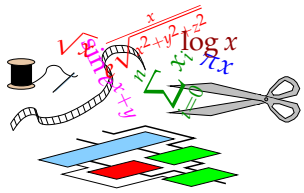


FloPoCo 2.5.0 developer manual

Florent de Dinechin, Bogdan Pasca

April 4, 2014

Welcome to new developers!

The purpose of this document is to help you use FloPoCo in your own project (section 2), and to show you how to design your own pipelined operator using the FloPoCo framework (sections 4 and 5).

1 Getting started with FloPoCo

1.1 Getting the source and compiling using CMake

It is strongly advised that you work with the svn version of the source, which can be obtained by following the instructions on `https://gforge.inria.fr/scm/?group_id=1030`. If you wish to distribute your work with FloPoCo, contact us.

If you are unfamiliar with the CMake system, there is little to learn, really. When adding .hpp and .cpp files to the project, you will need to edit `CMakeLists.txt`. It is probably going to be straightforward, just do some imitation of what is already there. Anyway `cmake` is well documented. The web page of the CMake project is `http://www.cmake.org/`.

1.2 Overview of FloPoCo

In FloPoCo, everything is an `Operator`. Operator is a virtual class, all FloPoCo operators inherit this class. A good way to design a new operator is to imitate a simple one. We suggest `Shifter` for simple integer operators, and `FPAdderSinglePath` for a complex operator with several sub-components. An example of assembling several FP operators in a larger pipeline is `Collision`.

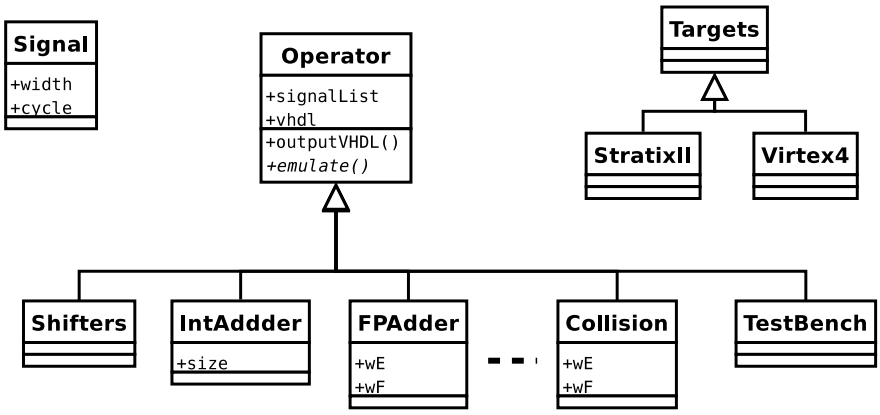
Meanwhile, browse through `Operator.hpp`. It has become quite bloated, showing the history of the project. Try not to use methods flagged as deprecated, as they will be removed in the future. Instead, use the automatic pipeline framework is described in Section 4 below.

Another important class hierarchy in FloPoCo is `Target`, which defines the architecture of the target FPGA. It currently has several sub-classes including, `VirtexIV`, `5`, `6` and `StratixII`, `IV`. You may want to add a new target, the best way to do so is by imitation. Please consider contributing it to the project.

To understand the command line, go read `main.cpp`. It is not the part we are the most proud of, but it does the job.

The rest is arithmetic!

And do not hesitate to contact us: `Florent.de.Dinechin` or `Bogdan.Pasca`, at `ens-lyon.fr`



2 Linking against FloPoCo

All the operators provided by the FloPoCo command line are availaible programmatically in libFloPoCo. A minimal example of using this library is provided in `src/main_minimal.cpp`.

The file `src/main.cpp` is the source of the FloPoCo command line, and as such uses most operators: looking at it is the quickest way to look for the interface of a given operator.

The other way is, of course, to look at the corresponding `hpp` file – they are all included by `src/Operator.hpp`. Some operators offer more constructors (richer interface options) than what is used in `src/main.cpp`.

There should be a Doxygen documentation of FloPoCo.

3 Fixed-point numbers in FloPoCo

In FloPoCo, a fixed point format is defined by a boolean true if signed, and two integers: the weights of the MSB and the LSB, which can be positive or negative. For instance the unit bit has weight 0, the point is between weights 0 and -1.

These two weights are inclusive: The size of the corresponding bit vector will be MSB-LSB+1. This is true for signed as well as unsigned numbers: If the format is signed, then the sign bit is the bit of weight MSB.

Now for a more stylistic, but nevertheless useful convention. Whenever an interface (be it to the command line, or to an internal function) includes the MSB and the LSB of the same format, they should appear in this order (MSB then LSB). This order corresponds to the order of the weights in the binary writing (the MSB is to the left of the LSB).

Examples:

- C char type corresponds to MSB=7, LSB=0.
- a n-bit unsigned number between 0 and 1 has MSB=-1 and LSB=-n
- a n-bit signed number between -1 and 1 has MSB=0 and LSB=-n+1

Finally, whenever we can live with integers, we should stick with integers and not obfuscate them as fixed-point numbers.

4 Pipelining made easy: a tutorial

If you want to experiment with a dummy operator and try the notions used in this section, consider reading and modifying the file `UserDefinedOperator.hpp` and `.cpp`. They contain an operator class `UserDefinedOperator` that you may freely modify without disturbing the rest of FloPoCo.

Let us consider a toy MAC unit, which in VHDL would be written

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library work;

entity MAC is
   port ( X   : in  std_logic_vector(63 downto 0);
          Y,Z : in  std_logic_vector(31 downto 0);
          R : out  std_logic_vector(63 downto 0)   );
end entity;

architecture arch of MAC is
  signal T: std_logic_vector(63 downto 0);
begin
   T <= Y * Z;
   R <= X + T;
end architecture;
```

We chose for simplicity a fixed-size operator, but all the following works as well for parameterized operators.

We have above the description of a combinatorial circuit. We now show how to turn it into a pipelined one.

4.1 First steps in FloPoCo operator writing

FloPoCo mostly requires you to copy the part of the VHDL that is between the `begin` and the `end` of the architecture into the constructor of a class that inherits from `Operator`. The following is minimal FloPoCo code for `MAC.cpp`:

```
#include "Operator.hpp"

class MAC : public Operator
{
public:
// The constructor
MAC(Target* target): Operator(target)
{
setName("MAC");
setCopyrightString("ACME MAC Co, 2009");

// Set up the IO signals
addInput ("X"  , 64);
addInput ("Y"  , 32);
addInput ("Z"  , 32);
addOutput("R"  , 64);
```

```
    vhdl << declare("T", 64) << " <= Y * Z;" << endl;
    vhdl << "R <= X + T;" << endl;
}

// the destructor
~MAC() {}
```

And that's it. `MAC` inherits from `Operator` the method `outputVHDL()` that will assemble the information defined in the constructor into synthesizable VHDL. Note that `R` is declared by `addOutput`.

So far we have gained little, except that is is more convenient to have the declaration of `T` where its value is defined. Let us now turn this design into a pipelined one.

4.2 Basic pipeline

Let us first insert a synchronization barrier between the result of the multiplication and the adder input. The code becomes:

```
(...)
    vhdl << declare("T", 64) << " <= Y * Z;" << endl;
    nextCycle();
    vhdl << "R <= X + T ;" << endl;
(...)
```

With the command-line option `-pipeline=yes`, this code will insert a synchronisation barrier before the adder, delaying `X` so that the operator is properly synchronized. It will produce a combinatorial operator (the same as previously) with `-pipeline=no`.

How does it work?

- `Operator` has a `currentCycle` attribute, initially equal to zero. The main function of `nextCycle()` is to increment `currentCycle`.
- Every signal declared through `addInput` or `declare` has a `cycle` attribute, which represents the cycle at which this signal is active. It is 0 for the inputs, and for signals declared through `declare()` it is `currentCycle` at the time `declare` was invoked.
- Every signal also possesses an attribute `lifeSpan` which indicates how many cycles it will need to be delayed. This attribute is initialized to 0, then possibly increased by each time the signal is used. When the `lifeSpan` of a signal `X` is greater than zero, `outputVHDL()` will create `lifeSpan` new signals `X_d1`, `X_d2` and so on, and insert registers between them. In other words, `X_d2` will hold the value of `X` delayed by 2 cycles.
- FloPoCo scans the VHDL and looks for right-hand side occurrences of declared signals. For instance, in the line after the `nextCycle`, it finds `X` and `T`. For such signals, it does the following. First, it compares `currentCycle` and the `cycle` declared for `X`, which we note `X.cycle`.
  - If they are equal, or if `-pipeline=no`, `X` is written to the VHDL untouched.
  - If `currentCycle` $<$ `X.cycle`, FloPoCo emits an error message complaining that `X` is being used before the cycle at which it is defined.
  - If `currentCycle` $>$ `X.cycle`, FloPoCo delays signal `X` by n=`currentCycle`$-$`X.cycle` cycles. Technically, it just replaces, in the output VHDL, `X` with `X_dn`. It also updates `X.lifeSpan` to be at least equal to n.
- All the needed signals will be declared in the output VHDL based on the `lifeSpan` information.

This whole scheme is actually run in two passes so that `currentCycle` may be moved forth and back in the FloPoCo code, which is useful in some situations.

This scheme gracefully degrades to a combinatorial operator. It also automatically adapts to random insertions and suppressions of synchronization barriers. Typically, one synthesizes an operator, and decides to break the critical path by inserting a synchronisation barrier in it. This may be as simple as inserting a single `nextCycle()` in the code. FloPoCo takes care of the rest.

It is also possible to have `if`s before some of the `nextCycle()`, so that the pipeline adapts to the frequency, the operator generic parameters, etc. See `IntAdder` for an example. However, starting with version 2.1.0 a finer-grain procedure for pipelining operators is introduced and will be explained section 4.4.

Some more notes:

- The second parameter of `declare()`, the signal width, is optional and defaults to 1 (a `std_logic` signal).
- Other functions allow to manipulate `currentCycle`
  - `setCycle(int n)` sets `currentCycle` to n.
  - `setCycleFromSignal(string s)` sets the `currentCycle` to the `cycle` of the signal whose name is given as an argument (going back if needed),
  - `syncCycleFromSignal(string s)` is similar to the previous but may only advance `currentCycle`. It allows to synchronise several signals by setting `currentCycle` to the max of their `cycle`.

    See `FPAdderSinglePath` or `FPLog` for examples of such synchronisations.

  All these functions have an optional boolean second argument which, if true, inserts in the generated VHDL a comment "– entering cycle n".
- If our toy example, is part of a larger circuit such that X is itself delayed, the pipeline will adapt to that.

4.3 Pipeline with sub-components

We now show how to replace the + and * with FloPoCo pipelined operators. These operators support frequency-directed pipelining, which means that the resulting MAC will also have its pipeline depth automatically computed from the user-supplied frequency (the `-frequency` option of the command-line).

```
(...)
// vhdl << declare("T", 64) << " <= Y * Z;" << endl;
IntMultiplier *my_mult = new IntMultiplier(target, 32, 32);
oplist.push_back(my_mult); // some day this will be an addOperator method
inPortMap   (my_mult, "X", "Y"); // formal, actual
inPortMap   (my_mult, "Y", "Z");
outPortMap  (my_mult, "R","T");
vhdl << instance(my_mult, "my_mult"); // 2nd param is the VHDL instance name

// advance to the cycle of the result
syncCycleFromSignal("T");
// pipelined operators do not have a register on the output
nextCycle();

// vhdl << "R <= X + T;" << endl;
IntAdder *my_adder = new IntAdder(target, 64);
```

```
oplist.push_back(my_adder);
inPortMap   (my_adder, "X", "X");
inPortMap   (my_adder, "Y", "T");
inPortMapCst(my_adder, "Cin", "0"); -- carry in
outPortMap  (my_adder, "R","RR");
vhdl << instance(my_adder, "my_add");

// advance to the cycle of the result
syncCycleFromSignal("RR");
   vhdl << "R <= RR;" << endl;
(...)
```

And that's it. In the code above, an `inPortMap()` does the same job as an occurrence of signal on the right-hand side, and an `outPortMap()` does the same job as a `declare()`, although it doesn't need a signal width since it can read it from the sub-component. `instance()` also has the side effect that `outputVHDL()` will declare this component in the VHDL header of `MAC`.

4.4 Frequency-directed pipelining

The command line of FloPoCo supports specifying the desired frequency of the generated operators by using the `-frequency` option. The philosophy behind is to generate the smallest (in terms of resource usage) and the shorter latency operator given this frequency specification (`-frequency` command-line option) for a given target (`-target` command-line option).

A fixed pipeline can be easily obtained using the `nextCycle()` function previously introduced. As suggested before, frequency directed pipelining can be accomplished by conditional statements around the `nextCycle()`. The following methodology does exactly that, but in a generic and (hopefully) future-proof way.

Let's go back to our basic MAC example (the one without components). Looking at the critical path it is clear that it goes through a multiplication and then an addition. Before emitting the code of an operation that will increase the critical path (in our example, the multiplication) we want to evaluate in advance what the critical path becomes if we add to it the delay of this operation. There are two cases:

- the new delay is greater than $1/f$. This means that it will be impossible to perform these two operations in the same cycle while ensuring proper operation at frequency f. In this case a `nextCycle()` must be called to insert a synchronisation barrier. This resets the critical path delay, which becomes the delay of the operation after the barrier (the multiplication in this example).
- this new critical path delay is smaller than the target period ($1/f$). In this case we just have to perform some bookkeeping: the critical path delay is incremented with the operation delay.

The FloPoCo command that does it all is `manageCriticalPath(double delay)`. It must be placed before each block of code that generates some hardware on the critical path – there is some designer expertise here.

The augmented FloPoCo code would look something like:

```
setCriticalPath(0.0);
manageCriticalPath( target->DSPMultiplierDelay() );
vhdl << declare("T", 64) << " <= Y * Z;" << endl;
manageCriticalPath( target->adderDelay(64) );
vhdl << "R <= X + T ;" << endl;
```

Note that the delay passed to `manageCriticalPath()` should be evaluated, whenever possible, using methods of the `Target` class. This ensure that this pipelining work is done once for all the possible targets.

4.5 Sub-cycle pipelining (optional)

A working pipeline using sub-components is typically obtained by placing synchronization barriers on the inputs and outputs. However, it is often an overkill: most of the times, the previous approach leaves the output cycle not fully consumed. Also, sometimes, one wants to perform only a very simple, low-delay operation on the inputs.

For such cases, operators can optionally :

- receive a list of delays on the inputs (("X",1.5e-9),("Y",1.2e-9)) representing the combinatorial delays already present on these signals.
- report the combinatorial delay on the output signals ("R", 2.0e-9).

Using this information, the sub-component constructor can properly adjust the pipeline for the given frequency. Here is full example. The input delays are given in the variable named `inputDelays`.

```
setCriticalPath( getMaxInputDelays(inputDelays)  );
manageCriticalPath( target->DSPMultiplierDelay() );
vhdl << declare("T", 64) << " <= Y * Z;" << endl;
manageCriticalPath( target->adderDelay(64) );
vhdl << "R <= X + T ;" << endl;
outDelayMap["R"] = getCriticalPath(); //returns the current delay on the critical path
```

In the case of the second, component-based design, the code becomes:

```
setCriticalPath( getMaxInputDelays(inputDelays)  );
// vhdl << declare("T", 64) << " <= Y * Z;" << endl;
IntMultiplier *my_mult = new IntMultiplier(target, 32, 32, inDelayMap("X",getCriticalPath()
oplist.push_back(my_mult); // some day this will be an addOperator method
inPortMap   (my_mult, "X", "Y"); // formal, actual
inPortMap   (my_mult, "Y", "Z");
outPortMap  (my_mult, "R","T");
vhdl << instance(my_mult, "my_mult"); // 2nd param is the VHDL instance name

// advance to the cycle of the result
syncCycleFromSignal("T");
setCriticalPath( my_mult->getOutputDelay("R") );

// vhdl << "R <= X + T;" << endl;
IntAdder *my_adder = new IntAdder(target, 64, inDelayMap("X",getCriticalPath()));
oplist.push_back(my_adder);
inPortMap   (my_adder, "X", "X");
inPortMap   (my_adder, "Y", "T");
inPortMapCst(my_adder, "Cin", "0"); -- carry in
outPortMap  (my_adder, "R","RR");
vhdl << instance(my_adder, "my_add");

// advance to the cycle of the result
syncCycleFromSignal("RR");
setCriticalPath( my_adder->getOutputDelay("R") );

   vhdl << "R <= RR;" << endl;
   outDelayMap["R"] = getCriticalPath();
```

For more information, check FPExp for example, and don't hesitate to contact us.

5 Test bench generation

5.1 Overview

`Operator` provides one more virtual method, `emulate`, to be overloaded by each Operator. As the name indicates, this method provides a bit-accurate simulation of the operator.

Once this method is available, the command
`flopoco FPAdder 8 23 TestBenchFile 500`
produces a test bench of 500 test vectors to exercise `FPAdder`.

This test bench is properly synchronized in case of a combinatorial operator. This is managed by the `TestBenchFile` operator: `emulate()` only has to specify the mathematical (combinatorial) functionality of the operator.

The `emulate()` method should be considered the specification of the behaviour of the operator. Therefore, as any instructor will tell you, it should be written *before* the code generating the VHDL of the operator!

5.1.1 TestBench and TestBenchFile

The `TestBenchFile` operator stores the generated test vectors in the text file `test.input`. There is another test-bench operator, `TestBench`, which stores the test vectors directly in the VHDL. In early debugging this may be more helpful, as this VHDL may include comments (see below). However compiling the VHDL test bench will take much more time than the simulation itself, so `TestBench` doesn't scale beyond a few thousand vectors. Compiling a `TestBenchFile` is in constant time.

5.2 `emulate()` internals

The simplest example of `emulate()` is that of `src/IntAdder.cpp`, copied below.

```
void IntAdder::emulate ( TestCase* tc ) {
// get the inputs from the TestCase
mpz_class svX = tc->getInputValue ( "X" );
mpz_class svY = tc->getInputValue ( "Y" );
mpz_class svC = tc->getInputValue ( "Cin" );

// compute the multiple-precision output
mpz_class svR = svX + svY + svC;
// Don't allow overflow: the output is modulo 2^wIn
svR = svR & ((mpz_class(1)<<wIn_)-1);

// complete the TestCase with this expected output
tc->addExpectedOutput ( "R", svR );
}
```

`emulate()` has a single argument which is a `TestCase`. This is a data-structure associating inputs to outputs. Upon entering `emulate()`, the input part is filled (probably by `TestBench` or `TestBenchFile`), and the purpose of `emulate()` is to fill the output part.

An input/output is a map of the name (which should match those defined by `addInput` etc.) and a `mpz_class`. This class is a very convenient C++ wrapper over the GMP multiple-precision library, which can almost be used as an `int`, without any overflow issue. When the input/outputs are integers, this is a perfect match.

When the input/outputs are floating-point numbers, the most convenient multiple-precision library is MPFR. However the I/Os are nevertheless encoded as `mpz_class`. The `emulate()` method therefore typically must

- convert the `mpz_class` inputs to arbitrary precision floating-point numbers in the MPFR format – this is done with the help of the `FPNumber` class;
- compute the expected results, using functions from the MPFR library;
- convert the resulting MPFR number into its bit vector, encoded in an `mpz_class`, before completing the `TestCase`.

This double conversion is a bit cumbersome, but may be copy-pasted from one existing operator: Imitate `FPAdderSinglePath` or `FPExp`.

5.2.1 Fully and weakly specified operators

Most operators should be fully specified: for a given input vector, they must output a uniquely defined vector. This is the case of `IntAdder` above. For floating-point operators, this unique output is the combination of a mathematical function and a well-defined rounding mode. The bit-exact MPFR library is used in this case. Imitate `FPAdderSinglePath` in this case.

Other operators are not defined so strictly, and may have several acceptable output values. The last parameter of `addOutput` defines how many values this output may take. An acceptable requirement in floating-point is *faithful rounding*: the operator should return one of the two FP values surrounding the exact result. These values may be obtained thanks to the *rounding up* and *rounding down* modes supported by MPFR. See `FPExp` or `FPLog` for a simple example, and `Collision` for a more complex example (computing the two faithful values for $x^2 + y^2 + z^2$).

5.3 Operator-specific test vector generation

Overloading `emulate()` is enough for FloPoCo to be able to create a generic test bench using random inputs. The default random generator is uniform over the input bit vectors. It is often possible to perform better, more operator-specific test-case generation. Let us just take two examples.

- A double-precision exponential returns $+\infty$ for all inputs larger than 710 and returns 0 for all inputs smaller than -746. In other terms, the most interesting test domain for this function is when the input exponent is between -10 and 10, a fraction of the full double-precision exponent domain (-1024 to 1023). Generating uniform random 64-bit integers and using them as floating-point inputs would mean testing mostly the overflow/underflow logic, which is a tiny part of the operator.
- In a floating-point adder, if the difference between the exponents of the two operands is large, the adder will simply return the biggest of the two, and again this is the most probable situation when taking two random operands. Here it is better to generate random cases where the two operands have close exponents. Besides, a big part of the adder architecture is dedicated to the case when both exponents differ only by 1, and random tests should be focused on this situation.

Such cases are managed by overloading the Operator method `buildRandomTestCases()`.

5.4 Corner-cases and regression tests

Finally, `buildStandardTestCases()` allows to test corner cases which random testing has little chance to find. See `FPAdderSinglePath.cpp` for examples.

Here, it is often useful to add a comment to a test case using `addComment`: these comments will show up in the VHDL generated by `TestBench`.