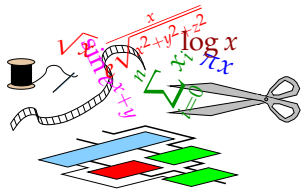


FloPoCo 5.0.git manual

Florent de Dinechin

January 25, 2023

Welcome to new developers!

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

Contents

1 FloPoCo from a user point of view

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 one of the project mainteners.

1.2 Overview

FloPoCo is an executable with a fairly simple command-line interface that may generate VHDL source code for most operators described in this book (and a few more).

An operator specification consists of an operator name followed by a list of parameter values.

Basic FloPoCo command-line example
The command
```
flopoco IEEEFPAdd wE=8 wF=23
```
produces a file called `flopoco.vhdl` containing synthesizable VHDL for an IEEE754 single precision floating-point adder.

Some operators come as a single VHDL entity, and in other cases the generated VHDL file contains many entities with a hierarchy of component instantiations. For instance the previous command outputs the following text on the console, showing 4 sub-components:

```
|---Entity RightShifterSticky26_by_max_25_comb_uid4
|      Not pipelined
|---Entity IntAdder_27_comb_uid6
|      Not pipelined
|---Entity LZC_26_comb_uid8
```

```
|      Not pipelined
|---Entity LeftShifter27_by_max_26_comb_uid10
|      Not pipelined
|---Entity IntAdder_31_comb_uid13
|      Not pipelined
Entity IEEEFPAdd_8_23_comb_uid2
   Not pipelined
```

The top-level entity, in such cases, is the last one.

It is also possible to have several operator specifications in the same command line.

An FPU for Nfloat format
The following command produces an adder, a multiplier, a divider and a square root operator for the Nfloat equivalent of single precision (all in the same VHDL file).

```
flopoco FPAdd wE=8 wF=23 FPMult we=8 wf=23 FPDiv we=8 wf=23 FPSqrt we=8 wf=23
```

Finally, it is possible to specify the value of global parameters such as the `frequency` in the example below (frequency-directed pipelining is discussed further in Section 1.8). Another important global parameter is `target` which specifies the target hardware.

An FPU pipelined for 200 MHz
The following command pipelines the previous FPU for 200 MHz.

```
flopoco frequency=200 FPAdd wE=8 wF=23 FPMult we=8 wf=23 FPDiv we=8 wf=23 FPSqrt we=8
   wf=23
```

The size of the generated VHDL code may vary from a few bytes to a few megabytes. It depends on the complexity of the operator, on the value of the parameters, but also on the very nature of the subcomponents: an addition may be described by a single character + in the VHDL implementation, whereas a table of precomputed values, provided in extension, may be arbitrarily large.

1.3 Data-types in FloPoCo

Let us detail those parameters that are related to data formats.

1.3.1 Fixed-point numbers

In FloPoCo, a fixed point format is defined by 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 (Figure 1): 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.

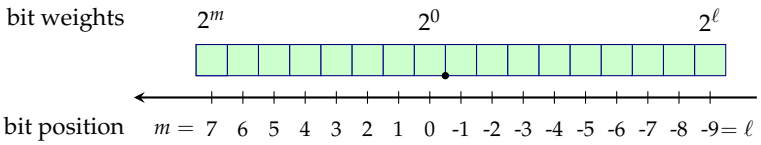


Figure 1: An unsigned fix-point format ufix(m, ℓ).

A fixed-point format may be signed or unsigned (Figure 1). For signed formats the sign bit is at the position m.

For some operators, the signedness is a parameter. In this case it is a boolean parameter (true for signed).

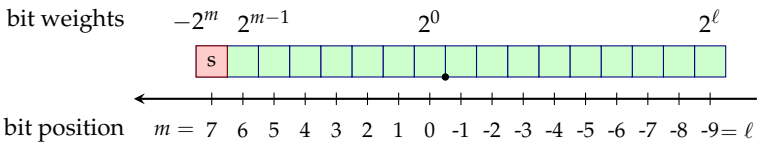


Figure 2: A signed fixed-point format sfix(m, ℓ).

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

Some examples of fixed-point formats:

- 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.

1.3.2 Floating-point numbers

FloPoCo partly supports two floating-point formats.

IEEE floating-point numbers The standard IEEE-754 is generalized to arbitrary exponent and mantissa sizes (see Figure 3)

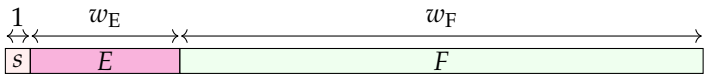


Figure 3: Bit fields of an IEEE754-like floating-point number of parameters (w_E, w_F)

The exponent field is used to encode

- infinities (exponent field $E = 11...11$, $F = 0$),
- Not a Number or NaN (exponent field $E = 11...11$, $F \neq 0$),
- subnormals (exponent field $E = 00...00$, $F \neq 0$)
- (signed) zeroes (exponent field $E = 00...00$, $F = 0$)

When not infinity and not NaN, the value of a floating-point vector may be defined as follows (where E_0 is the *exponent bias* and n is the "is normal" bit):

$$\begin{aligned} E_0 &= 2^{w_E-1} - 1 \\ n &= \begin{cases} 0 & \text{if } E = 0 \\ 1 & \text{otherwise} \end{cases} \\ X &= (-1)^s \times 2^{E-E_0+1-n} \times (n + F) \quad . \end{aligned} \tag{1}$$

Simple floating-point numbers FloPoCo also supports a simpler format where the exceptional cases (zero, infinity, NaN) are encoded in two additional bits (see Figure 4 and Table 1). This makes it more hardware-efficient (no decoding/encoding of these exceptional cases in the exponent), but less memory-efficient. Also, this format does not support subnormal numbers.

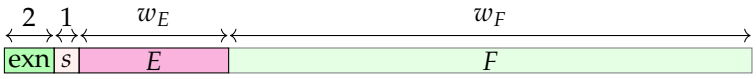


Figure 4: Bit fields of a flopoco floating-point number of parameters (w_E, w_F)

When exn= 01 (normal numbers), the value encoded is

$$X = (-1)^s \times (1 + F) \times 2^{E-E_0} \tag{2}$$

where E_0 is the same exponent bias as in IEEE-754 formats: $E_0 = 2^{w_E-1} - 1$.

Table 1: Encoding of exceptional cases in the FloPoCo floating-point format

exn	meaning
00	zero (there are two zeroes, noted $+0$ and -0, according to s)
01	normal numbers
10	infinity ($+\infty$ or $-\infty$, according to s)
11	NaN (Not a Number)

Brief comparison of the two formats Since the extremal values zero and infinity are not encoded as special exponent values, the exponent range for normal numbers is slightly larger than in the IEEE754 for the same value of w_E, as Table 2 shows. However a number in the FloPoCo format two more bits than a number in the IEEE754 format, for the same (w_E, w_F).

Table 2: Comparison of properties of the IEEE and FloPoCo formats.

	IEEEfloat$(w_\mathrm{E}, w_\mathrm{F})$	Nfloat$(w_\mathrm{E}, w_\mathrm{F})$
bias value	$E_0 = 2^{w_E-1} - 1$	
Total size	$w_\mathrm{E} + w_\mathrm{F} + 1$ bits	$w_\mathrm{E} + w_\mathrm{F} + 3$ bits
e_min	$-2^{w_E-1} + 2$	$-2^{w_E-1} + 1$
e_max	$2^{w_E-1} - 1$	2^{w_E-1}
Smallest	$2^{e_\mathrm{min}-w_\mathrm{F}} = 2^{-2^{w_E-1}+2-w_\mathrm{F}}$	$2^{e_\mathrm{min}} = 2^{-2^{w_E-1}+1}$
Largest	$(2 - 2^{-w_\mathrm{F}}) \cdot 2^{e_\mathrm{max}}$	$(2 - 2^{-w_\mathrm{F}}) \cdot 2^{e_\mathrm{max}}$

1.3.3 Format conversion utilities for debugging

FloPoCo includes small useful programs that convert the binary string of a floating-point number to human-readable decimal, and back:

- `fp2bin` and `bin2fp` for the FloPoCo format;
- `fp2ieee` and `ieee2fp` for the IEEE format.

`fp2bin`

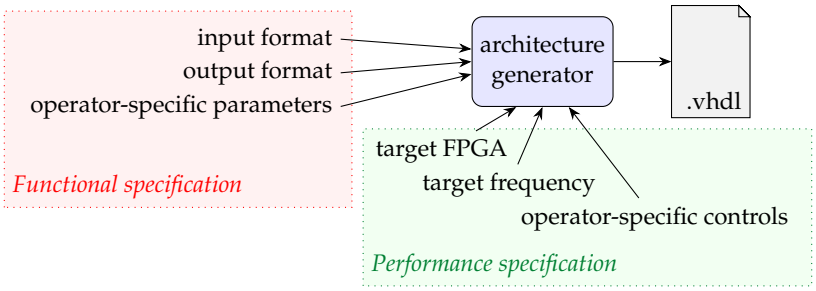


Figure 5: Interface to a flopoco Operator

1.4 More on parameters

All the parameters are provided to FloPoCo as *name=value* pairs.

Each operator is heavily parameterized with functional and performance parameters. To know the parameters of a given operator, just invoke flopoco for this operator without parameters, e. g., `flopoco FPAdd`.

As illustrated in Figure 5, parameters can be broadly separated in two classes: *functional* parameters specify the function, while *performance* parameters control the performance and its many trade-offs (such as memory versus compute, area versus speed, frequency versus latency, etc.).

A general rule is that functional parameters should be mandatory with no default value, while performance parameters should be optional with a sensible default value. However, there are exceptions to both rules. For instance, there is one parameter to `FPAdd` which decides if the operator should be an adder or a subtracter. This is a functional parameter, but its default is set to addition, because we feel it is the most common case. Conversely, `FixFunctionByPiecewisePoly` has a mandatory parameter for the degree to be used, although it is definitely a performance parameter. The reason here is that we feel that the decision of a "sensible default value" is too application-dependent.

For a given operator, there are usually many parameter combinations that do not make sense. We attempt to filter them and provide sensible error messages in such cases, but users often have more imagination that developers can anticipate, and invalid combinations of parameter values may lead to unexpected crashes or non-working VHDL.

1.5 Global options

Several global options are available. They are also specified as *name=value* pairs and will typically change the operators occuring after them in the list. Here is a non-exhaustive list, the full list is available on the command line when typing `flopoco` with no argument.

- `target=Virtex5` sets the target hardware family. For a list of supported families see the command line. We typically target the highest speed grade available for a family.
- `frequency=300` sets the target frequency (in MHz). FloPoCo will attempt to pipeline the operator for this frequency (see Section 1.8 below).
- `name=UserProvidedName` replaces the (ugly and parameter-dependent) entity name generated by FloPoCo for the next operator. This allows in particular to change parameters while keeping the same entity name, so that these changes are transparent to the rest of the project.
- `plainVHDL=yes` instructs FloPoCo to output concise and readable VHDL, using only + and * VHDL operators instead of FloPoCo adders and multipliers. This helps understanding the algorithms used by FloPoCo, but typically prevents or degrades automatic pipelining.

- `useHardMult=no` instructs FloPoCo not to use hard multipliers or DSP block.
- `hardMultThreshold=0.7` instructs FloPoCo to use a hard multiplier (or DSP block) if less than 70% of this hard multiplier are unused. The ratio is between 0 and 1, such that 0 means: any sub-multiplier that does not fully fill a DSP goes to logic; 1 means: any sub-multiplier, even very small ones, will consume a DSP.
- `generateFigures=1`, for some operators, will generate relevant graphics in SVG or LaTeX.
- `dependencyGraph=<no|compact|full>` generates data dependency graphs.

1.6 Do not trust FloPoCo, the test bench is included

FloPoCo is able to generate an infinite number of different operators, and the developers have obviously not tested each of them. This issue is intrinsic to application-specific arithmetic. A simple solution developed in FloPoCo is that each operator comes with a test bench generator that will exercise exactly the operator being generated.

Details about the construction of these test benches can be found in the developer manual (see `http://flopoco.org/` for the latest version). The point to stress here is that the resulting test benches can be trusted by users. The main reason for this is that the test vectors are not built out of the VHDL being generated, but out of the mathematical specification of the operator (as a mathematical function combined with a well-defined rounding, implemented in C++, see Section 3 below). Thus, the probability of a bug in the VHDL being hidden by a bug in the test bench is very low. In addition, the code that generates testbenches is very small, quite boilerplate, and based on reference multiple-precision libraries that are well specified and well established: GMP and MPFR. For this reason, the confidence in the test bench is much higher than the confidence in the generated VHDL.

The test bench generation also provides how the open-source VHDL simulator nvc (`https://github.com/nickg/nvc`) can be invoked to test the circuit. We highly recommend this fast and reliable command line simulator. The command line is sufficient as it tells the user/developer if everything went fine (`0 error(s) encoutered`) or the test cases that failed. In this case, waveform are written by nvc and calls to the open-source waveform viewer gtkwave (`https://gtkwave.sourceforge.net`) are provided that allows to further debug what went wrong.

The user interface to the test bench generation is kept extremely simple, to encourage users to test their FloPoCo operators.

TestBench generation in FloPoCo

The following command builds a divider by 3 for 16-bit integers, and an exhaustive test bench for this operator:

```
flopoco IntConstDiv wIn=16 d=3 TestBench
```

This command creates a VHDL file and a (human-readable) `test.input` file containing all the possible values of the input and the corresponding expected output. It also outputs instructions to launch this test using a choice of VHDL simulators.

Operation-specific testBench generation
For operators having a large input word size, exhaustive testing is out of reach. In such cases, the test should be limited to a smaller number of test vectors. The following command is an example of a (Nfloat) binary32 floating-point adder with 100,000 random tests.

```
flopoco FPAdd we=8 wf=23 TestBench n=100000
```

This testbench actually begins with all the corner-case test vectors that the developers of `FPAdd` could think of, including a few regression test vectors corresponding to past bugs.
In addition, the random number generator used there is specific to the operator under test: with two random bit strings as inputs, the probability of a cancellation in the addition would be quite low, since it only happens when the exponents differ at most by 1. To properly test this important feature of floating-point addition, the random generator used has been slightly tweaked to make cancellations more frequent. See the developer manual for more details.

1.7 Obscure branches and code attics

The installation instructions on the web site currently use the `master` branch of the Git repository of FloPoCo. Beyond this branch, there are quite a few other, more experimental branches and forks where one may find rarer operators, or code snapshots that ensure that a publication is reproducible.

In principle, developers are encouraged to merge their work in the master branch as soon as it may be of general use. They are also encouraged to clean up the master branch of experimental code, over-parameterization for research purpose, or obsolete variants for which a better option is available. Unfortunately, they often do not have the time for these two time-consuming tasks, and we do apologize for this. Any feedback on what should be prioritized will be welcome.

Quite often, a change to the FloPoCo framework (or to one of the libraries it uses) breaks some operators. If these operators are not fixed by their original developers, the maintainers end up disabling them in the master branch. Usually the code is just unplugged, not removed, : it is kept in "attic" directories or "obscure branches", where it may still compile, or not. In any case, this code is still there for review and/or porting to the current framework, should the need arise. Examples of code which is currently disabled but would deserve to be revived include the HOTBM function approximator by Jrmie Detrey [1], the LNS operators by Caroline Collange [2], the FPGA-specific random number generators by David Thomas [3], among many others.

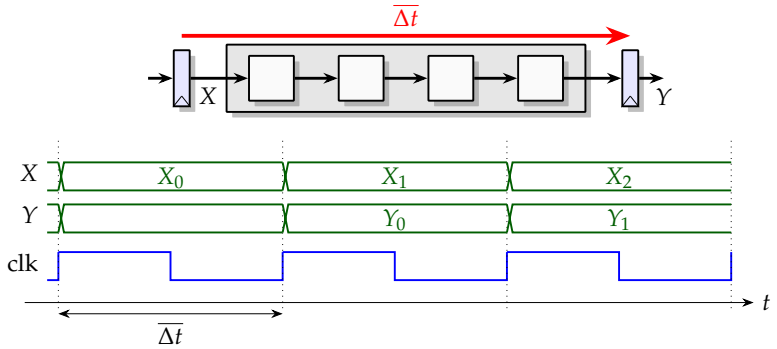
1.8 Automatic pipelining, the user point of view

Most operators presented in this book are combinatorial circuits: they have no memory, the result only depends of the input, not of the past history of the operator. This is another direct consequence of the definition of an operator as the composition of a mathematical function[1] and a well-defined rounding.

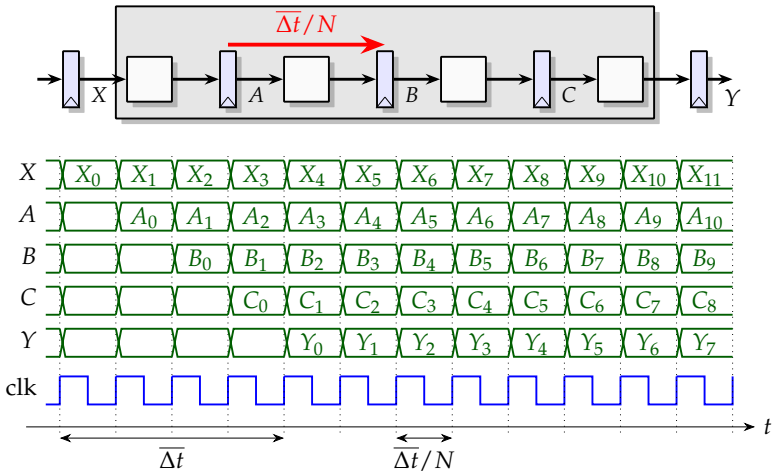
Another point of view is that the directed graph of subcomponents that describes the circuit (down to the gates or whatever elementary processing elements of the technology) is acyclic. However, each component of this graph needs some time (and some energy) to perform its computation. If an input is presented at time t_0, the correct output will be computed after some delay Δt (measured in seconds). This delay may depend on the input. Its maximum value over all the possible inputs is called the *critical path delay* of the circuit, noted $\overline{\Delta t}$. If the operator is sandwiched between input registers and output registers as illustrated by Figure 6(a), then the input registers may present a new input every $\overline{\Delta t}$ period, and the circuit will have the time to process this input (and register the result in the output registers) before the next input is presented. In other words, the operator can operate at frequency $1/\overline{\Delta t}$.

Pipelining is a technique that allows an operator to function at a higher frequency by dividing the circuit graph into N slices called *pipeline stages*, determined such that the critical path delay of each slice is about

[1] The main exceptions are the floating-point accumulators and the filters. There could be some day iterative variants of dividers, CORDIC operators, and in general digit-recurrence algorithms, but none of these is currently available in FloPoCo: the tool provides only the unrolled, combinatorial variants.



(a) A combinatorial operator, with registers that produce its inputs and consume its outputs



(b) The same operator, pipelined into $N = 4$ stages: frequency can be multiplied by 4.

Figure 6: Pipelining a combinatorial operator

$\overline{\Delta t}/N$. Registers are inserted between slices to hold intermediate computations, as illustrated by Figure 6(b). If the pipeline is ideally balanced, and ignoring the delay added by these extra registers, the frequency can become $N/\overline{\Delta t}$: compared to the combinatorial circuit, the frequency has been multiplied by N. The cost is additional registers. In practice, however, it is difficult to achieve a perfectly balanced pipeline, and the registers add some delay which limits the achievable frequency [4, 5].

Pipelining combinatorial operators is simple in theory but time-consuming if performed by hand. As Figure 7 illustrates, actual operators are often more complex than Figure 6. FloPoCo therefore automates this task. The interface for it is simply the target frequency provided through the `frequency` parameter. An important remark on Figure 7 is that FloPoCo does not add any input and output registers. All it does is insert the intermediate (pipeline) registers. The "pipeline depth" it reports is the number of synchronization barriers inserted (this information is also provided in comments before each entity declaration in the generated VHDL).

The resulting pipeline is quite good for practical purposes. Note however that FloPoCo does not pretend to generate an optimal pipeline (an optimal pipeline would minimize the overall cost of the registers while achieving a given target frequency [6]). It does not even guarantee that the operator will operate at the prescribed frequency – we are at the mercy of the backend tools that will transform the VHDL into an actual circuit. Indeed, these tools are the proper place for fine-tuning a pipeline. However, it does guarantee that the pipeline is well synchronized (and `TestBench` adapts to pipelined operators for users to check). Hence, cases where the required frequency is not reached or even too high can be easily fixed by adjusting the target frequency accordingly. Note that this may be done on a per-operator basis, as in:

```
flopoco FPAdd frequency=200 wE=11 wF=53 FPMult frequency=300 wE=8 wF=23
```

Sandwiching an operator with register

In order to measure the actual operating frequency at which an operator runs, it is important to add the registers on the inputs and outputs. FloPoCo provides an operator that sandwiches between registers the operator preceding it on the command line (just like `TestBench` tests the preceding operator).

```
flopoco frequency=400 FPAdd wE=8 wF=23 RegisterSandwich
```

1.9 Synthesis helper tools

FloPoCo also provides (in the `tools/` directory) small python scripts that will synthesize an operator for an FPGA using either Vivado or Quartus. These tools read in the VHDL file the target information, launch synthesis accordingly, and report the main synthesis result on the console.

2 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

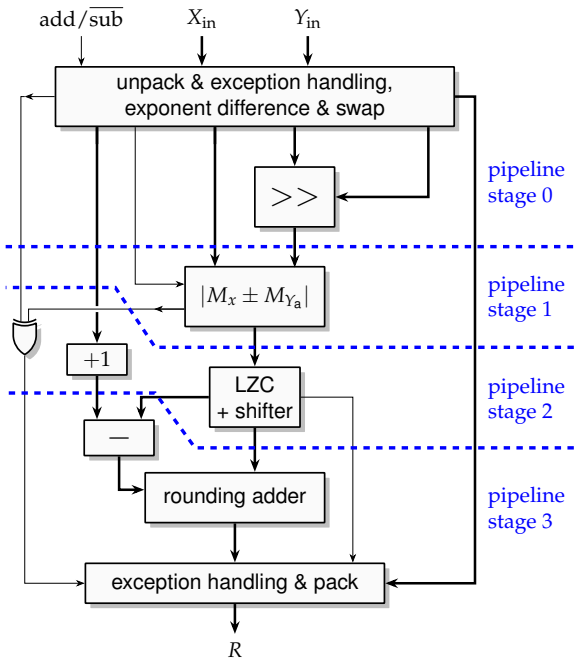
```
./flopoco TutorialOperator
```

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

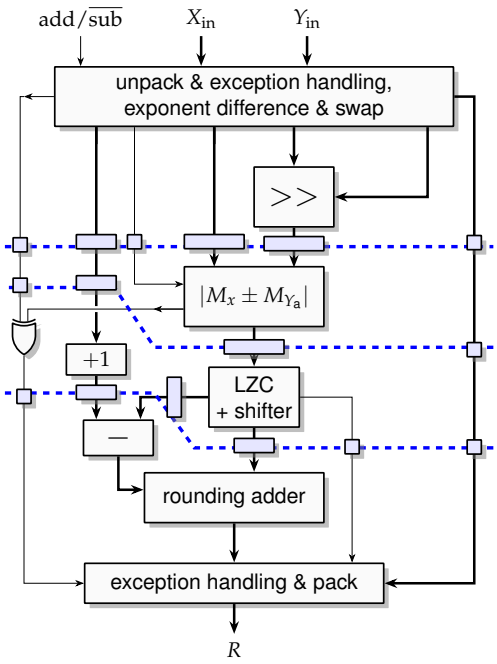
Now run in a terminal

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

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



(a) Adding 3 synchronization barriers – FloPoCo reports a pipeline depth of 3 – means that there are 4 pipeline stages



(b) The corresponding registers are inserted automatically by FloPoCo

Figure 7: Pipelining a floating-point adder

2.1 Overview of FloPoCo code organization

In the `code/` directory, there are several subdirectories:

HighLevelCore is a library of arithmetic optimization code, independent of the actual HDL generation. For instance, it will contain the code for polynomial or multipartite approximation to functions, or code that optimizes the shift-and-add tree of a constant multiplier.

VHDLOperators is the library contains the bulk of FloPoCo code that generates VHDL.

FloPoCoBin contains the source of the `flopoco` executable (linking both previous libraries) and of a few other helper executables.

This separation is still work in progress, therefore expect to find in VHDLOperators a lot of code that should be in HighLevelCore.

The core of FloPoCo is the `Operator` class (in VHDLOperators). 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.

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.

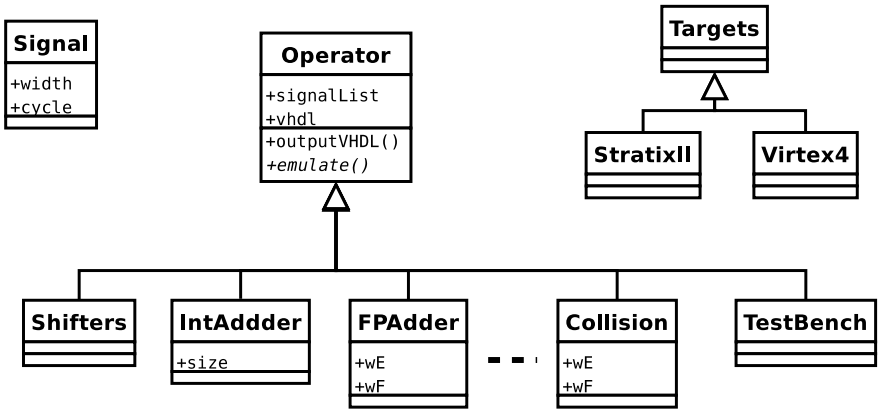
The command-line parser is in `UserInterface` but in principle you won't have to edit it. It takes information from each operator's documentation strings defined in their `OperatorDescription` construction. Again, we hope that you can design the interface to your operator by imitation of existing ones.

2.2 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 the `CMakeList.txt` of his directory.

If you are unfamiliar with the CMake system[2], there is little to learn, really. Straightforward imitation of existing content will cover most cases, otherwise `cmake` is well documented.



2.3 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);
```

[2] `http://www.cmake.org/`

```
    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.

2.4 Adding delay information

A latency (in seconds) may be passed as first optional argument to `declare()`. This value decribes the contribution of this VHDL statement to the critical path, and will be used to pipeline the operator (see Section 4 for details). This latency is best defined using methods of `Target`: this way the pipeline will adapt to the target. See `FPAddSinglePath` for examples.

2.5 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.

2.6 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.

2.7 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.

2.8 Linking against FloPoCo

All the operators provided by the FloPoCo command line are available programmatically in libFloPoCo.

There are two ways to instantiate an operator. One is to use its factory, which replicates the command line.

The other one is to use the constructor (whose interface is defined in the corresponding `hpp` file). There is no one-to-one correspondance.

- Sometimes the command-line interface regroups several Operators. For instance FPAddSinglePath and FPAddDualPath are, for historical reasons, two different Operators with two different constructors, but are exposed on the interface as one FPAdd with an option.
- Sometimes, the constructor has more interface options than what ends up in the command-line interface, either that some options turned out not to provide interesting solutions, or that they were designed for research purpose only.

3 Test bench generation

3.1 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.

3.2 `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`.

3.3 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 common requirement is *faithful rounding*: the operator should return one of the two FP values surrounding the exact result[3] These values may be obtained thanks to the *rounding up* and *rounding down* modes supported by MPFR. See `FPExp` or `FPLog` for examples in floating point, and `FixSinCos` for an example in fixed point (which is also an example where the function has two outputs).

3.4 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.

[3] The allowed error is twice as large as correct rounding, so a correctly rounded result on p bits is as accurate as a faithful result on $p+1$ bits. However, (for reasons too long to detail here) a faithful operator may be much, much cheaper to build than a correctly rounded one. In such cases it is cheaper to build a faithful operator to $p+1$ bits than a correctly rounded one to precision p.

- 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()`. See `FPExp.cpp` and `FPAdd.cpp` for the examples above.

3.5 Corner-cases and regression tests

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

3.6 Regression testing, build test

The command `flopoco Autotest Operator=FPadd` tests FPAdd for a range of relevant parameter values. For each parameter vector, it reports if the FloPoCo command succeeded, if the VHDL was syntactically correct, and if the simulation of this VHDL was successful.

The enumeration of the parameter vectors is defined in the (optional) static method `FPAdd::unitTest()`. This is a relatively recent addition to the framework, and not all operators support it, but it is strongly advised to set it up in the early steps of operator development: it helps uncover bugs that occur for specific values of the parameters.

The command `flopoco Autotest Operator=all` runs this test for all the operators that support it. It may take long...

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 [7].

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 8 (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 8, 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`.

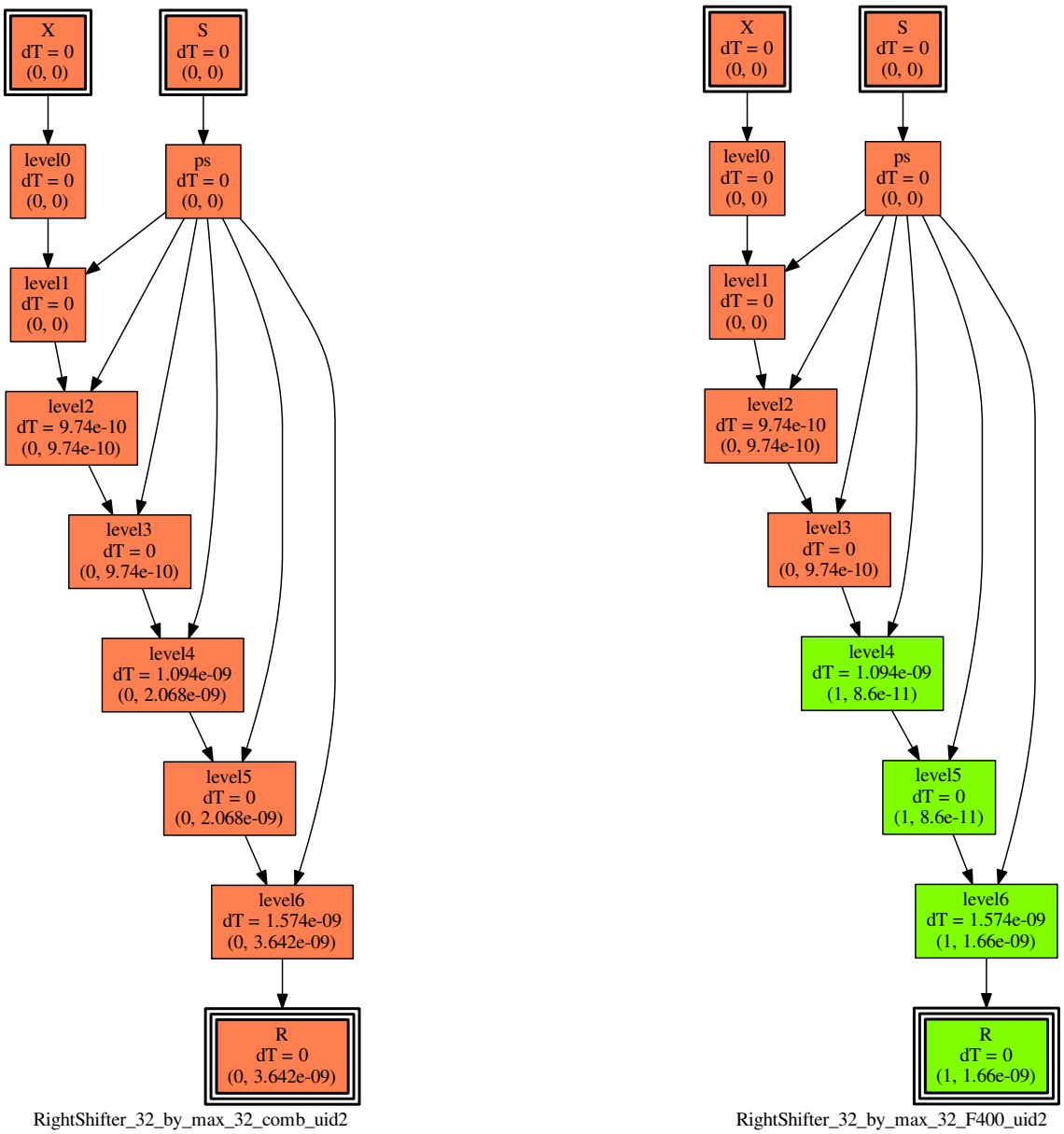


Figure 8: S-Graph for a combinatorial 8-bit barrel shifter, combinatorial (left) and pipelined (right)

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. 8, 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 $\tau_1 > \tau_2$.

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`.

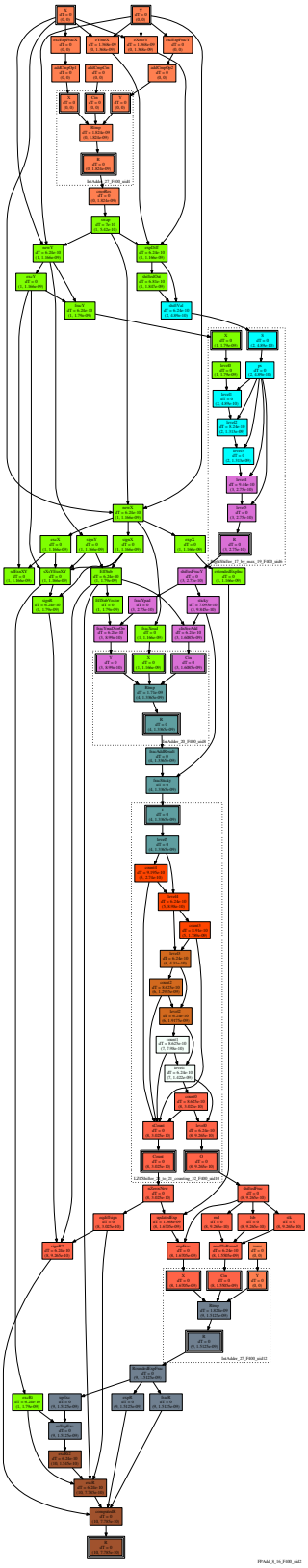


Figure 9: S-Graph for a pipelined FPAddSinglePath operator. Zoom on the Shifter component and observe that it has been pipelined for its context.

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 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 [8, 9, 10, 11] (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 3. 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`.

5.1 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.

Table 3: 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 `cst` $\cdot 2^{pos}$
`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 10: Bit heap obtained for a 16-bit, 8-tap half-sine pulse shaping FIR filter operator `FixHalfSine`

The `BitHeap` class also offers methods to visualize a bit heap as a dot diagram like the one shown in Figure 10. 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.

5.2 Compressor tree generation

Once the data structure of the BitHeap is created by using the methods of Table 3, 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`.

6 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] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2005, pp. 328–333.

[2] P. D. Vouzis, C. Collange, and M. G. Arnold, "A novel cotransformation for LNS subtraction," *Journal of Signal Processing Systems*, vol. 58, no. 1, pp. 29–40, 2010.

[3] D. B. Thomas, "Parallel generation of gaussian random numbers using the table-hadamard transform," in *FPGAs for custom computing machines*, 2013.

[4] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *Proceedings of the 29th annual international symposium on computer architecture*. IEEE Computer Society, 2002, pp. 14–24.

[5] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 2002, pp. 25–34.

[6] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5 – 35, 1991.

[7] 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

[8] 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.

[9] 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.

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

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