1 LSQ Rewrite Report

1.1 Introduction

This report summarises and explains the changes made to the Python script to generate the LSQ, allowing it to generate the LSQ in both Verilog and VHDL. Instead of directly adding strings to the output file, the script now uses a simple intermediate representation to create statements, which are then handled differently by two different code emitters. This allows for generation of both Verilog and VHDL using the same code. This single unified script allows for easier code modifications in the future.

1.2 Existing structure

The old version has the following structure: The files in `vhdl_gen/generators` are used to generate components in the form of strings, such as the group allocator. Each of these generators is then used in `vhdl_gen/lsq.py` which generates the overall LSQ.

Each component is generated as follows: There is an architecture string (`arch`) keeping track of all the statements, which is initially empty. In addition, there is a `VHDLContext` instance, used to keep track of data such as the current indentation, initialisation and port strings.

Statement strings are then added to the `arch` string. These statements can be typed out manually (e.g. `arch += 'a and b'`), but more often statements are generated using helper functions.

In order to handle signal initialisation, there is a group of `Signals` classes. These classes handle the naming, initialisation and updating of signals. A signal can be initialised as follows:

```
ldq_tail_i = LogicVec(ctx, 'ldq_tail', 'i', self.configs.ldqAddrW)
```

This adds the required initialisation string for a signal called `ldq_tail` to the context. In this case the string is added as an input port as it is an input variable as denoted by the `'i'`. If the variable is specified as either a wire or register it is added to the instantiation string of the `Context` instead. The resulting `Signal` instance can be used to generate statement code by getting the read/write name using `ldq_tail_i.getNameRead()` and `ldq_tail_i.getNameWrite()`.

The most used helper function to generate statements is the `Op` function. This is the function used to generate an assignment (`a <= b`). The function can be used as follows:

```
arch += Op( ctx, empty_loads, self.configs.numLdqEntries,
      'when', ldq_empty_i,
      'else', '(', '\'0\'', '&', loads_sub, ')')
```

Which generates:

```
empty_loads <= "10000" when ldq_empty_i else ( '0' & loads_sub );
```

As can be seen, the `Op` function can accept either strings, numbers or variables, these are then formatted accordingly.

In addition to the `Op` helper function, there are also other helper functions such as `Reduce` or `WrapAdd` however, as most of these use `Op` to generate statements, it is not necessary to go into detail regarding them.

So overall there are two main operations: Signals are instantiated, and then used in `Op` assignments to assign them various values.

1.3 Modifications

Overall the approach is to adjust as little code as possible in the generator code, but to enable Verilog code generation in addition to VHDL. As the use of the `Signal` and `Op` classes already provides a layer of abstraction over directly printing strings, modifying this was sufficient to allow for Verilog generation.

This is done by moving the text-specific generation to two different emitters (VHDL and Verilog). The `Op` statement is replaced by `em.add_assignment(..)`, meaning that it depends on the emitter what string is used. The emitters also replace the `VHDLContext` class, keeping track of current indentation and other string states.

The arguments passed to `Op` need to be adjusted as they are language specific. Take for example the `when` and **`else`** in the previous example, which would be `:` and `?` in Verilog. So the way to pass the structure of the statements is done using a simple intermediate representation (found in `IR.py`)

There is one abstract class `Statement` which each of the subclasses inherit. The subclasses include `Val`, to pass a single value, `Bit` for single-bit values, `Bin` and `Un` to handle binary and unary statements, `WhenElse` to handle conditional statements and `CustomStatement` to allow direct passing of VHDL and Verilog strings. The `Signal` classes also inherit `Statement`.

The Statement class overrides various Python operators, and contains some helper functions to allow for clean initialisation. For instance the previous example is now generated as follows:

```
em.add_assignment(
  empty_loads, Val(self.configs.numLdqEntries)
  .when(ldq_empty_i)
  .else_(Bit(0).concat(loads_sub)),
)
```

Statements are then translated to strings using a visitor pattern, where each of the emitters has different implementations for `[statement]_to_str(..)` functions.

In the code there are two uses of the `CustomStatement` class, requiring to manually put both Verilog and VHDL code. These two sections contain process code, requiring code snippets such as: **`if`** `rising_edge(clock)then ...`, which cannot be represented in the current IR nor handled by the emitters. Ideally, the existing framework should be expanded to include **`if`** statements.

However, the decision was made to use customStatements as there are only two small sections where this is required. And it is tricky to take the different ways that VHDL and Verilog handle process triggers into account. Maybe if in future modifications such a pattern would be used more it would be a worthy tradeoff.

The Signal classes are also modified to use `em.[signal` **`type`**`]_signal_init` and `em.[signal_type]_reg_init` such that the signal initialisation and update string generation is handled by the emitters.

Behaviourally, exactly the same code is generated compared to the original VHDL code, with the exception of one modification. Not all registers in the original generated code are initialised. However, this created issues with the Verilog code as uninitialised registers are propagated differently. To solve this, all registers are initialised to 0 by default. As the value of uninitialised registers should not matter this should not change anything in the behaviour, unless there is already an existing bug in the original code. As the integration tests still pass with a similar amount of clock cycles, it is assumed that this does not change the behaviour of the LSQ.

1.3.1 Other refactors

Some general refactors are made to remove duplicate code. The biggest is the removal of the `VHDLSignal` classes. These classes contain almost the same code as the `Signal` classes with one minor difference: The naming is handled differently to be compatible with Dynamatic naming conventions. The old `VHDLSignal`s were used only in the generation of the wrapper.

Instead of having two different signal classes, the `Signal` classes now have a flag `dyn_comp` to enable the different naming convention. The wrapper generation now uses the new signal classes with the flag enabled. The old signal classes are removed.

1.4 Testing

The resulting program was tested using two different methods: The generated VHDL text is compared to the original generated VHDL file. As the structure of the generated code should be the same as the original code, running a `diff` command on both files should result in the same file.

This is almost the case, with the exception of brackets. As the modified script automatically generates brackets, it gets rid of some unnecessary brackets in the original script. In addition, it also generates some unnecessary brackets in case of equal precedence (e.g. `(a` **`and`** `b)`**`and`** `c`). To account for this, when comparing, brackets were ignored using the command

```
diff -w <(sed 's/[()]/ /g' original.vhd) <(sed 's/[()]/ /g' new.vhd)
```

Resulting files were manually checked to see if they were equal.

A better method of testing would be to see if the two files are logically or even structurally equal, however no good open source tools to do this were found. By testing the VHDL code, this also ensures that the Verilog at least has similar structure, so assuming both emitters are implemented correctly it also follows that the Verilog code is correct.

The second method was simply using the integration tests. Dynamatic has 112 integration tests. These tests run the whole Dynamatic workflow, compiling C++ code pieces to VHDL and Verilog, simulating the resulting hardware, and checking if the output is correct. Using these tests, both the generated VHDL and Verilog are tested. The final script has all tests pass, except two, which also don't pass in the original code.

It was manually checked whether there were differences in clock cycles between the original and modified code. Differences of maximum one clock cycle were found, which can be a natural difference between different simulation instances. From this we can assume that the LSQ exhibits the same behaviour between both the original, and between the VHDL and Verilog instances.

1.5 Flaws and possible future modifications

As mentioned before, the requirement for custom code snippets is not ideal.

Overall, the intermediate representation can be a bit janky, as it is possible to create an IR statement which generates incorrect code without throwing errors, take for example:

```
em.add_assignment(
  double_out, double_in & ~(double_in - (Val(0).concat(base)))
)
```

Which generates incorrect code as the width of `Val(0)` is automatically set to be equal to that of the output value (`double_out` in this case), while it needs to be the width of `double_in - base`. This behaviour is not very transparent.

This can be fixed by manually specifying the size:

```
em.add_assignment(
  double_out, double_in & ~(double_in - (Val(0, din.size).concat(base)))
)
```

Overall the issue is that by abstracting away from the exact code, it can be unpredictable what the exact behaviour is with regards to variable width or other language-specific quirks. This might be an issue when writing new code using the IR and the generated code would need to be inspected.

1.6 Conclusion

Overall, a Python script generating both VHDL and Verilog is successfully made and tested. The resulting branch can be found here. The code has significant refactors, as all the `Op` calls had to be modified to use the IR.