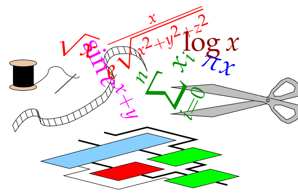


FloPoCo 5.0.git developer manual

Florent de Dinechin+

September 1, 2022

Welcome to new developers!

The purpose of this document is to help you use FloPoCo in your own project (section 1 to 2), and to show you how to design your own pipelined operator using the FloPoCo framework (sections 3 and 5).

Contents

1 Getting started with FloPoCo **2**
- 1.1 Getting the source and compiling using CMake . . . 2
- 1.2 Linking against FloPoCo . . . 2
- 1.3 Overview of FloPoCo code organization . . . 2
- 1.4 Adding a new operator to FloPoCo . . . 3

2 Data-types in FloPoCo **3**
- 2.1 Floating-point numbers . . . 3
- 2.2 Fixed-point numbers . . . 3

3 Tutorial for new developers **4**
- 3.1 First steps in FloPoCo operator writing . . . 4
- 3.2 Adding delay information . . . 5
- 3.3 Sub-components: unique instances . . . 5
- 3.4 Sub-components: shared instances . . . 5
- 3.5 Using the Table object . . . 6

4 Frequency-directed pipeline **6**
- 4.1 VHDL generation for a simple component . . . 6
  - 4.1.1 First VHDL parsing and signal graph construction . . . 6
  - 4.1.2 Scheduling of the signal graph . . . 6
  - 4.1.3 Back-annotation of the VHDL stream with delay information . . . 8
  - 4.1.4 Final VHDL output . . . 8
- 4.2 Subcomponents and instance . . . 8
  - 4.2.1 Unique instances . . . 8
  - 4.2.2 Shared instances . . . 9

5 Test bench generation **9**
5.1 Overview 9
5.2 emulate() internals 11
5.3 Fully and weakly specified operators 11
5.4 Operator-specific test vector generation 11
5.5 Corner-cases and regression tests 12

6 Regression testing, build test **12**

7 Bit heaps **12**
7.1 The data structure 13
7.2 Compressor tree generation 13

8 Writing a new target **14**

1 Getting started with FloPoCo

1.1 Getting the source and compiling using CMake

It is strongly advised that you work with the git version of the source, which can be obtained by following the instructions on `http://flopoco.org/flopoco_installation.html`. 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 `src/SourceFileList.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 Linking against FloPoCo

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

It is best to use an operator through its factory. 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.

1.3 Overview of FloPoCo code organization

The core of FloPoCo is the `Operator` class. Operator is a virtual class from which all FloPoCo operators inherit.

The FloPoCo source includes a dummy operator, `TutorialOperator`, for you to play with. Feel free to experiment within this one. By default it is compiled but unplugged. To plug it back, just comment the corresponding line in main.cpp.

A good way to design a new operator is to imitate a simple one. We suggest `Shifter` for simple integer operators, and `FPAddSinglePath` for a complex operator with several sub-components.

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. Hopefully.

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

The command-line parser is in `UserInterface` but in principle you won't have to edit it. It takes information from each operator's documentation strings defined in their `registerFactory()` methods. Again, we hope that you can design the interface to your operator by imitation of existing ones.

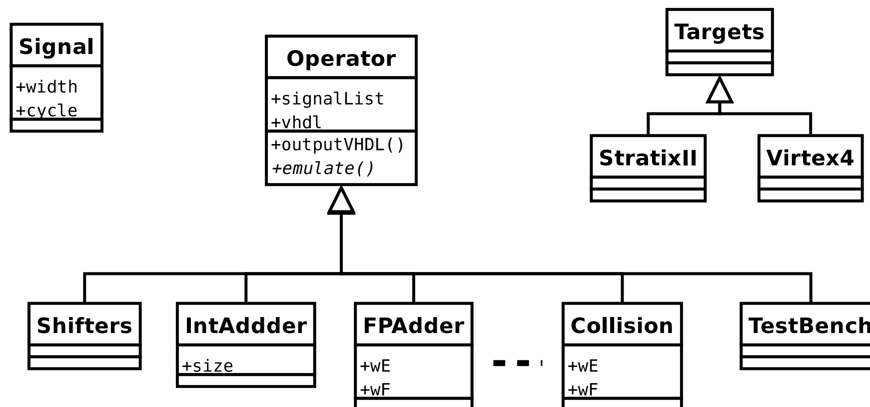
1.4 Adding a new operator to FloPoCo

To add a new operator to FloPoCo, you need to

- write its `.cpp` and `.hpp` (we suggest you start with a copy of `TutorialOperator`, which is an almost empty skeleton);
- add it to `src/SourceFileList.txt` so that it gets compiled;
- add it to `src/Factories/Interfaced.txt` if you want it to appear in the command-line interface.

That should be all. The rest is arithmetic!

And do not hesitate to contact us: `Florent.de.Dinechin@insa-lyon.fr`.



2 Data-types in FloPoCo

2.1 Floating-point numbers

FloPoCo partly supports two floating-point formats: the standard IEEE-754 (generalized to arbitrary exponent and mantissa sizes), and its internal format (see the web documentation). The internal format lacks subnormal support, but is more efficient and therefore should be preferred for application-specific development.

2.2 Fixed-point numbers

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). When a boolean sign is passed as well, it should be first, for the same reason (the sign bit is the leftmost bit).

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.

3 Tutorial for new developers

The FloPoCo distribution include a dummy tutorial operator in `src/TutorialOperator.hpp` and `src/TutorialOperator.cpp`. It describes an operator class `TutorialOperator` that you may freely modify without disturbing the rest of FloPoCo.

`TutorialOperator` is heavily documented, and this section assumes that you are looking at it.

After compiling FloPoCo, run in a terminal

```
./flopoco TutorialOperator
```

You will obtain the documentation on the parameters of this operator. This documentation is defined by the `TutorialOperator::registerFactory` method.

Now run in a terminal

```
./flopoco TutorialOperator param0=8 param1=8
```

and you should obtain some VHDL in `flopoco.vhdl`

3.1 First steps in FloPoCo operator writing

FloPoCo mostly requires you to embed 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.

3.2 Adding delay information

A latency may be passed as first optional argument to `declare()`. This value decribes the contribution of this VHDL statement to the critical path. It is best defined using methods of `Target`. See `FPAddSinglePath` for examples.

3.3 Sub-components: unique instances

In FloPoCo, most instances are *unique*: an `Operator` is built for a specific context, optimized for this context, and only one instance of this Operator will be used in the VHDL. This is the default situation, because it allows the tool to optimize the pipelining of each component for its context. The preferred method to use in such case is the *newInstance()* method of Operator. See `FPAddSinglePath` for an example of a large component that instantiates many sub-components (several `IntAdder`, `Shifter`, etc).

`newInstance()` uses the factory-based user interface. It is common to need a table of pre-computed values. To get a unique instance of such a table, first build the table content as a `vector<mpz_class>`, then call `Table::newUniqueInstance()`. See examples in `Trigs/FixSinCos.cpp`.

If for some reason you want to use a unique instance but don't want/need to expose a user interface for it, you just have to follow the same sequence of calls as you may find in `Table::newUniqueInstance()`.

Beware, the order is important for the operator scheduling to work properly, and it has changed since version 5.0.

3.4 Sub-components: shared instances

A component may also be shared (i.e. the same component is reused many times). Simple examples are the tables in `FPConstDiv`, or in `IntConstDiv`.

The preferred method to use in such case is the *newSharedInstance()* method of Operator, as in the following:

```
Operator* op = new MySubComponent(...);
op -> setShared();
//now some loop that creates many instances
for (....) {
  string myInstanceName = ...;
  string actualX = ...;
  string actualR = ...;
  vhdl << declare(actualX, ..) << " <= " << ...;
  newSharedInstance(op, myInstanceName, "X=>"+actualX, "R=>"+actualR);
}
```

See `IntConstDiv` or `FPDiv` for detailed examples.

3.5 Using the Table object

Small tables of precomputed values are very powerful components, especially when targetting FPGAs. They are quite often shared.

See `IntConstDiv` or `FPDiv` for examples of small, shared tables (intended to be implemented as LUTs on FPGAs, and as logic gates on ASIC).

See `FixFunctionByTable` for an example how to inherit `Table`.

See `FixFunctionByPiecewisePoly` for an example how to instanciate a `Table` as a sub-components.

TODO (not repaired yet): See `FPExp` for an example of unique Table intended to fit in a block RAM.

4 Frequency-directed pipeline

The pipeline framework is implemented mostly in the `Operator` and `Signal` classes, and we refer the reader to the source code for the full details. More details can also be found in [1].

4.1 VHDL generation for a simple component

4.1.1 First VHDL parsing and signal graph construction

The `vhdl` stream is parsed (as the constructor writes to it) to locate VHDL signal identifiers.

This pass builds a signal graph, an example of which is shown on Figure 1 (it was obtained in `flopoco.dot` by the command `./flopoco Shifter wIn=8 maxshift=8 dir=1`)

In this graph, the nodes are signals (of the FloPoCo `Signal` class), and the edges are signal dependencies, *i.e.* which signal is computed out of which signal. Technically, the graph is built by defining predecessors and successors of each `Signal`.

The operations between the signals are not kept in this graph: they are kept in the `vhdl` stream. However, their latency (passed as first optional argument to `declare()`) is used to label each signal. This value decribes the contribution of this VHDL statement to the critical path.

In Figure 1, the first line of each box is the signal name. The second line is the critical path contribution of each signal. The third line is the actual global timing of each signal, which is computed in the following.

The reader interested in this first parsing pass should have a look at `FlopocoStream.cpp`.

4.1.2 Scheduling of the signal graph

The second step of automatic pipelining is the scheduling of the signal graph. It is implemented in the method `Operator::schedule()`. It is an ASAP (as soon as possible) scheduling: starting from the input, we accumulate the critical path along the edges of the signal graph.

With the `pipeline=no` option, what we obtain in `flopoco.dot` is an estimate of the critical path from an input to each signal.

With `pipeline=yes`, the schedule constructs a pipeline. Each signal is assigned a cycle and a critical path within this cycle (i.e. what we obtain in `flopoco.dot` is an estimate of the critical path from the output of a register to each signal.

The timing of a signal is therefore expressed as a pair (c, τ), where

- c is an integer that counts the number of registers on the longest path from an input to s.
- τ is a real number that represents the critical path delay (in seconds) from the last register or earliest input to s.

The colors on Fig. 1, right, indicate the cycle. The complete lexicographic time of each signal is given by the third line of each signal box.

There is a lexicographic order on such timings: $(c_1, \tau_1) > (c_2, \tau_2)$ if $c_1 > c_2$ or if $c_1 = c_2$ and $\tau_1 > \tau_2$.

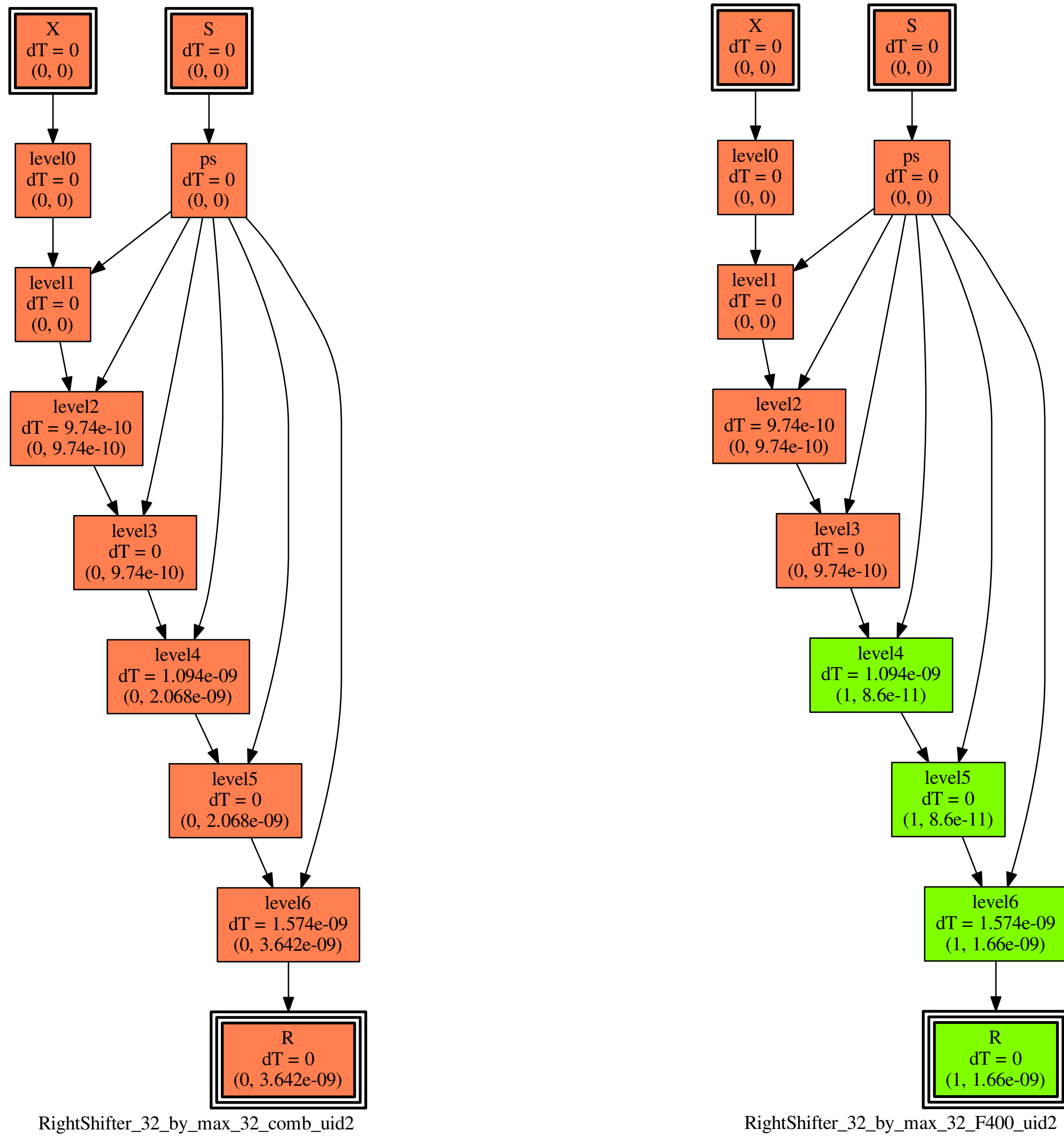


Figure 1: S-Graph for a combinatorial 8-bit barrel shifter, combinatorial (left) and pipelined (right)

4.1.3 Back-annotation of the VHDL stream with delay information

Once each signal is scheduled, there is a second parsing step of the VHDL stream that delays each signal where it is needed by the proper number of cycle. Technically, when parsing `A <= B and C;`, the schedule has ensured that `B.cycle`$\leq$ `A.cycle`. If `A.cycle` $>$ `B.cycle`, FloPoCo delays signal `B` by n=`A.cycle` $-$ `B.cycle` cycles.

Technically, it just replaces, in the output VHDL, `B` with `B_dn`. It also updates bookkeeping information that gives the life span of each signal.

This process is performed by the `Operator::applySchedule()` method.

4.1.4 Final VHDL output

The final step adds to the VHDL stream constructed from previous step all the declarations (entities, signals, etc) as well as the shift registers that delay signals. It is performed by the `Operator::outputVHDL()` method.

4.2 Subcomponents and instance

Now consider the more complex situation of a component that include other subcomponents. There are two distinct situations:

- either the subcomponent is used only once, in which case we want to schedule it in its context. This is the default situation. An extensive example of a complex component built by assembling simpler ones is `FPAddSub/FPAddSinglePath`.
- Or, the subcomponent is used many times (a typical example is the compressor in a bit heap), in which case all the instances will necessarily share the same schedule. In FloPoCo, we add a constraint in this case: such operators remain very small and thus shall not be pipelined. This covers 100% of the use cases so far. Such components have to be declared shared by calling `Operator::setShared()`.

In the following we detail these two cases and what happens under the hood in terms of scheduling.

4.2.1 Unique instances

In this case, the entity of the subcomponent is used in only one VHDL instance.

FloPoCo provides for this case a single method, `Operator::newInstance()`. Its inputs are those provided on the command-line interface, therefore this method will only work for operators which implement the factory methods. It returns a pointer to the newly created `Operator`.

In terms of VHDL, `Operator::newInstance()` creates both the entity of the subcomponent (by calling its constructor) and an instance of this entity in the `vhdl` stream of the current buffer.

Let us now see what happens in terms of scheduling and pipelining.

- In the signal graph, `Operator::newInstance()` connects the actual signals to the subcomponent ports, with simple wires (no delay added to the critical path). The `flopoco.dot` output shows a box around the signals of the subcomponent, but there is one single graph linking `Signal` objects.
- It is useful that the constructor of the subcomponent may take decisions based on the schedule of its inputs (example: the `IntAdder` pipelined integer splits its inputs depending on their critical path). Therefore, `Operator::newInstance()` calls `Operator::schedule()` (step 4.1.2 above).

  Since there is only one big signal graph, `Operator::schedule()` first gets to the root of the component hierarchy, before actually computing the schedule, starting from the inputs of this root.
- When the inputs to a sub-component are not synchronized, they will be synchronized inside the subcomponent.

- It is important to understand that `Operator::schedule()` can be invoked on an incomplete graph. In such an ASAP scheduling, the schedule of a signal is only defined by the schedule of its predecessors: once it is computed, it will no longer change, so `Operator::schedule()` may be called several times. It will be called by default after the end of the constructor of a root operator (so the signal graph is complete).
- All this probably works best (only works?) if the VHDL is written in the natural order, from inputs to outputs...

4.2.2 Shared instances

Again, shared instances are small, purely combinatorial components.

Here are the main differences:

1. The constructor of the subcomponent must be called only once.
2. The instances themselves must be somehow replicated in the signal graph.

The solution chosen is to replace *in the signal graph* instances with links between the inputs and outputs. Each output is labeled with a critical path contribution, equal to the critical path of this output in the instance.

This is performed under the hood by the `Operator::inPortMap()`, `Operator::outPortMap()` and `Operator::instance()` methods.

An instance is combinatorial, hence lives within a single cycle. Therefore, all the outputs of a shared instance have this same cycle. All the inputs are also input at this same cycle to the instance (they are delayed in the `port map`. If a pipeline register is inserted to account for the delay of a shared instance, it is inserted on the outputs.

The simplest example of shared instances is currently `FPDivSqrt/FPDiv`.

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 FPAdd we=8 wf=23 TestBench n=500`
produces a test bench of 500 test vectors to exercise `FPAdd`.

This test bench is properly synchronized if the operator under test happens to be pipelined: `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 (test-driven design).

To see examples of `emulate()` functions, see

- `IntAdder` or `IntConstDiv` for an operator with integer inputs and outputs; For these, the GNU Multiple Precision library is your friend.
- `FixRealKCM` for an operator with fixed-point inputs and outputs;
- `FPAdd` for an operator with floating-point inputs and outputs; For these, your friend is the GNU MPFR library, and FloPoCo provides all the needed helper functions to convert between bit vectors and MPFR numbers.

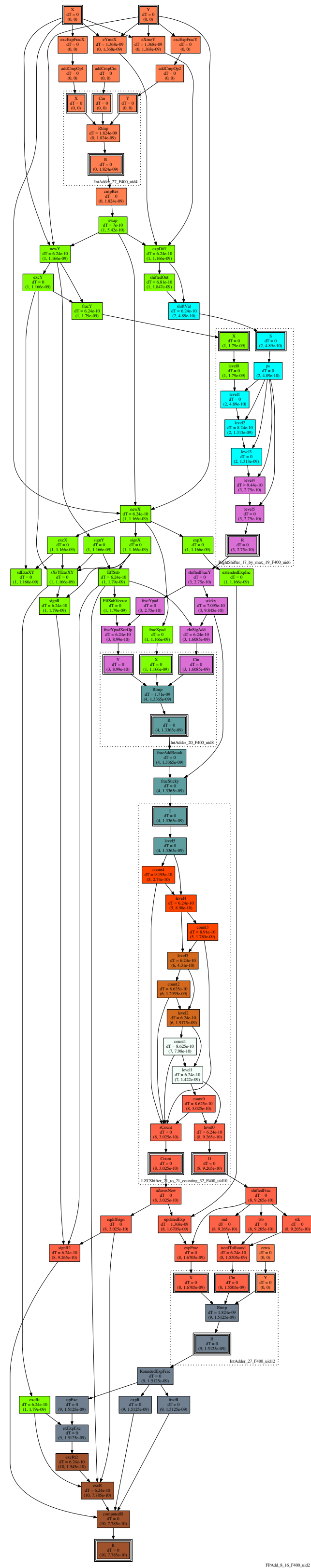


Figure 2: S-Graph for a pipelined FPAddSinglePath operator. Zoom on the Shifter component and observe that it has been pipelined for its context.

5.2 `emulate()` internals

`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`), and the purpose of `emulate()` is to fill the output part. `emulate()` is completely generic: Both inputs and outputs are specified as bit vectors. However these vectors are stored for convenience in `mpz_class` numbers. This class is a very convenient C++ wrapper around GMP, which can almost be used as an `int`, but without any overflow issue.

Therefore an input/output is a map of the name (which should match those defined by `addInput` etc.) and a `mpz_class`. 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 `FPAddSinglePath` or `FPExp`.

5.3 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 `FPAddSinglePath` 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.

5.4 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()`.

Listing 1: Typical code for an operator involving a bit heap

```
bh = new BitHeap(...);  // create an empty bit heap

// construction of the bit heap
for (...) {
  // generate VHDL that defines signals such as
  ///  mySingleBit or myBitVector
  ...

  // then add these signals to the bit heap:
  bh->addBit(myBit, pos1); // no VHDL generation here
  bh->addSignal(mybitVector, pos2); // nor here
}

// generate the compressor tree for the bit heap
bh->startCompression(); // this line generates a lot of VHDL
```

5.5 Corner-cases and regression tests

Finally, `buildStandardTestCases()` allows to test corner cases which random testing has little chance to find. See `FPAddSinglePath.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 file=false`.

6 Regression testing, build test

TODO, in between see FPAdd

7 Bit heaps

The main interface to the developer is the `BitHeap` class, which encapsulates all the needed data structures and methods. The bits to be added to a `BitHeap` are simply signals of the operator being generated. This makes it possible to define the initial bit heap, which can be of any shape or size. The `BitHeap` class also implements the compressor tree optimization techniques presented in [2, 3, 4, 5] (the compression algorithm is selected on the command line), and can generate the corresponding hardware.

The typical code to generate the architecture of an operator involving a bit heap is shown in Listing 1. The main methods to manipulate the bit heap are shown in Table 1. Basically, there is everything to add or subtract single bits, signed or unsigned numbers held in bit vectors, or constants. Once all the bits have been thrown on the bit heap, the `startCompression()` method launches the optimization of the compressor tree as well as the VHDL code generation.

Some operators that use the BitHeap have two constructors:

- one standalone, classical; The BitHeap is constructed in the operator and used only for this operator
- one virtual; a BitHeap from another operator is provided that is shared among the operators (which can build a more complex operator).

In the latter case, the typical arithmetic flow requires to first perform some error analysis to determine a number of guard bits to add to the bit heap. This must be done before any VHDL generation. If we want to delegate some of this error analysis to the subcomponents, then either it must be implemented in some static method of the Operator, or the constructor must delay the actual VHDL output to another method.

For an example of this in practice, see `FixFilters/FixSOPC`.

Table 1: The main methods of the `BitHeap` interface

Method	Description
`void addBit(string sigName,int pos)`	Add a single bit
`void addSignal(string sigName,int shift)`	Add a fixed-point signal, with sign extension if needed
`void subtractSignal(string name,int shift)`	Subtract a fixed-point signal, with sign extension if needed
`void addConstantOneBit(int pos)`	Add a constant one bit
`void addConstant(mpz_class cst,int pos)`	Add the constant $\texttt{cst} \cdot 2^{\text{pos}}$
`void startCompression()`	Generate a compressor tree

When the argument is a signal name, it is used to refer to a FloPoCo object of the class Signal, which encapsulate a fixed-point format with its signedness, its MSB, and its LSB.

Figure 3: Bit heap obtained for a 16-bit, 8-tap half-sine pulse shaping FIR filter operator `FixHalfSine`

7.1 The data structure

This section and the following describe the internals of the `BitHeap` class, and are intended for developers wishing to extend the BitHeap framework itself (e. g., with new compressors or compression algorithms).

A *weighted bit* is a data structure consisting essentially of a signal name, a bit position, and various fields used when building the compressor tree. In FloPoCo the `Signal` class also encapsulates the timing of each signal, so each weighted bit also carries its arrival time.

A column of the bit heap is represented as a list of weighted bits. This list is sorted by the arrival time of the bits, so that compression algorithms can compress first the bits arrived first.

The complete bit heap data structure essentially consists of its MSB and LSB, and an array of columns indexed by the positions.

The `BitHeap` class also offers methods to visualize a bit heap as a dot diagram like the one shown in Figure 3. Use the `generateFigures` flag to enable the output of `SVG` and `.tex` files (these use the macros found in `dot_diag_macros.tex`). Different arrival times may be indicated by different colors.

FloPoCo may generate such figures as SVG files that can be opened in a browser, in which case hovering the mouse over one bit shows its signal name and its arrival instant in circuit time.

7.2 Compressor tree generation

Once the data structure of the BitHeap is created by using the methods of Table 1, the compressor tree generation is started by calling `startCompression()`. It involves the following steps:

1. The algorithm for solving the compressor tree problem is selected (a derived class from `CompressionStrategy`).
2. A list of possible (target-dependent) compressors is generated (class `BasicCompressor` for representing the shape and class `Compressor` implementing the Operator performing the compression).
3. Then, the bits of the bit heap are scheduled to different stages according to their cycle and combinatorial arrival time.

4. Next, the compressor trees are optimized (by a class derived from `CompressionStrategy`) and a `BitHeapSolution` object is created. This solution contains the used compressors per stage and column.

5. Finally, the VHDL code of the compressor tree is generated.

New compressor tree optimization methods can be implemented by extending `CompressionStrategy`.

8 Writing a new target

Try to fill data such as LUT input size (`lutInputs()`), etc.

Here are some operators that can be used to calibrate delay functions. You may use the scripts in `tools/` to launch syntheses and get critical path reports. It is better to ask for a relatively low frequency (say, half peak, 200MHz) to avoid troubles with I/O buffer delays when using post place-and-route synthesis.

- To calibrate the logic delay, use

  ```
  ./flopoco IntConstDiv wIn=64 d=3 Wrapper
  ```

  If you have specified your lutInputs properly, the architecture should be a sequence of LUTs connected by local routing. First check that the estimated cost reported by FloPoCo matches the actual cost.

- To calibrate the IntAdder delay, use

  ```
  ./flopoco IntAdder wIn=64 Wrapper
  ```

  for increasing values of 64.

References

[1] M. Istoan and F. de Dinechin, “Automating the pipeline of arithmetic datapaths,” in *DATE 2017*, Lausanne, Switzerland, Mar. 2017. [Online]. Available: https://hal.inria.fr/hal-01373937

[2] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, “Arithmetic core generation using bit heaps,” in *Field-Programmable Logic and Applications*, Sep. 2013.

[3] M. Kumm and P. Zipf, “Efficient High Speed Compression Trees on Xilinx FPGAs,” in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014.

[4] ——, “Pipelined Compressor Tree Optimization Using Integer Linear Programming,” in *IEEE International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–8.

[5] M. Kumm and J. Kappauf, “Advanced Compressor Tree Synthesis for FPGAs,” *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.