.

The asynchronous call syntax is as follows:

```
ID <- (P_0, ..., P_{N-1}); // with parameters
ID <- (); // without parameters
```

Where `ID` is the instanced algorithm name. When the call is made with parameters, all have to be specified. The parameters are given in the same order they are declared in the algorithm being called. The call without parameters is typically used when inputs are bound or specified directly. Note that when parameters are bound, only the call without parameters is possible.

When an asynchronous call is made, the instanced algorithm will start processing at the next clock cycle. The algorithm runs in parallel to the caller and any other instanced algorithm. To wait and obtain the result, the join syntax is used:

```
(V_0, ..., V_{N-1}) <- ID; // with receiving variables
() <- ID; // without receiving variables
```

Note: Calling algorithms across clock domains is not yet supported. For such cases, autorun with bindings is recommended.

4.4 Subroutines

Algorithms can contain subroutines. These are local routines that can be called by the algorithm multiple times. A subroutine takes parameters, and has access to the variables, instanced algorithms and instanced modules of the parent algorithm – however access permissions have to be

explicitly given. Subroutines offer a simple mechanism to allow for the equivalent of local functions, without having to wire all the parent algorithm context into another module/algorithm. Subroutines can also declare local variables, visible only to their scope. Subroutines avoid duplicating a same functionality over and over, as their code is synthesized a single time in the design, regardless of the number of times they are called.

A subroutine is declared as:

```
subroutine ID(
input TYPE ID
...
output TYPE ID
...
reads ID
...
writes ID
...
readwrites ID
...
calls ID
) {
  DECLARATIONS
  INSTRUCTIONS
  return;
}
```

(return is optional)

Here are simple examples:

```
algorithm main(output uint8 led)
{
  uint8  a       = 1;

  subroutine shift_led(readwrites a) {
    a = a << 1;
    if (a == 0) {
      a = 1;
    }
  }

  subroutine wait() {
    uint20 counter = 0;
    while (counter != 0) {
      counter = counter + 1;
    }
  }

  led := a;

  while(1) {
    () <- wait <- ();
    () <- shift_led <- ();
```

```
  }
}
```

Subroutines permissions. Subroutine permissions ensure only those variables given read/write permission can be manipulated. This mitigates the fact that a subroutine may directly manipulate variables in the parent algorithm. The format for the permissions is a comma separated list using keywords `reads`, `writes`, `readwrites`

Why subroutines? There is a fundamental difference between a subroutine and an algorithm called from a host: the subroutine never executes in parallel, while a child algorithm could. However, if this parallelism is not necessary, subroutines offer a simple mechanism to repeat internal tasks.

Global subroutines Subroutines may be declared outside of an algorithm. Such subroutines are called *global* and may be called from any algorithm. This is convenient to share subroutines across algorithms. If a global subroutine requests access to parent algorithm variables (read/write permissions), an algorithm calling the subroutine has to have a matching set of variables in its scope. Note that global subroutines are not technically shared, but rather copied in each calling algorithm's scope at compile time.
Nested calls

Subroutines can call other subroutines, and have to declare this is the case using the keyword `calls`.

```
subroutine wait(input uint24 delay)
{
  // ...
}
subroutine init_device(calls wait)
{
  // ...
  () <- wait <- (1023);
}
```

Note that re-entrant calls (subroutine calling itself, even through other subroutines) are not possible.

4.5 Circuitry

Sometimes, it is useful to write a generic piece of code that can be instantiated within a design. Such an example is a piece of circuitry to write into an SDRAM, which bit width may not ne known in advance.

A circuitry offers exactly this mechanism in Silice. It is declared as:

```
circuitry ID(
input ID
output ID
inout ID
) {
INSTRUCTIONS
}
```

Note that there is no type specification on inputs/outputs as these are resolved during instantiation. Here is an example of circuitry:

```
circuitry writeData(inout sd,input addr,input data) {
  // wait for sdram to not be busy
  while (sd.busy) { /*waiting*/ }
  // write!
  sd.data_in      = rdata;
  sd.addr         = addr;
  sd.in_valid     = 1;
}
```

Note the use of inout for sd (which is a group, see Section 3.7). A circuitry is not called, it is instantiated. This means that every instantiation is indeed a duplication of the circuitry.

Here is the syntax for instantiation:

```
(output_0,...,output_N) = ID(input_0,...input_N)
```

As for algorithms and subroutines, inputs and outputs have to appear in the order of declaration in the lists. An inout appears twice, both as output and input. Following the previous example here is the instantiation from an algorithm:

```
(sd) = writeData(sd,myaddr,abyte);
```

Note: currently circuitry instantiation can only be made on VIO identifiers (no expressions, no bit-select or part-select. This restriction will be removed at some point.

4.6 Combinational loops

When writing code, you may encounter a case where the Silice compiler raises an error due to an expression leading to a combinational loop. This indicates that the sequence of operations is leading to a cyclic dependency in the combinational chain. These are in most cases undesirable as they lead to unpredictable hardware behaviors.

A trivial solution is such a situation is to split the combinational chain using the *step* operator (see Section 5.1). However, doing so automatically would introduce sequencing 'behind the curtain' which we absolutely avoid in Silice, as this can lead to all sort of synchronization issues (see Section 5 for more information). Instead, an error is reported and you are invited to either manually add a step, or to revise the code. In many cases a slight rewrite avoids the issue entirely.

Example of a combinational loop:

```
algorithm main()
{
  uint8 a = 0;
  a = a + 1; // triggers a combinational loop error
}
```

So what happens? It might seem that `a = a + 1` is the problem here, as it writes as a cyclic dependency. In fact, that is not the sole cause. On its own, this expression is perfectly fine: for each variable Silice tracks to versions, the one of the previous clock tick and the one of the current. So `a = a + 1` in fact means $a_{current} = a_{previous} + 1$.

The problem here comes from the initialization of a, in the declaration. This already set a new value to $a_{current}$, and thus the next expression `a = a + 1` would have to use this new value (otherwise it would be ignored entirely!). This now leads to $a_{current} = a_{current} + 1$ which this time is a combinational loop.

The solution is simple:

```
algorithm main()
{
  uint8 a = 0;
  ++: // wait one cycle
  a = a + 1; // now this is fine
}
```

It would be difficult to manually keep track of all these potential chains, which is why Silice does it for you! In practice, such loops are rarely encountered, and in most cases easily resolved with slight changes to arithmetic and logic.

Note: combinational loops *across* algorithms (i.e. through `output!`) are currently not detected.

4.7 Always assignments

After declarations, optional always assignments can be specified. These use the operators defined in Section 3.6.5. Always assignments allow to track the value of an expression in a variable, output, or instanced algorithm input or output.
Example:

```
algorithm main(output uint8 led)
{
  uint8 r = 0;
  adder ad1;

  led   := r;
  ad1.a := 1;
}
```

These assignments are performed immediately after each rising clock and are order dependent. For instance:

```
b := a;
c := b;
```

is not the same as:

```
c := b;
b := a;
```

In the first case, c will immediately take the value of a, while in the second case c will take the value of a after one clock cycle. This is useful, for instance, when crossing a clock domain as it allows to implement a two stages flip-flop. In fact, Silice provides a shortcut for exactly this purpose, making the temporary variable b unnecessary:

```
c ::= a;
```

4.8 Always block

An algorithm can have an always block. This block has to be combinational (no goto, while or step operator inside). It is always running, regardless of the state of the rest of the algorithm, and is executed immediately after always assignments, and before anything else.

The syntax simply is:

```
always {
  // instructions
  // ...
}
```

Assigning a variable in an always block or using an always assignment is in fact equivalent, that is:

```
always {
  a = 0;
}
```

is functionally the same as

```
a := 0;
```

4.9 Clock and reset

All algorithms receive a `clock` and `reset` signal from their parent algorithm. These are intrinsic variables, are always defined within the scope of an algorithm and have type `int1`. The clock and reset can be explicitly specified when an algorithm is instanced (Section 4.2).

4.10 Modifiers

Upon declaration, modifiers can be specified (see `<MODS>`) in the declaration). This is a comma separated list of any of the following:

- **Autorun.** Adding the `autorun` keyword will ask the compiler to run the algorithm upon instantiation, without waiting for an explicit call.
- **Internal clock.** Adding a `@ID` specifies the use of an internally generated clock signal. It is then expected that the algorithm contains a `int1 ID = 0;` variable declaration, which is bound to the output of a module producing a clock signal. This is meant to be used together with Verilog PLLs, to produce new clock signals. The module producing the new clock will typically take `clock` as input and `ID` as output.
- **Internal reset.** Adding a `!ID` specifies the use of an internally generated reset signal. It is then expected that the algorithm contains a `int1 ID = 0;` variable declaration, which is bound to the output of a module producing a reset signal. This may be used, for instance, to filter the signal from a physical reset button.

- **Stack.** Adding the `stack:N` modifier (with N an integer ¿ 1) will allocate a stack of size N for the algorithm subroutine nested calls.
- **One-hot.** Adding the `onehot` modifier will use a 'onehot' state numbering for the algorithm state machine. On small algorithms with few states this can result in smaller, faster designs.

Here is an example:

```
algorithm main(output int1 b) <autorun,@new_clock>
{
  int1 new_clock = 0;

  my_pll pll(
    base_clock <: clock,
    gen_clock :> new_clock
  );

  // the main algorithm is sequenced by new_clock
  // ...
}
```

Which signals are allowed as clocks and resets depends on the FPGA architecture and vendor toolchain.

5 Execution flow and cycle utilization rules

Upon compilation, Silice breaks down the code into a finite state machine (FSM) and corresponding combinational chains. Silice attempts to form the longest combinational chains, or equivalently to minimize the number of states in the FSM. That is because going from one state to the next requires one clock cycle, delaying further computations. Of course, longer combinational chains also lead to reduced clock frequency, so there is a tradeoff. This is why Silice lets you explicitly specify where to cut a combinational chain using the step operator `++:`

Note that if a state contains several independent combinational chain, they will execute as parallel circuits (e.g. two computations regarding different sets of variables). Grouping such computations in a same state increases parallelism, without deepening the combinational chains.

The use of control flow structures such as `while` (or `goto`), as well as synchronization with other algorithm also require introducing states. This results in additional cycles being introduced. Silice follows a set of precise rules to do this, so you always know what to expect.

5.1 The step operator

Placing a `++:` in the sequence of operations of an algorithm explicitly asks Silice to wait for exactly one cycle at this precise location in the execution flow. Each next `++:` waits one additional cycle.

This has several important applications such as waiting for a memory read/write, or breaking down long combinational chains that would violate timing.

5.2 Control flow

The core control flow operations are `goto`, `call` and `return` (`call` is a `goto` preparing a return address for `return`). While `goto` and `call` may be used directly, it is recommended to prefer higher level primitives such as `while` and subroutines, because `goto` often leads to harder to read

and maintain code[3]. Yet, they are a fundamental low-level operation and Silice is happy to expose them for your enjoyment!

`goto` and `call` always require one cycle to 'jump': this is a change of state in the FSM. Entering a `while` takes one cycle and then, if there is a single combinational chain inside, it takes exactly one cycle per iteration.

Now, `if-then-else` is slightly more subtle. When applied to sequences of operations not requiring any control flow, an `if-then-else` synthesizes to a combinational `if-then-else`. This means that both the 'if' and 'else' code are evaluated in parallel as combinational chains, and a multiplexer selects which one is the result based on the condition. This applies recursively, so combinational `if-then-else` may be nested.

When the `if-then-else` contains additional control flow (e.g. a subroutine call, a `while`, a `++:`, a `goto`, etc.) it is automatically split into several states. It then takes one cycle to exit the 'if' or 'else' part and resume operations. If only the 'if' or the 'else' requires additional states while the other is a single combinational chain (possibly empty), an additional state is still required to 'jump' to what comes after the `if-then-else`. So in general this will cost one cycle. However, in cases where this next state already exists, for instance when the `if-then-else` is at the end of a `while` loop, this existing state is reused resulting in no overhead.

This has interesting implications. Consider the following code:

```
1  while(...) {
2    if (condition) {
3      ...
4  ++:
5      ...
6    }
7    a = b + 1;
8  }
```

When the `if` is not taken, we still have to pay one cycle to reach line 7. That is because it has been place into a separate state since the `if` contains non-combinational code that has to jump to reach it. Hence, the while loop will always take at least two cycles.

However, you may choose to duplicate some code (and hence some circuitry!) to avoid the extra cycle, rewriting as:

```
1  while(...) {
2    if (condition) {
3      ...
4  ++:
5      ...
6      a = b + 1;
7    } else {
8      a = b + 1;
9    }
10 }
```

This works because when the `if` is taken the execution simply goes back to the start of the `while`, a state that already exists. The `else` being combinational and also followed by the start of the `while` no extra cycle is necessary! We just tradeoff a bit of circuit size for extra performance.

[3]See the famous Dijkstra's paper about this [?]

5.3 Cycle costs of calls to algorithms and subroutines

A synchronized call to an algorithm takes at best two cycles: one cycle for the algorithm to start processing, and one cycle to register that the algorithm is done. An asynchronous call is a combinational operation (and the algorithm will start running on the next clock cycle), while a synchronization inserts a state in the control flow and always requires at least one cycle, even if the algorithm is already finished.

A synchronized call to a subroutine takes at best two cycles: one cycle to jump to the subroutine initial state, and one cycle to jump back.

Note: Keep in mind that algorithms can also autorun and have their inputs/outputs bound to variables (see Section 4.2). Algorithms may also contain a combinational `always` block that is always running.

5.4 Pipelining

TODO: describe syntax `{ } -> { }`

6 Lua preprocessor

A preprocessor allows to change the way the source code is seen by the compiler. It sits between the input source code and the compiler, and basically rewrites the original source code in a different way, before it is input to the compiler. This allows compile-time behaviors, such as adapting the code to a target platform.

Having a strong preprocessor is important for hardware design. For instance, designing a sort network of some size N is best done by generating the code automatically from a preprocessor, so that one can easily reuse the same code for different values of N. The same is true of a division algorithm for some bit width, or for the pre-computations of lookup tables.

The Silice preprocessor is built above the Lua language `https://lua.org`. Lua is an amazing powerful, lightweight scripting language that originated in the Computer Graphics community. The preprocessor is thus not just a macro system, but a fully fledged programming language.

6.1 Principle and syntax

Preprocessor code is directly interleaved with Silice code. At any point, the source code can be escaped to the preprocessor by starting a line with $$. The full line is then considered as preprocessor Lua code.

The pre-processor sees Silice source lines as strings to be output. Thus, it outputs to the compiler any line that is met during the execution of the preprocessor. This means it does not output those that are not reached, and it outputs multiple times those that appear in a loop.

Let's make a quick example:

```
1  algorithm main()
2  {
3
4  $$if true then
5    uint8 a=0;
6    uint8 b=0;
7  $$end
8
9  $$if false then
10 uint8 c=0;
11 $$end
```

```
12
13  $$for i=0,1 do
14  b = a + 1;
15  $$end
16  }
```

The code output by the preprocessor is:

```
1   algorithm main()
2   {
3
4   uint8 a=0;
5   uint8 b=0;
6
7   b = a + 1;
8   b = a + 1;
9
10  }
```

Note that source code lines are either fully Silice, of fully preprocessor (starting with $$). A second syntax allows to use preprocessor variables directly in Silice source code, simply doing $varname$ with `varname` the variable from the preprocessor context. In fact, a full Lua expression can be used in between the $ signs; this expression is simply concatenated to the surrounding Silice code.

Here is an example:

```
1   algorithm main()
2   {
3   $$for i=0,3 do
4   uint8 a_$i$ = $100+i$;
5   $$end
6   }
```

The code output by the preprocessor is:

```
1   algorithm main()
2   {
3   uint8 a_0 = 100;
4   uint8 a_1 = 101;
5   uint8 a_2 = 102;
6   uint8 a_3 = 103;
7   }
```

If a large chunk of Lua code has to be written, you can simply write it in a separate `.lua` file and use:

```
1   $$dofile('some_lua_code.lua')
```

6.2 Includes

The preprocessor is also in charge of including other Silice source code files, using the syntax `$include('source.ice')`. This loads the entire file (and recursively its own included files) into the preprocessor execution context.

6.3 Pre-processor image and palette functions

The pre-processor has a number of function to facilitate the inclusion of images and palette data ; please refer to the example projects `vga_demo`, `vga_wolfpga` and `vga_doomchip`.

7 Interoperability with Verilog modules

Silice can inter-operate with Verilog.

Verilog source code can be included in two different ways: `append('code.v')` and `import('code.v')` where `code.v` is a Verilog source code file.

7.1 Append

Append is very simple: the Verilog source code is directly appended to the output of the compiler (which is a Verilog source code) without any processing. This makes this Verilog code available for other Verilog modules, in particular those that are imported (see next). The reason for `append` is that Silice is currently not able to fully parse Verilog when importing.

7.2 Import

Import is the most interesting way to inter-operate with Verilog. Once imported, all Verilog modules from the Verilog source file will be available for inclusion in algorithms.

The modules are instantiated in a very similar way as algorithms, with bindings to variables. However, modules cannot be called like algorithms, and the 'dot' syntax to read/write outputs and inputs is not available for modules.

7.3 Wrapping

Due to the fact that Silice only understands a subset of Verilog, there are cases where a module cannot be imported directly. In such cases, wrapping offers a solution. The idea is to write a simpler Verilog module that can be imported by Silice and wraps the instantiation of the more complex one.

Then the complex module is included with `append` and the wrapper with `import`.

8 Host hardware frameworks

The host hardware framework typically consists in a Verilog glue file. The framework is appended to the compiled Silice design (in Verilog). The framework instantiates the *main* algorithm. The resulting file can then be processed by the vendor or open source FPGA toolchain (often accompanied by a hardware constraint file).

Silice comes with the following host hardware frameworks:

- **mojo_basic** : framework to compile for the Alchitry Mojo 3 FPGA board.
- **mojo_hdmi_sdram** : framework to compile for the Alchitry Mojo 3 FPGA board equipped with the HDMI shield.
- **icestick** : framework for the Lattice ice40 icestick board.

- **verilator_bare** : framework for use with the *verilator* hardware simulator
- **verilator_sdram_vga** : framework for use with the *verilator* hardware simulator, with VGA and SDRAM emulation
- **icarus_bare** : framework for use with the *icarus*
- **icarus_vga** : framework for use with the *icarus* hardware simulator, with vga emulation

For practical usage examples please refer to the *Getting started* guide at `https://github.com/sylefeb/Silice/blob/master/GetStarted.md`.

8.1 VGA emulation

Silice comes with a tool called `silicehe` for *Silice hardware emulation*. This tool will read the output of the `icarus_vga` simulator (the fst file storing signals) and produce a sequence of images from the stored VGA signals.

Simply call it with the fst file as the second argument.