FPGA-specific arithmetic pipeline design using FloPoCo

Bogdan Pasca, Arénaire

CARAMEL, 17/02/2011

Outline

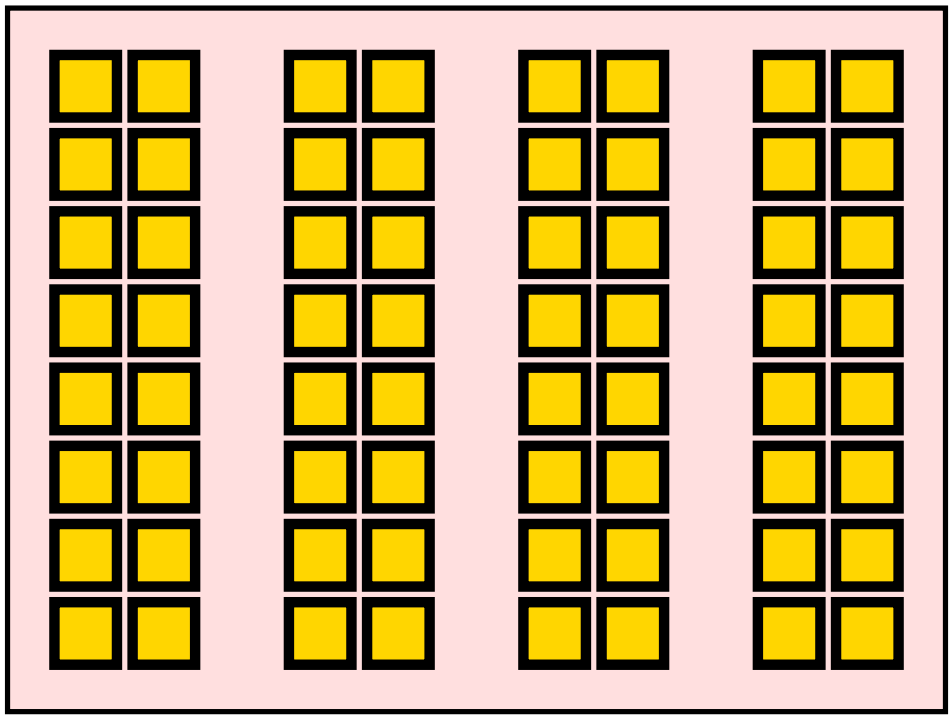
FPGAs and floating-point

What's an FPGA?

Field **P**rogrammable **G**ate **A**rray

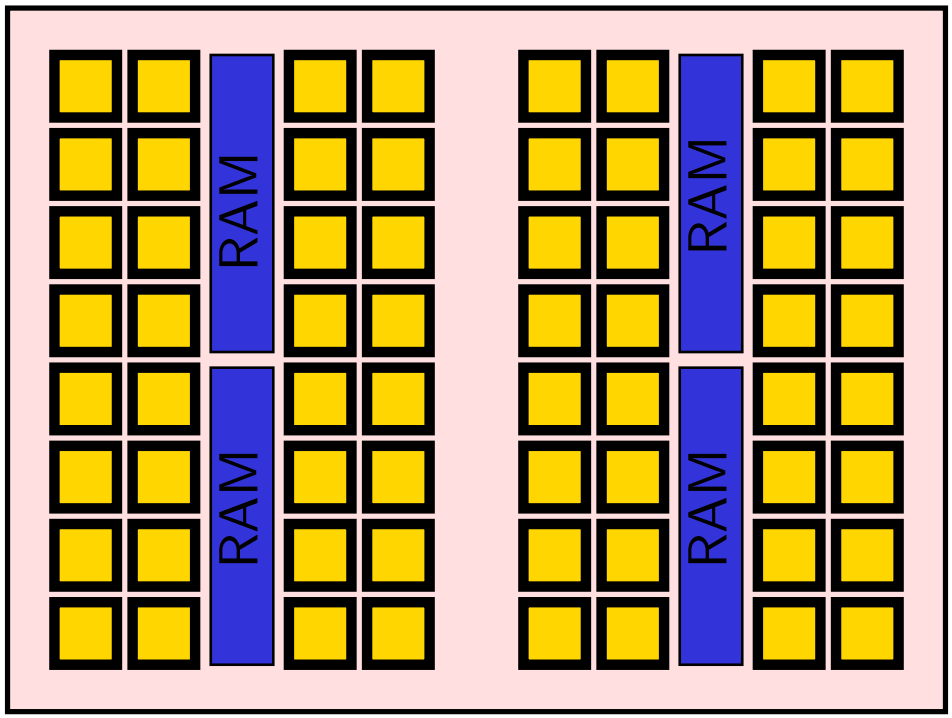
- integrated circuit
- has a regular architecture (hence **array**)
- logic elements can be programmed to perform various functions

Modern FPGA Architecture



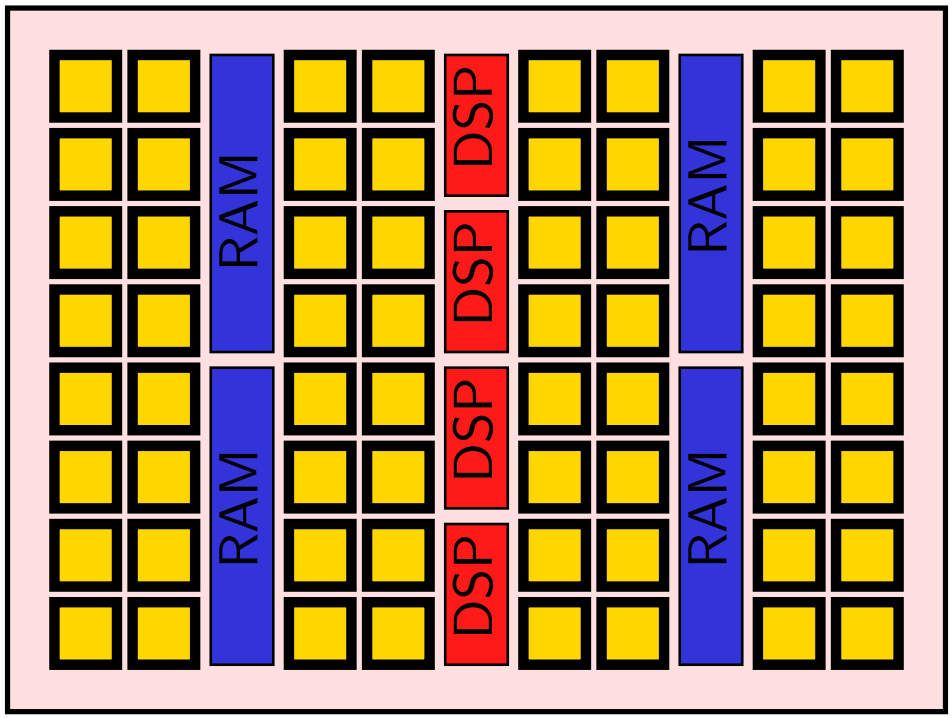
- a set of **configurable** logic elements

Modern FPGA Architecture



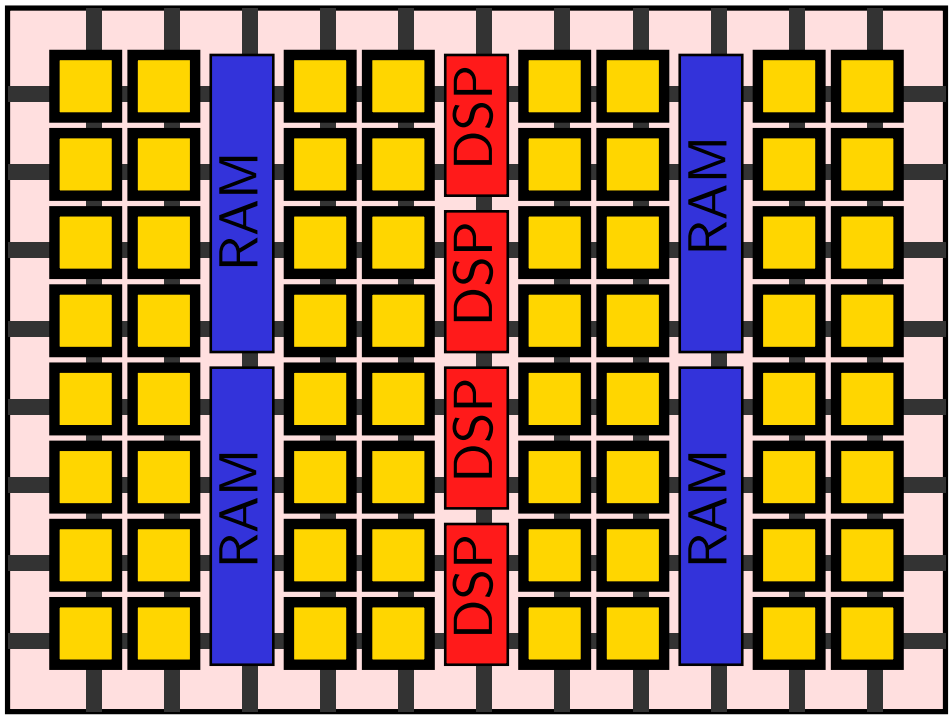
- a set of **configurable** logic elements
- on chip memory blocks

Modern FPGA Architecture



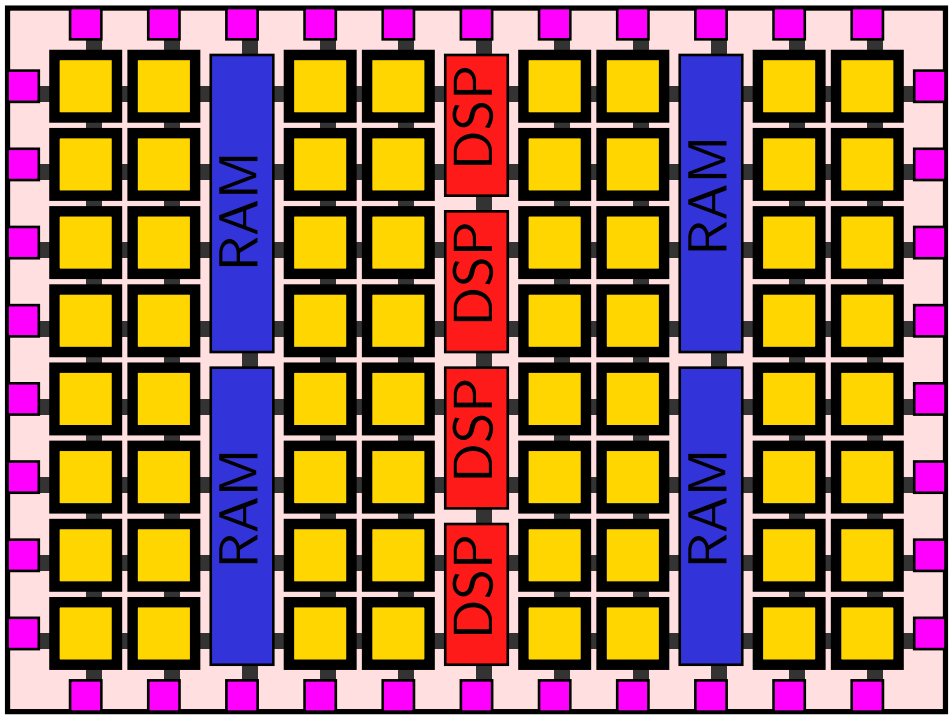
- a set of **configurable** logic elements
- on chip memory blocks
- digital signal processing (DSP) blocks (including multipliers)

Modern FPGA Architecture



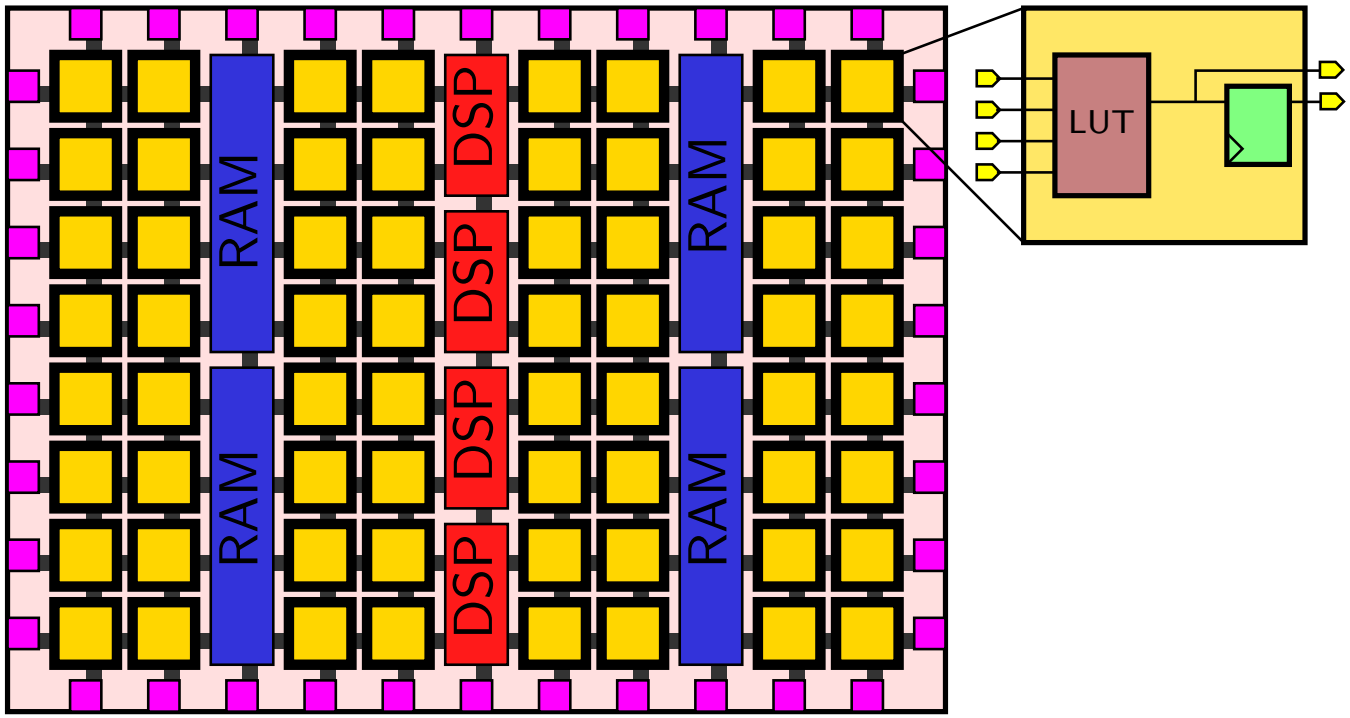
- a set of **configurable** logic elements
- on chip memory blocks
- digital signal processing (DSP) blocks (including multipliers)
- connected by a **configurable** wire network

Modern FPGA Architecture



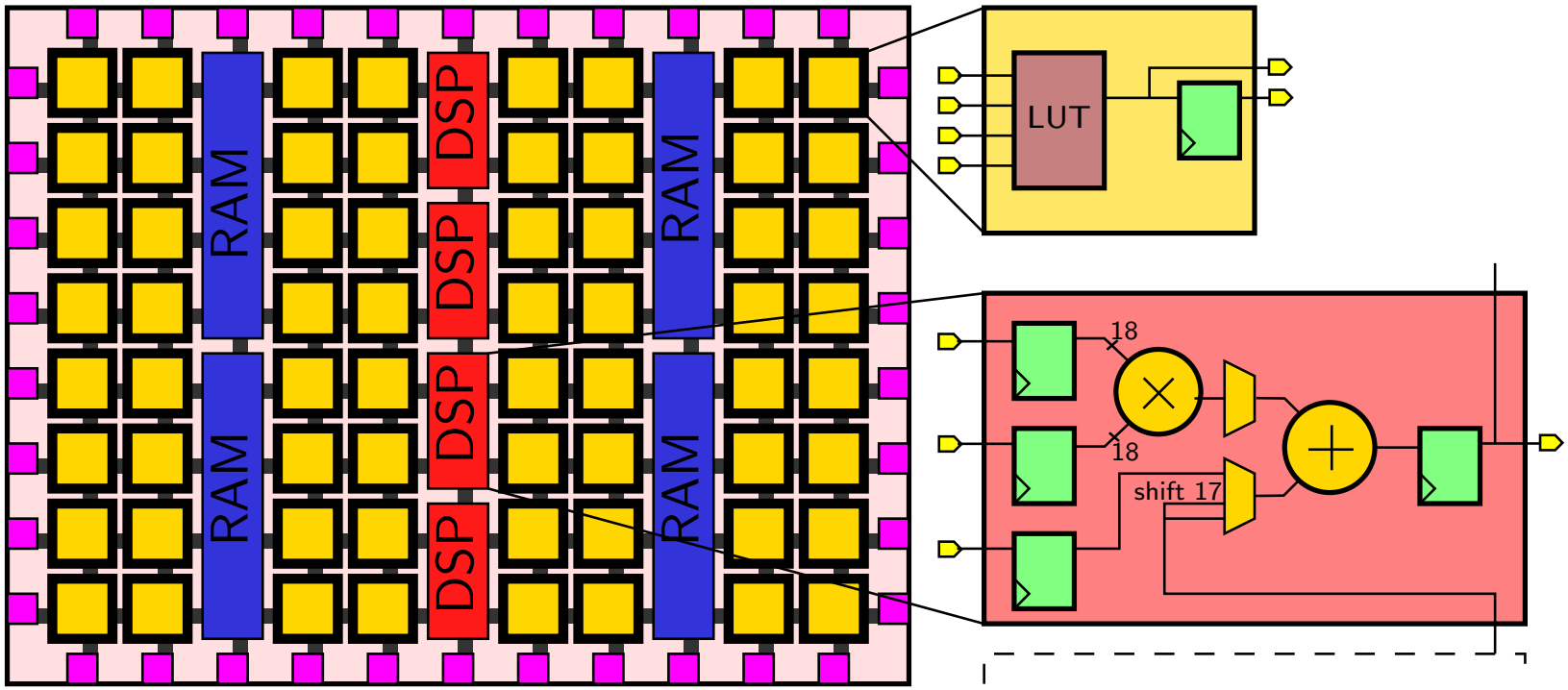
- a set of **configurable** logic elements
- on chip memory blocks
- digital signal processing (DSP) blocks (including multipliers)
- connected by a **configurable** wire network
- all connected to outside world by I/O pins

Modern FPGA Architecture



- a set of **configurable** logic elements
- on chip memory blocks
- digital signal processing (DSP) blocks (including multipliers)
- connected by a **configurable** wire network
- all connected to outside world by I/O pins

Modern FPGA Architecture



- a set of **configurable** logic elements
- on chip memory blocks
- digital signal processing (DSP) blocks (including multipliers)
- connected by a **configurable** wire network
- all connected to outside world by I/O pins

A bit of history

Year	1995	2011	
FPGA	XC4010	XC6VHX565T	5SGXAB
Capacity (K LE)	1	500	1.000
DSPs	-	1K	1.5K
Bock RAM	-	2K (18Kb)	2K (20Kb)
Frequency (MHz)	10	600	
FPAdder $(w_E = 6, w_F = 9)$[1]	28%	0.05%	0.025%
FPMultiplier $(w_E = 6, w_F = 9)$	44%	*[2]	*
FPDivider $(w_E = 6, w_F = 9)$	46%	0.1%	0.05%

[1] Shirazi et al., *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines*(1995)

[2] Multiplications are usually implemented using DSPs on modern FPGAs

A bit of history

Year		1995	2011	
FPGA		XC4010	XC6VHX565T	5SGXAB
Capacity (K LE)		1	500	1.000
DSPs		-	1K	1.5K
Bock RAM		-	2K (18Kb)	2K (20Kb)
Frequency (MHz)		10	600	
FPAdder	$(w_E = 6, w_F = 9)$[1]	28%	0.05%	0.025%
FPMultiplier	$(w_E = 6, w_F = 9)$	44%	*[2]	*
FPDivider	$(w_E = 6, w_F = 9)$	46%	0.1%	0.05%

FPGAs are now large enough to implement complex datapaths

[1] Shirazi et al., *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines*(1995)

[2] Multiplications are usually implemented using DSPs on modern FPGAs

So, are FPGAs any good at floating-point in 2011?

So, are FPGAs any good at floating-point in 2011?

Today's basic operations: $+, -, \times$

- ☢ Highly **optimized FPU** in the processor
- ☢ Each operator **10x slower** in an FPGA
- ★ Massive **parallelism** on an FPGA

$\rightarrow$ FPGA faster than PC, but no match to GPGPU, Cell ...

So, are FPGAs any good at floating-point in 2011?

Today's basic operations: $+, -, \times$

- ☢ Highly **optimized FPU** in the processor
- ☢ Each operator **10x slower** in an FPGA
- ★ Massive **parallelism** on an FPGA
- → FPGA faster than PC, but no match to GPGPU, Cell ...

If you lose according to a metric, change the metric.

Peak figures for double-precision floating-point exponential[3].

- Pentium core: 20 cycles / DPExp @ 3GHz: **150 MDPExp/s**
- FPGA: 1 DPExp/cycle @ 400MHz: **400 MDPExp/s**
- Chip vs chip: 8 Pentium cores vs 150 FPExp/FPGA

★ **Power consumption** also better

(Intel MKL vector libm, vs FPExp in FloPoCo version 2.0.0)

[3] de Dinechin, Pasca. *Floating-point exponential functions for DSP-enabled FPGAs*(2010)

The FloPoCo project: Not your neighbour's FPU

Useful operators that would not be economical in a processor

The FloPoCo project: Not your neighbour's FPU

Useful operators that would not be economical in a processor

- ★ Elementary functions (sine, exponential, logarithm...)
- ★ Algebraic functions ($\frac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- ★ Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- ★ Floating-point sums, dot products, sums of squares
- ★ Specialized operators: constant multipliers, squarers, ...
- Complex arithmetic
- ★ LNS arithmetic
- ★ Decimal arithmetic
- Interval arithmetic
- ...

The FloPoCo project: Not your neighbour's FPU

Useful operators that would not be economical in a processor

- ★ Elementary functions (sine, exponential, logarithm...)
- ★ Algebraic functions ($\frac{x}{\sqrt{x^2+y^2}}$, polynomials, ...)
- ★ Compound functions ($\log_2(1 \pm 2^x)$, e^{-Kt^2}, ...)
- ★ Floating-point sums, dot products, sums of squares
- ★ Specialized operators: constant multipliers, squarers, ...
- Complex arithmetic
- ★ LNS arithmetic
- ★ Decimal arithmetic
- Interval arithmetic
- ...
- Oh yes, basic operations, too.

VHDL Limitations

One instance: double-precision, Virtex4, 400MHz - FPExp:

- 52 pipeline stages
- 37 subcomponents
- 6000 lines of VHDL

VHDL Limitations

One instance: double-precision, Virtex4, 400MHz - FPExp:

- 52 pipeline stages
- 37 subcomponents
- 6000 lines of VHDL vs 600 lines of FloPoCo

VHDL Limitations

One instance: double-precision, Virtex4, 400MHz - FPExp:

- 52 pipeline stages
- 37 subcomponents
- 6000 lines of VHDL vs 600 lines of FloPoCo

Our questions for today:

How to productively design an optimized architecture?

VHDL Limitations

One instance: double-precision, Virtex4, 400MHz - FPExp:

- 52 pipeline stages
- 37 subcomponents
- 6000 lines of VHDL vs 600 lines of FloPoCo

Our questions for today:

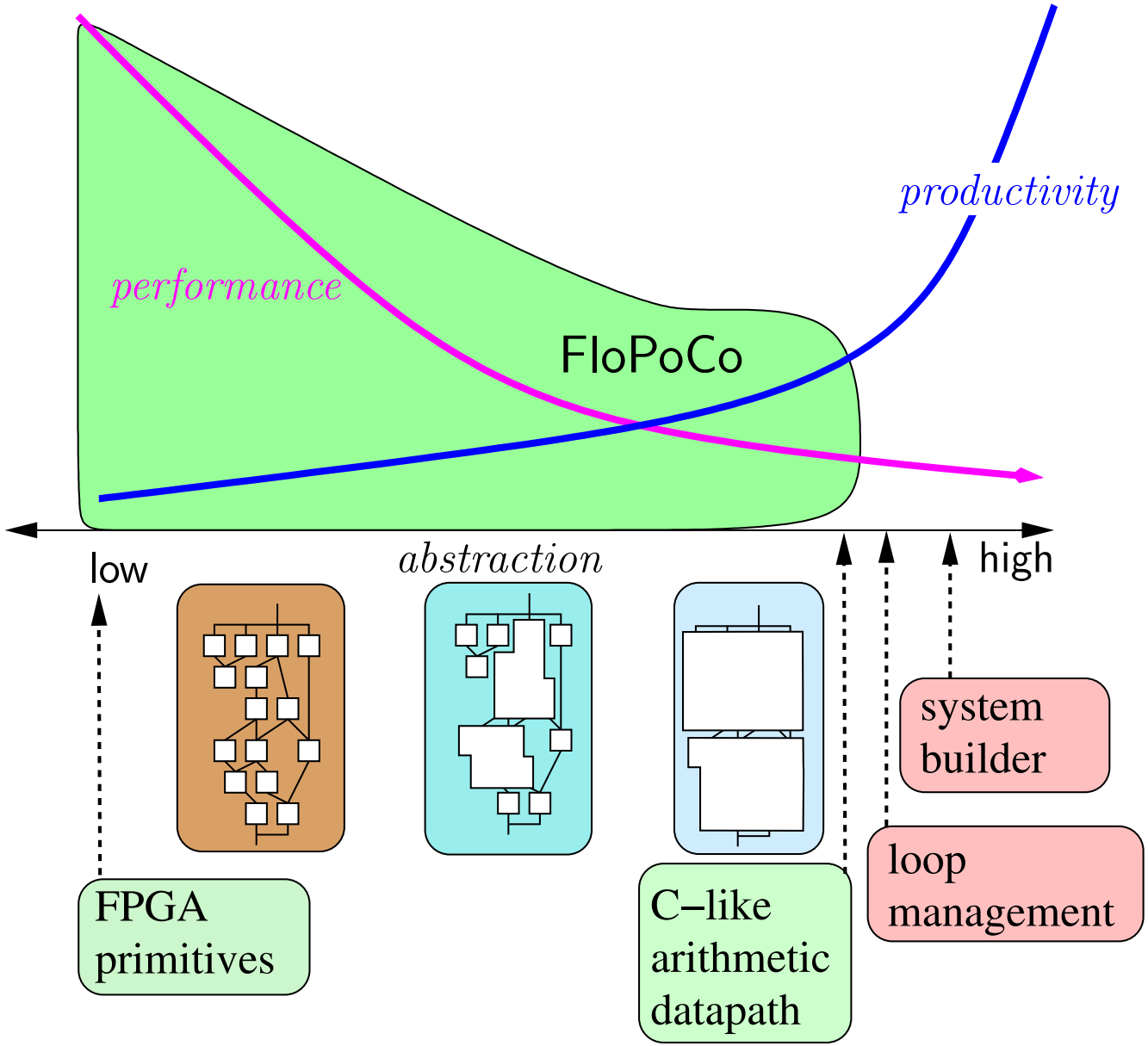
How to productively design an optimized architecture?

How to be **future-proof**?

- need a different precision
- target a different FPGA family (different multiplier sizes)
- need faster frequency

Datapath design using FloPoCo

A question of granularity



Sum of squares: performance approach

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

Sum of squares: performance approach

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required

Sum of squares: performance approach

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless

Sum of squares: performance approach

$$x^2 + y^2 + z^2$$

(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - $(x^2 + y^2) + z^2$: asymmetrical

Sum of squares: performance approach

$$x^2 + y^2 + z^2$$

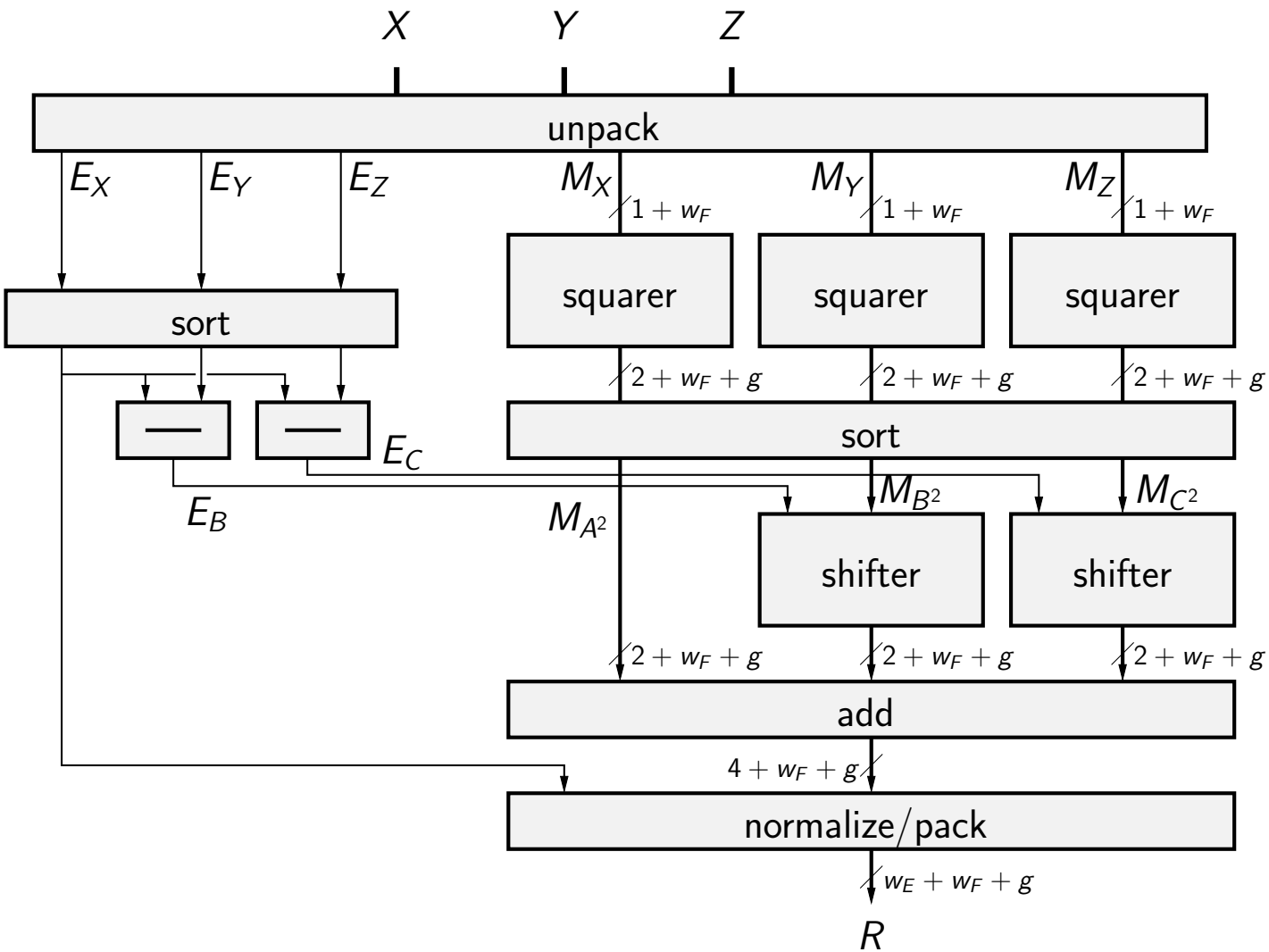
(not a toy example but a useful building block)

- A square is simpler than a multiplication
  - half the hardware required
- x^2, y^2, and z^2 are positive:
  - one half of your FP adder is useless
- Accuracy can be improved:
  - 5 rounding errors in the floating-point version
  - $(x^2 + y^2) + z^2$: asymmetrical

The FloPoCo recipe for optimal performance

- build a fixed-point architecture
- keep the FP interface
- ensure a clear accuracy specification

Architecture: Optimal Performance



The FloPoCo recipe for high productivity

flopoco FPPipeline expr.in 8 23

```
/* sum of squares */
r = sqr(x) + sqr(y) + sqr(z);
output r;
```

```
flopoco FPPipeline expr.in 8 23

Final report:
|   |---Entity IntSquarer_24_uid8:
|   |      Pipeline depth = 4
|   |---Entity IntAdder_33_f400_uid10:
|   |      Pipeline depth = 1
(...)
|---Entity FPSquarer_8_23_23_uid30:
|      Pipeline depth = 7
|   |---Entity FPAdder_8_23_uid41_RightShifter:
|   |      Pipeline depth = 1
|   |---Entity IntAdder_27_f400_uid45:
|   |      Pipeline depth = 1
|   |---Entity LZCShifter_28_to_28_counting_32_uid50:
|   |      Pipeline depth = 5
|   |---Entity IntAdder_34_f400_uid52:
|   |      Pipeline depth = 2
|---Entity FPAdder_8_23_uid41:
|      Pipeline depth = 14
|   |---Entity FPAdder_8_23_uid63_RightShifter:
|   |      Pipeline depth = 1
|   |---Entity IntAdder_27_f400_uid67:
|   |      Pipeline depth = 1
|   |---Entity LZCShifter_28_to_28_counting_32_uid72:
|   |      Pipeline depth = 5
|   |---Entity IntAdder_34_f400_uid74:
|   |      Pipeline depth = 2
|---Entity FPAdder_8_23_uid63:
|      Pipeline depth = 14
Entity Pipeline2:
   Pipeline depth = 36
Output file: flopoco.vhdl
```

Synthesis Results

A few results for floating-point sum-of-squares on Virtex4:

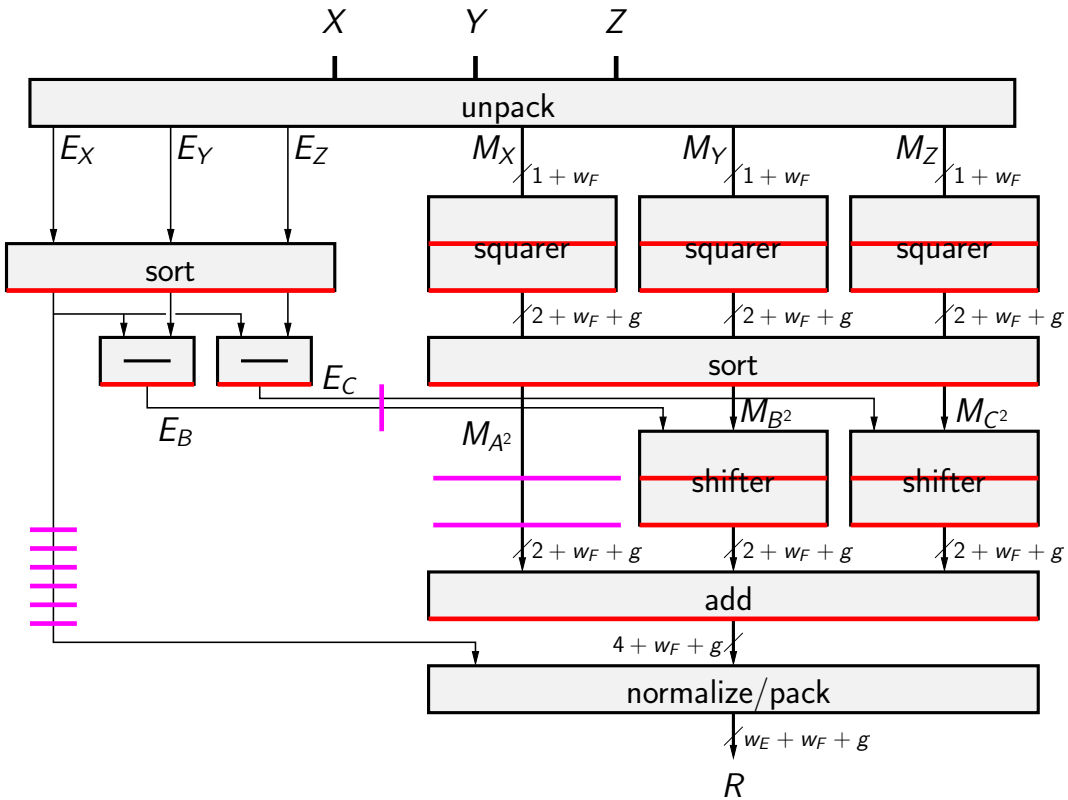
Single Precision	area	performance	design time
LogiCore classic[4]	1282 slices, 20 DSP	43 cycles @ 353 MHz	hours
FloPoCo compiler	1047 slices, 9 DSP	36 cycles @ 357 MHz	seconds
FloPoCo custom	453 slices, 9 DSP	11 cycles @ 368 MHz	days

Double Precision	area	performance	design time
LogiCore classic	3942 slices, 48 DSP	52 cycles @ 279 MHz	hours
FloPoCo compiler	3354 slices, 18 DSP	49 cycles @ 348 MHz	seconds
FloPoCo custom	1845 slices, 18 DSP	16 cycles @ 362 MHz	**seconds**

- ★ all performance metrics improved, FLOP/s/area more than doubled
- ★ custom operator more accurate, and symmetrical

[4] Assembling floating-point operators

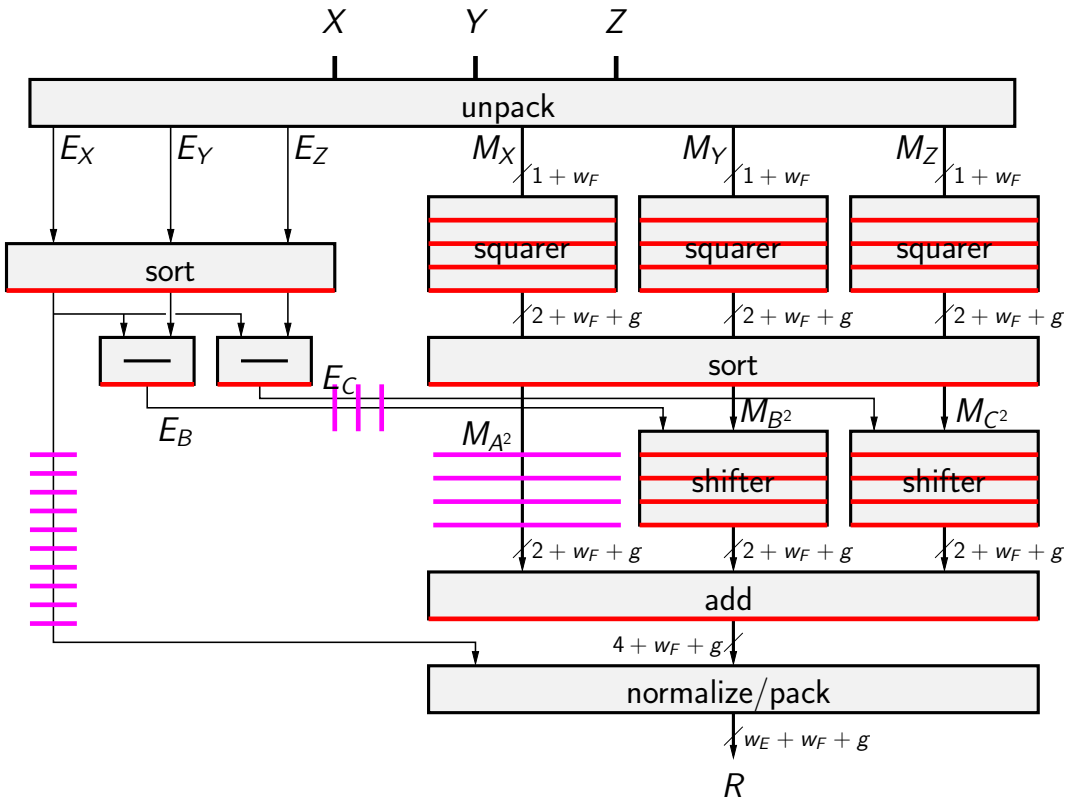
Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption

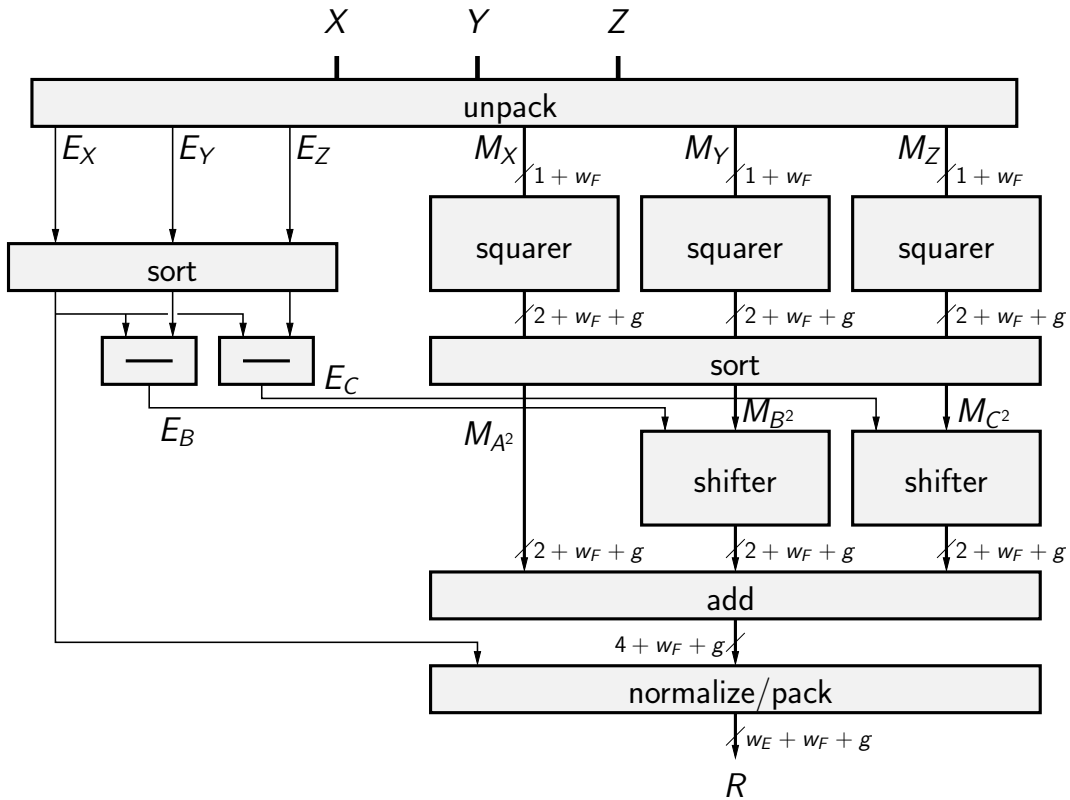
Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)

Adapting to context: frequency-directed pipeline



One operator does not fit all

- Low frequency, low resource consumption
- Faster but larger (more registers)
- Combinatorial

Inside FloPoCo

FloPoCo

FloPoCo is not a library, but a *generator* of operators written in C++.

- Command line syntax: a sequence of operator specifications
- Options: target frequency, target hardware, ...
- Output: synthesizable VHDL.

Here should come a demo!

FloPoCo

FloPoCo is not a library, but a *generator* of operators written in C++.

- Command line syntax: a sequence of operator specifications
- Options: target frequency, target hardware, ...
- Output: synthesizable VHDL.

Here should come a demo!

FloPoCo also provides a framework for designing these operators!

`http://flopoco.gforge.inria.fr/`

A modestly object-oriented approach

FloPoCo is not a C++-based HDL, but more of a mix

A modestly object-oriented approach

FloPoCo is not a C++-based HDL, but more of a mix

- VHDL generation is "print-based"

```
1 vhdl << "SoS <= EA(wE-1 downto 0) & Fraction;" ;
```

  - easy to port existing work (FPLibrary)
  - easy learning curve for the VHDL-litterate
  - at least the expressive power of VHDL!

A modestly object-oriented approach

FloPoCo is not a C++-based HDL, but more of a mix

- VHDL generation is "print-based"

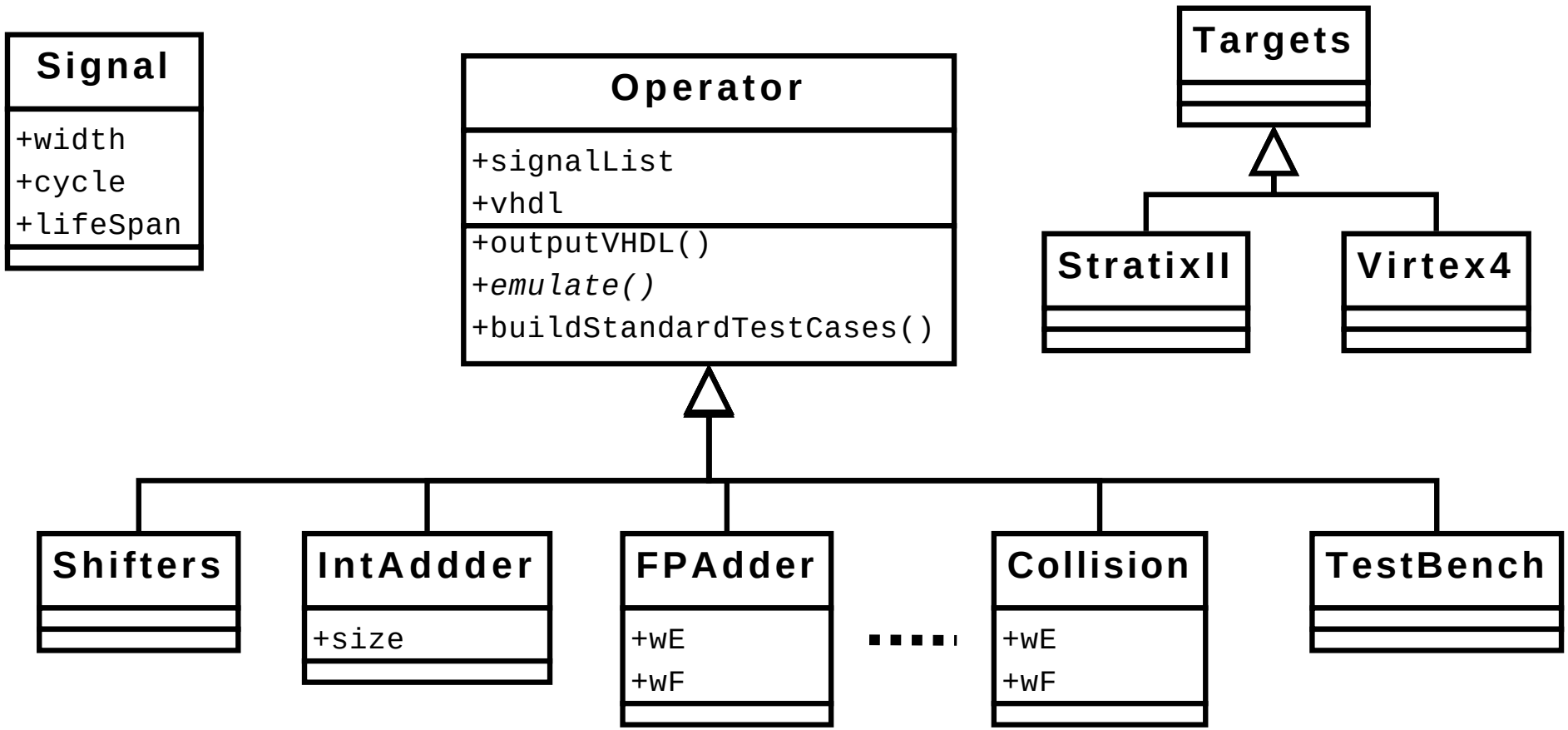
```
1 vhdl << "SoS <= EA(wE-1 downto 0) & Fraction;" ;
```

  - easy to port existing work (FPLibrary)
  - easy learning curve for the VHDL-litterate
  - at least the expressive power of VHDL!

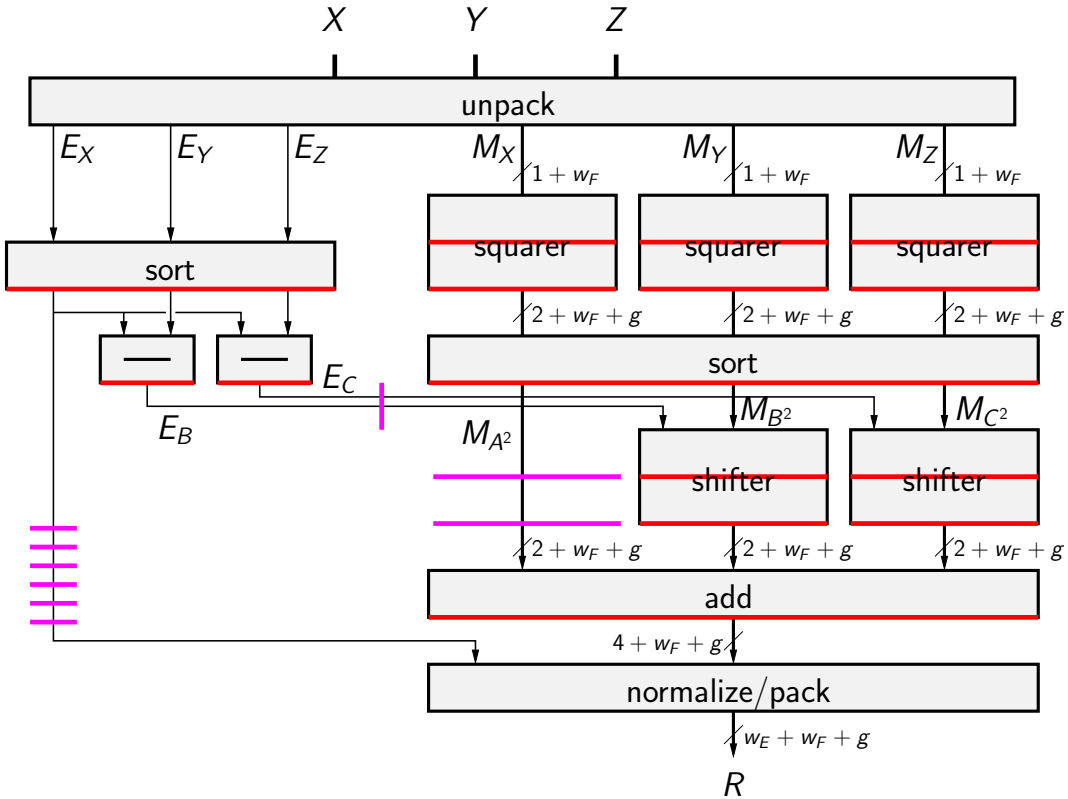
- Many helper functions help doing the `prints`
  Example: VHDL signal declaration

```
1 vhdl << declare("SoS", wE+wF+g)
2      << " <= EA(wE-1 downto 0) & Fraction;" ;
```

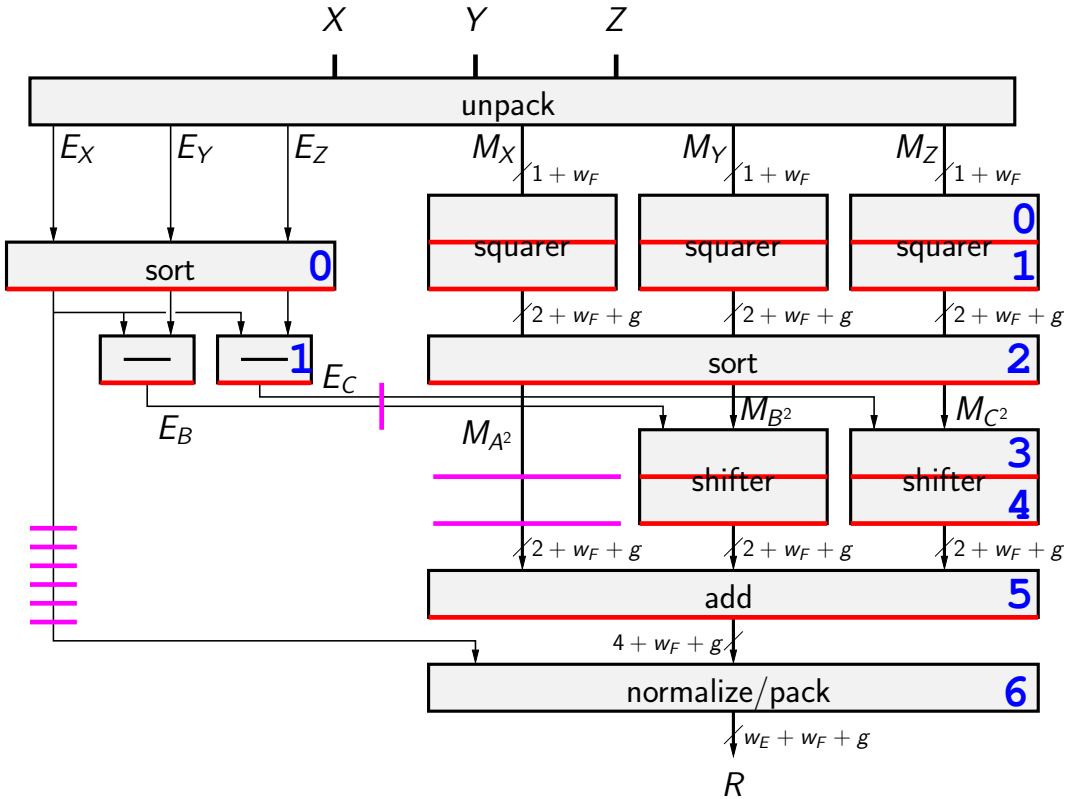
FloPoCo class hierarchy



Pipeline made easy

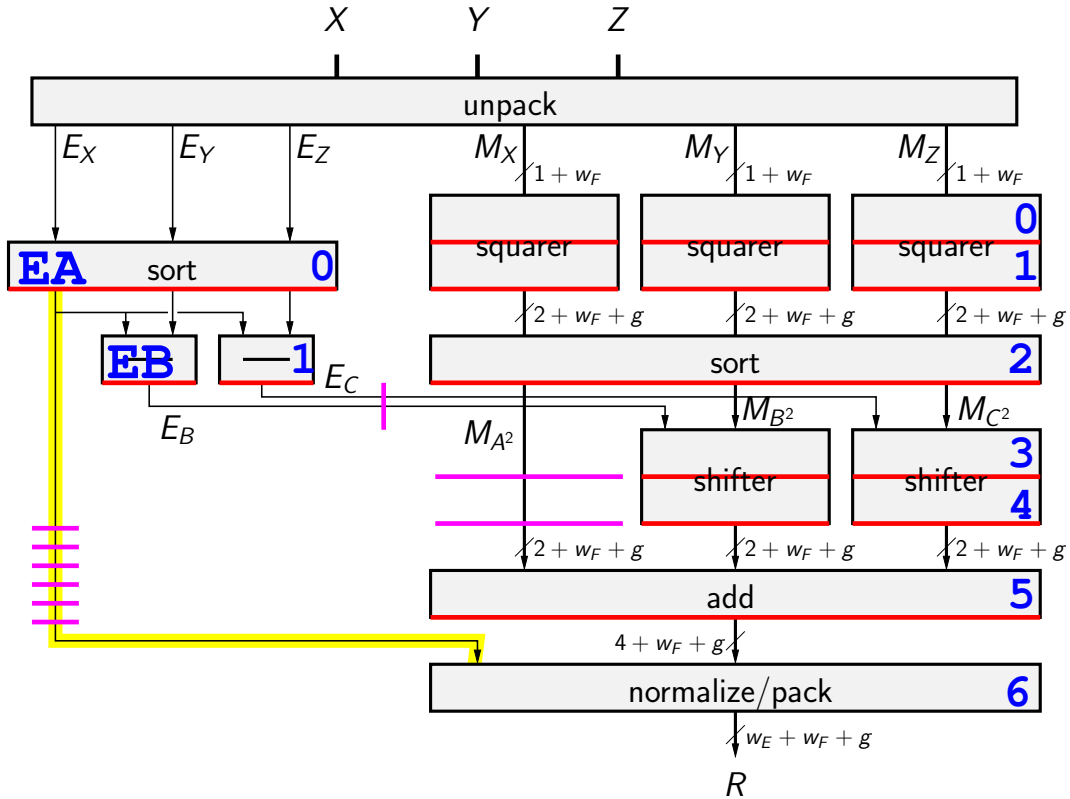


Pipeline made easy



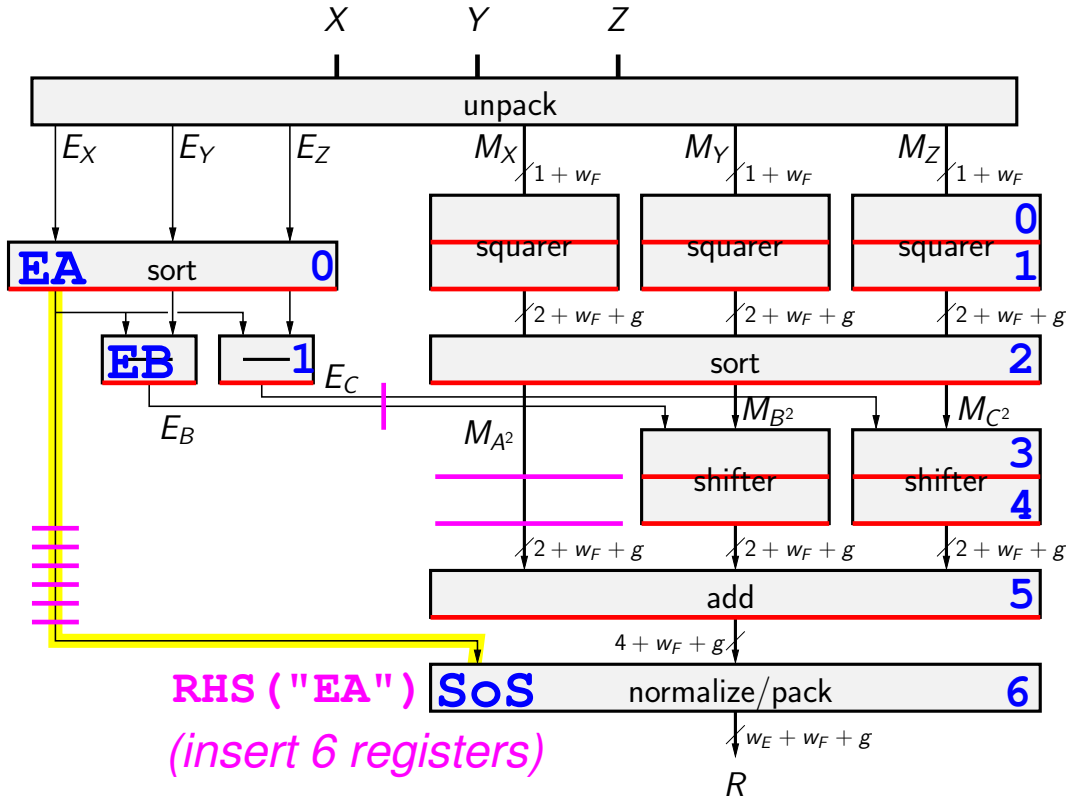
- Notion of current cycle during VHDL output
- Each signal has an active cycle

Pipeline made easy




```
1 vhdl << declare("EA", wE) << " <= ... ;" ; // cycle(EA)=0
2 nextCycle();
3 vhdl << declare("EB", wE) << " <= ... ;" ; // cycle(EB)=1
4 setCycle(0);
5 vhdl << ... ;
```

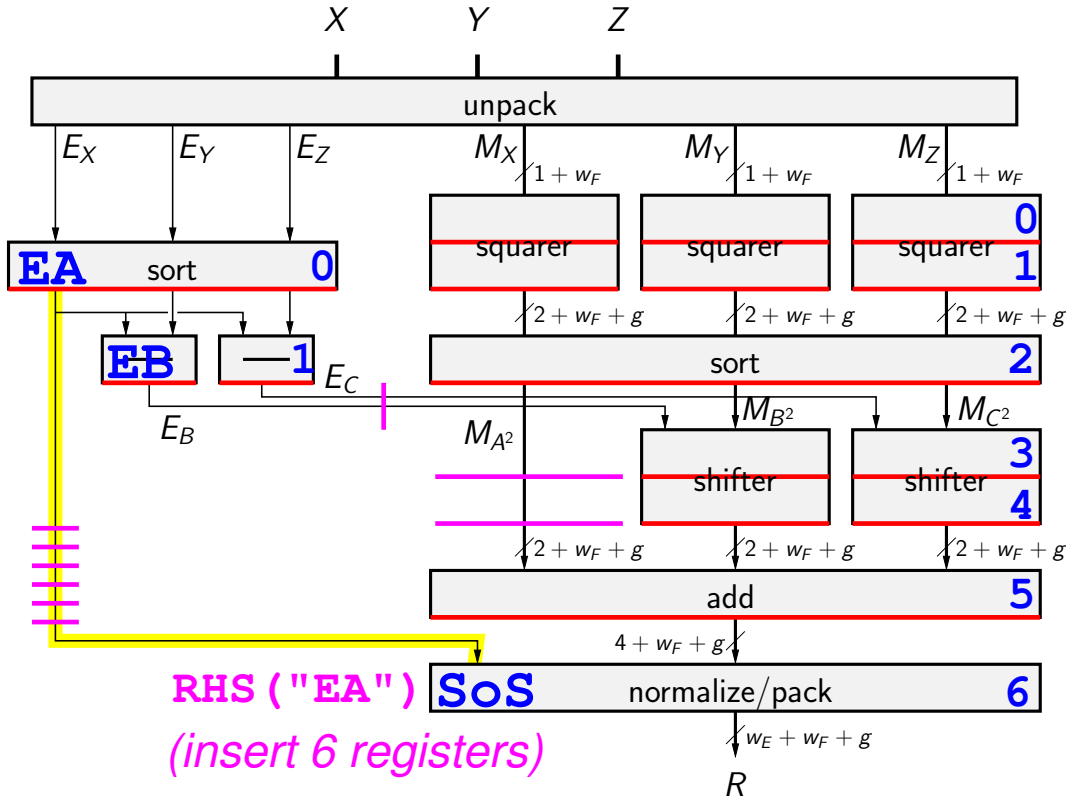
Pipeline made easy



Look for signal names on the right-hand side, and delay them by (current - active) cycles:

```
1  vhdl <<    declare("SoS", wE+wF+g)
2       <<    " <=  EA(wE-1 downto 0) & Fraction;"    ;
```

Pipeline made easy

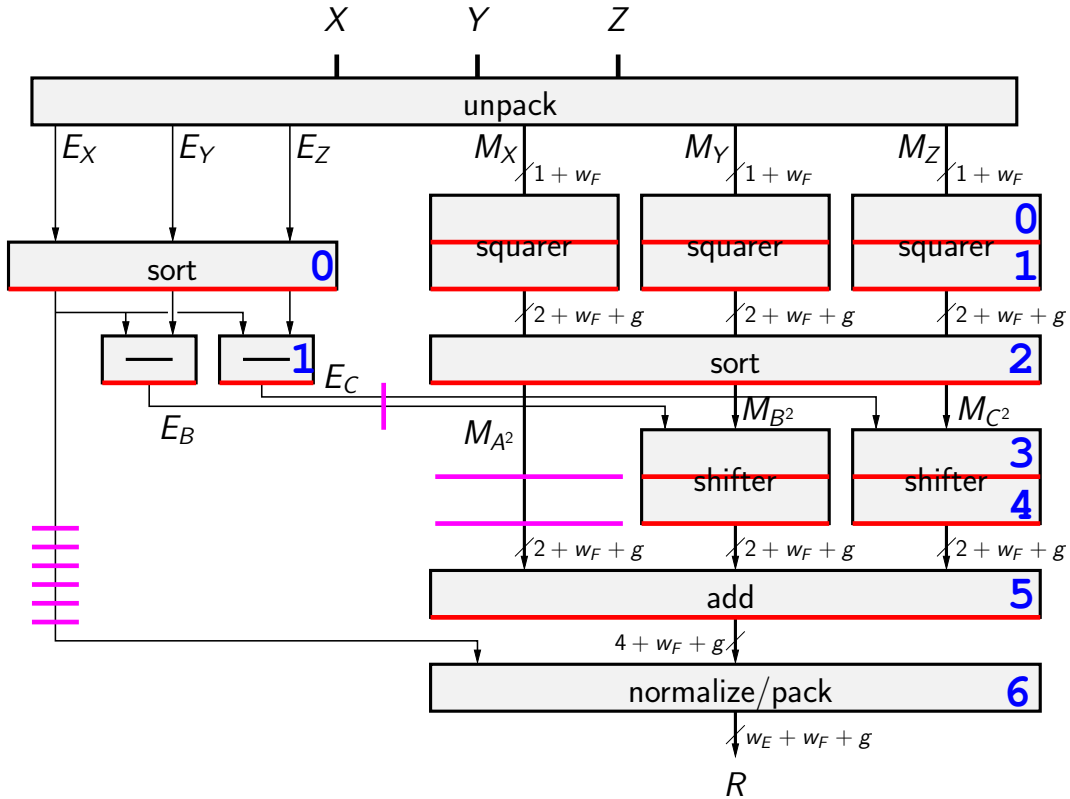


Look for signal names on the right-hand side, and delay them by (current - active) cycles: Output

```
1 SoS <=  EA_d6(wE-1 downto 0) & Fraction_d1;
```

(and transparently declare and build the needed shift registers)

Pipeline made easy



Managing the current cycle:

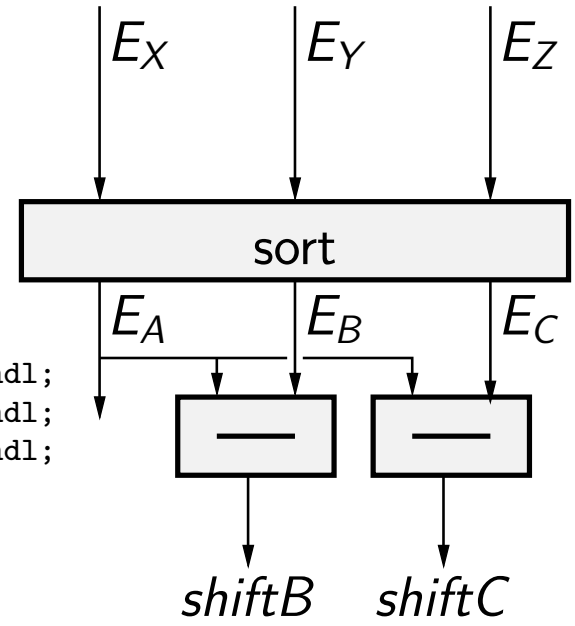
```
n=getCycle();
setCycle(n);
nextCycle();
syncCycleWithSignal("EA");
```

Frequency-directed pipelining:

```
manageCriticalPath(
  target->adderDelay(n) );
```

In-depth view of FloPoCo code

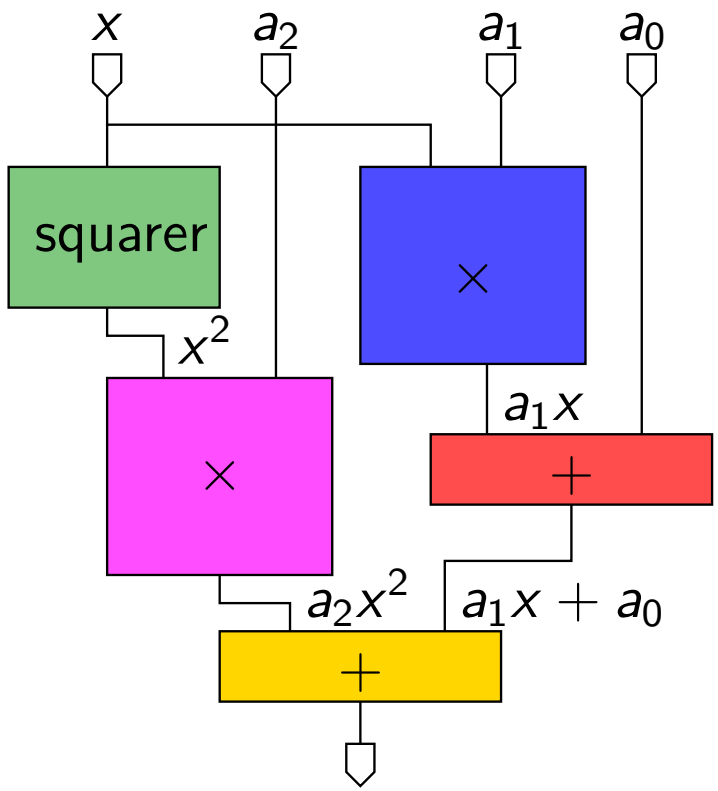
```
//The expSort box
manageCriticalPath(  // evaluate the delay
   target->adderDelay(wE+1)   // exp. diff.
 + target->localWireDelay(wE) // wE is the fanout
 + target->lutDelay()    );   // mux

// determine the max of the exponents
vhdl << declare("DEXY", wE+1) << " <=   ('0' & EX) - ('0' & EY);" << endl;
vhdl << declare("DEYZ", wE+1) << " <=   ('0' & EY) - ('0' & EZ);" << endl;
vhdl << declare("DEXZ", wE+1) << " <=   ('0' & EX) - ('0' & EZ);" << endl;
vhdl << declare("XltY") << "<= DEXY(wE);" << endl;
vhdl << declare("YltZ") << "<= DEYZ(wE);" << endl;
vhdl << declare("XltZ") << "<= DEXZ(wE);" << endl;

// rename  exponents  to A,B,C with A>=(B,C)
vhdl << declare( "EA", wE)  <<  " <= EZ when (XltZ='1') and (YltZ='1')  else "
                            <<  "EY when (XltY='1') and (YltZ='0')  else "
                            <<  "EX;"   << endl;
vhdl << declare( "EB", wE)  <<  " <= " << (...);
vhdl << declare( "EC", wE)  <<  " <= " << (...)

// the parallel subtractions
manageCriticalPath(target->adderDelay(wE-1) );

vhdl << declare( "shiftB", wE) <<  " <=  (EA(wE-1 downto 0) - EB (wE-1 downto 0);";
vhdl << declare( "shiftC", wE) <<  " <=  (EA(wE-1 downto 0) - EC (wE-1 downto 0);";
```



VHDL Output

```
DEXY <=   ('0' & EX) - ('0' & EY);
DEYZ <=   ('0' & EY) - ('0' & EZ);
DEXZ <=   ('0' & EX) - ('0' & EZ);
XltY <=   DEXY(8);
YltZ <=   DEYZ(8);
XltZ <=   DEXZ(8);
EA <= EZ when (XltZ='1') and (YltZ='1')  else
      EY when (XltY='1') and (YltZ='0')  else
      EX;
EB <= (...)
EC <= (...)
--Synchro barrier, entering cycle 1--
shiftB <= EA_d1(7 downto 0) - EB_d1(7 downto 0) ;
shiftC <= EA_d1(7 downto 0) - EC_d1(7 downto 0) ;
```

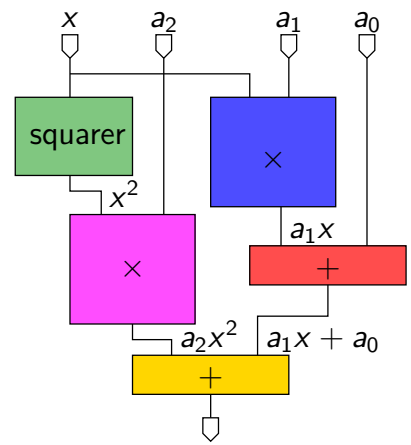
Multiple Path Designs



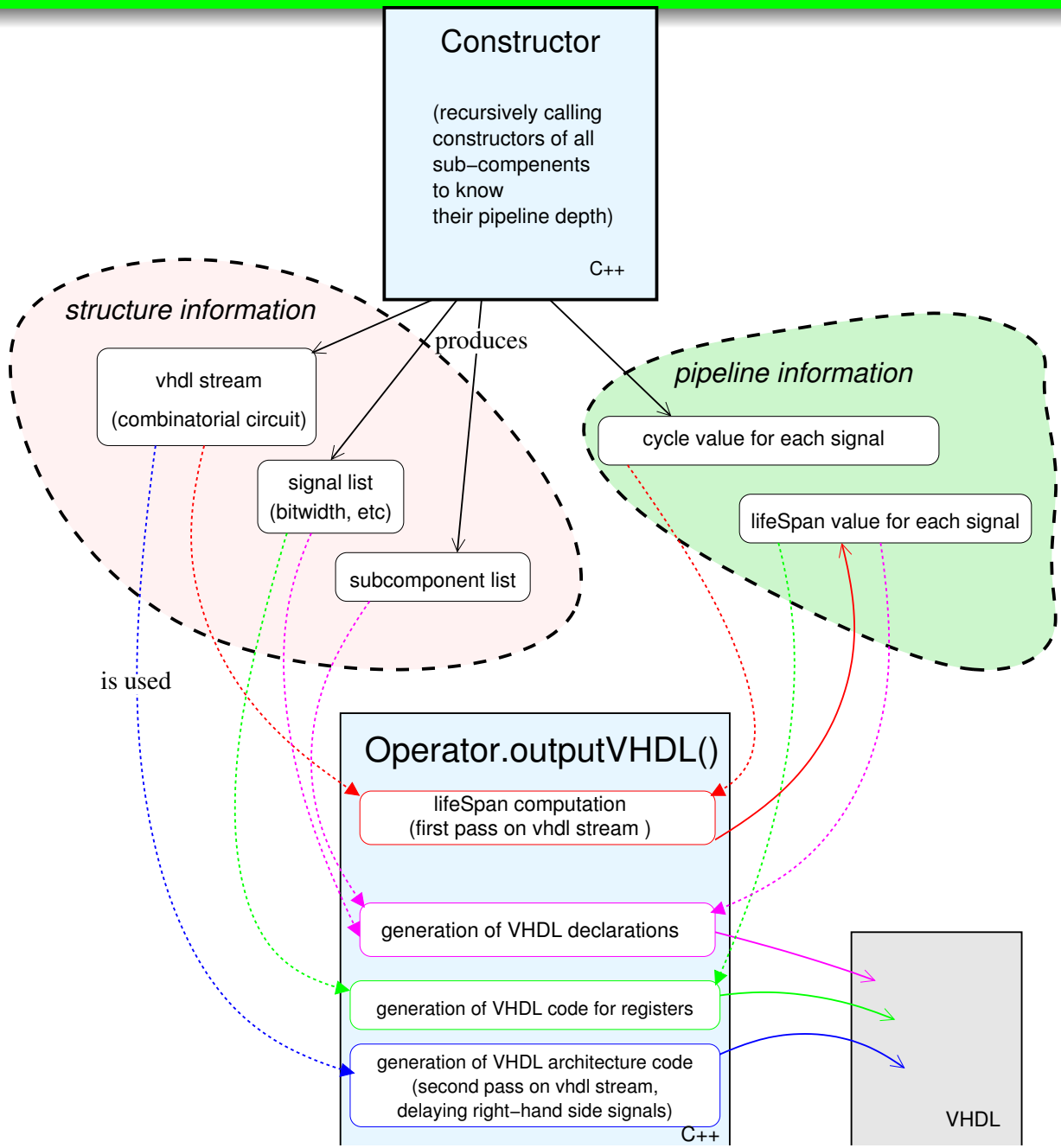
Multiple Path Designs

```
int wE, wF;
addFPInput("X",wE,wF);
addFPInput("a2",wE,wF);
addFPInput("a1",wE,wF);
addFPInput("a0",wE,wF);

FPSquarer *fps = new FPSquarer(target, wE, wF);
oplist.push_back(fps);

inPortMap (fps, "X", "X");
outPortMap(fps, "R", "X2");
vhdl << instance(fps, "squarer");

syncCycleFromSignal("X2");// advance depth
nextCycle();//register level

FPMultiplier *fpm = new FPMultiplier(target,wE,wF);
oplist.push_back(fpm);

inPortMap (fpm, "X", "X2");
inPortMap (fpm, "Y", "a2");
outPortMap(fpm, "R", "a2x2");
vhdl << instance(fpm, "fpMuliplier_a2x2");

//describe the second thread
setCycleFromSignal("a1"); -- the current cycle = 0

inPortMap (fpm, "X", "X");
inPortMap (fpm, "Y", "a1");
outPortMap(fpm, "R", "a1x");
vhdl << instance(fpm, "fpMuliplier_a1x");

syncCycleFromSignal("a1x");// advance depth
nextCycle();//register level
```

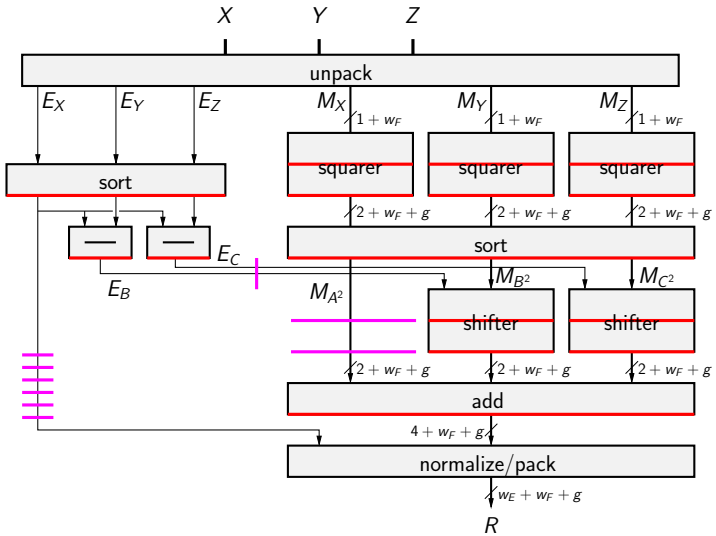
```
FPAdder *fpa = new FPAdder(target, wE, wF);
oplist.push_back(fpa);

inPortMap (fpa, "X", "a1x");
inPortMap (fpa, "Y", "a0");
outPortMap(fpa, "R", "a1x_p_a0");
vhdl << instance(fpa, "fpAdder_a1x_p_a0");

syncCycleFromSignal("a1x_p_a0");//advance
//join the threads
syncCycleFromSignal("a2x2");//possibly advance
nextCycle();//register level

inPortMap (fpa, "X", "a2x2");
inPortMap (fpa, "Y", "a1x_p_a0");
outPortMap(fpa, "R", "a2x2_p_a1x_p_a0");
vhdl << instance(fpa, "fpAdder_a2x2_p_a1x_p_a0");

syncCycleFromSignal("a2x2_p_a1x_p_a0");
vhdl << "R <= a2x2_p_a1x_p_a0; " << endl;
```



Operator data-flow



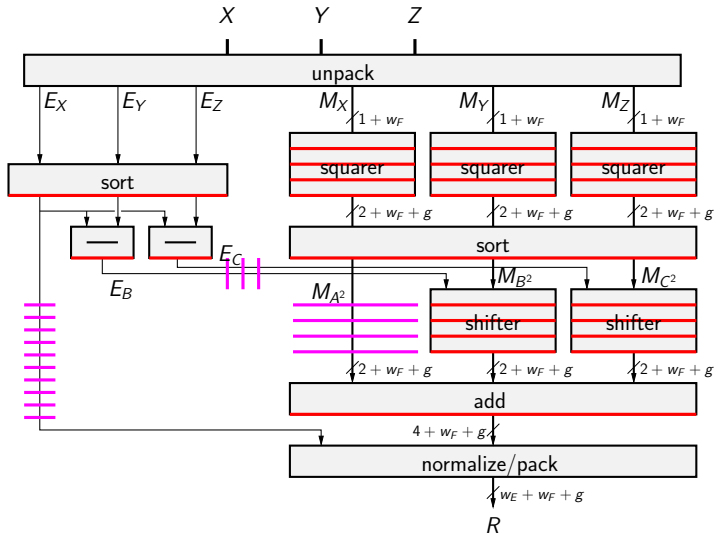
Pipeline made easy



Correct-by-construction pipelines, and more

- Conceptually simple

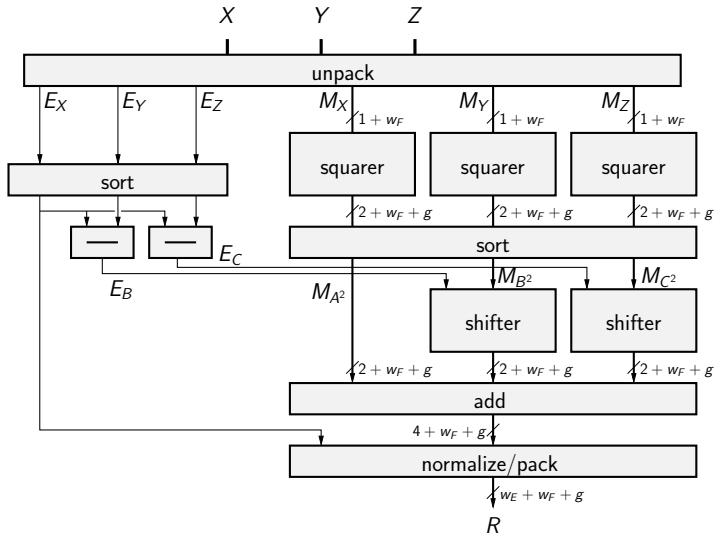
Pipeline made easy



Correct-by-construction pipelines, and more

- Conceptually simple
- Adapts to random insertions of pipeline levels anywhere
  - to break the critical path
  - for frequency-directed pipelining

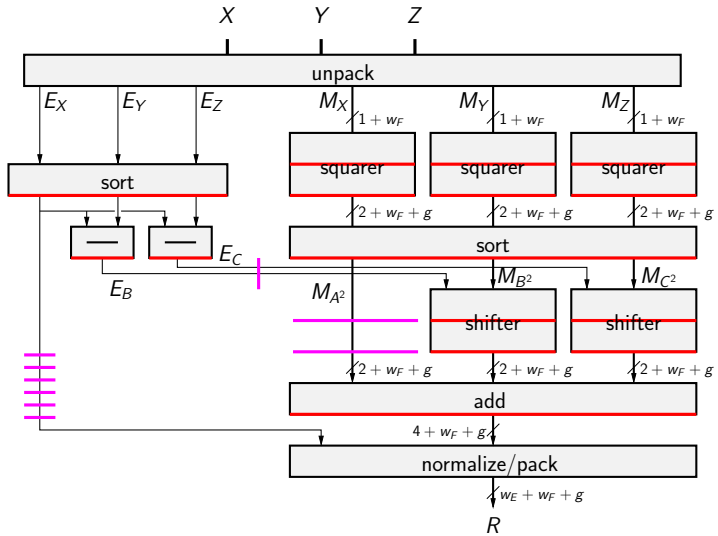
Pipeline made easy



Correct-by-construction pipelines, and more

- Conceptually simple
- Adapts to random insertions of pipeline levels anywhere
  - to break the critical path
  - for frequency-directed pipelining
- Gracefully degrades to a combinatorial operator

Pipeline made easy



Correct-by-construction pipelines, and more

- Conceptually simple
- Adapts to random insertions of pipeline levels anywhere
  - to break the critical path
  - for frequency-directed pipelining
- Gracefully degrades to a combinatorial operator
- Keeps the "print-based" philosophy

Believe it or not, FloPoCo code is much shorter than the VHDL it generates.

Signals automatically managed by FloPoCo

```
1  signal R2pipe, R2pipe_d1, R2pipe_d2, R2pipe_d3, R2pipe_d4, R2pipe_d5, R2pipe_d6,
       R2pipe_d7, R2pipe_d8, R2pipe_d9, R2pipe_d10, R2pipe_d11 :  std_logic_vector(30
       downto 0);
2  signal EX :  std_logic_vector(7 downto 0);
3  signal EY :  std_logic_vector(7 downto 0);
4  signal EZ :  std_logic_vector(7 downto 0);
5  signal DEXY :  std_logic_vector(8 downto 0);
6  signal DEYZ :  std_logic_vector(8 downto 0);
7  signal DEXZ :  std_logic_vector(8 downto 0);
8  signal XltY, XltY_d1, XltY_d2, XltY_d3, XltY_d4, XltY_d5 : std_logic;
9  signal YltZ, YltZ_d1, YltZ_d2, YltZ_d3, YltZ_d4, YltZ_d5 : std_logic;
10 signal XltZ, XltZ_d1, XltZ_d2, XltZ_d3, XltZ_d4, XltZ_d5 : std_logic;
11 signal EA, EA_d1, EA_d2, EA_d3, EA_d4, EA_d5, EA_d6, EA_d7, EA_d8, EA_d9, EA_d10 :
       std_logic_vector(7 downto 0);
12 signal EB, EB_d1 :  std_logic_vector(7 downto 0);
13 signal EC, EC_d1 :  std_logic_vector(7 downto 0);
14 signal fullShiftValB, fullShiftValB_d1 :  std_logic_vector(7 downto 0);
15 signal fullShiftValC, fullShiftValC_d1 :  std_logic_vector(7 downto 0);
16 signal shiftedOutB : std_logic;
17 signal shiftValB, shiftValB_d1, shiftValB_d2, shiftValB_d3, shiftValB_d4 :
       std_logic_vector(4 downto 0);
18 signal shiftedOutC : std_logic;
19 signal shiftValC, shiftValC_d1, shiftValC_d2, shiftValC_d3, shiftValC_d4 :
       std_logic_vector(4 downto 0);
20 signal mX :  std_logic_vector(23 downto 0);
21 signal mX2 :  std_logic_vector(47 downto 0);
22 signal mY :  std_logic_vector(23 downto 0);
23 signal mY2 :  std_logic_vector(47 downto 0);
24 signal mZ :  std_logic_vector(23 downto 0);
25 signal mZ2 :  std_logic_vector(47 downto 0);
26 signal X2t, X2t_d1 :  std_logic_vector(27 downto 0);
27 signal Y2t, Y2t_d1 :  std_logic_vector(27 downto 0);
28 signal Z2t, Z2t_d1 :  std_logic_vector(27 downto 0);
29 signal MA, MA_d1, MA_d2, MA_d3 :  std_logic_vector(27 downto 0);
30 signal MB, MB_d1 :  std_logic_vector(27 downto 0);
31 signal MC, MC_d1 :  std_logic_vector(27 downto 0);
32 signal shiftedB :  std_logic_vector(55 downto 0);
```

One slide on Targets

Purpose

- target-optimal architectures
- frequency-directed pipeline

Try to avoid:

```
1 if(target=="StratixII"){
2         // output some
              VHDL
3 }
4 else if(target=="Virtex4")
      {
5         // output
              other VHDL
6 }
7 else ...
```

- Model architecture details
  - LUTs:
    `target->getLUTSize()`
  - DSP blocks
    `target->getDSPWidths(x,y);`
  - memory blocks
    `target->sizeOfMemoryBlock()`
- Model delays
  `target->lutDelay()`
  `target->localWireDelay()`
  `target->adderDelay(n)`

Definitely an endless effort.

Testing

Test against the mathematical specification!

Testing

Test against the mathematical specification!

- `emulate()` performs bit-accurate emulation
  - not by architecture simulation!
  - By expressing the **mathematical specification**
  - (typically a few lines of MPFR – see `www.mpfr.org`)

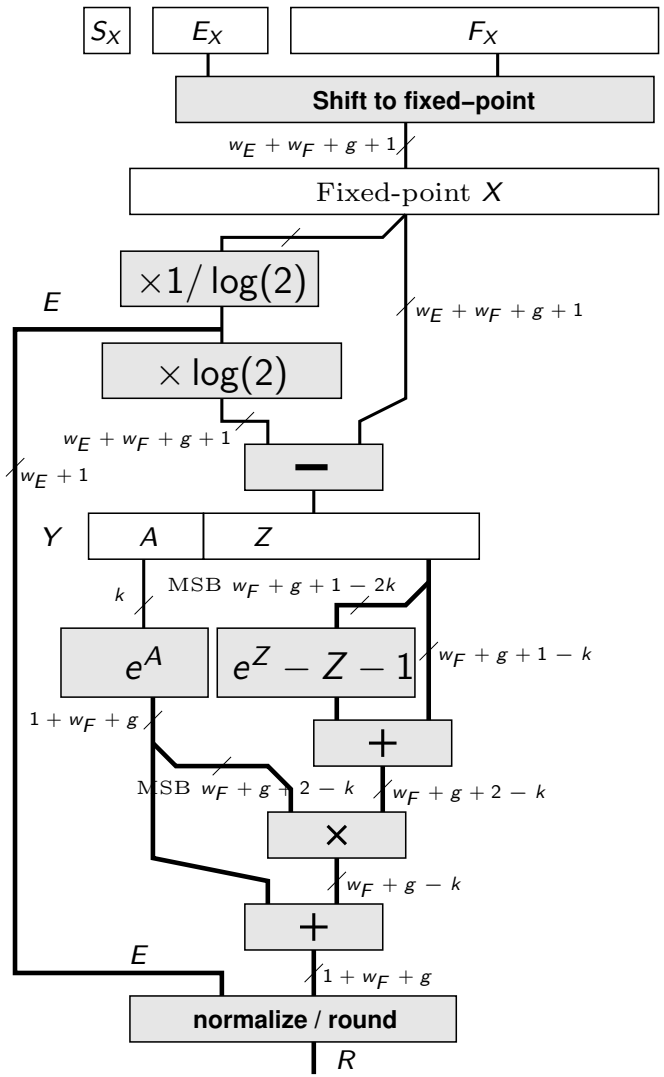
Testing

Test against the mathematical specification!

- `emulate()` performs bit-accurate emulation
  - not by architecture simulation!
  - By expressing the **mathematical specification**
  - (typically a few lines of MPFR – see `www.mpfr.org`)
- `buildStandardTestCases()`, `buildRandomTestCases()`
  - have sensible defaults
  - should be overloaded by each `Operator` in an operation-specific way
- The special `TestBench` and `TestBenchFile` operators invoke these methods

Example of emulate() for e^X

```
void FPExp::emulate(TestCase * tc){
    /* Get I/O values */
    mpz_class svX = tc->getInputValue("X");

    /* Compute correct value */
    FPNumber fpx(wE, wF);
    fpx = svX;

    mpfr_t x, ru,rd;
    mpfr_init2(x,  1+wF);
    mpfr_init2(ru, 1+wF);
    mpfr_init2(rd, 1+wF);
    fpx.getMPFR(x);
    mpfr_exp(rd, x, GMP_RNDD);
    mpfr_exp(ru, x, GMP_RNDU);
    FPNumber  fprd(wE, wF, rd);
    FPNumber  fpru(wE, wF, ru);
    mpz_class svRD = fprd.getSignalValue();
    mpz_class svRU = fpru.getSignalValue();
    tc->addExpectedOutput("R", svRD);
    tc->addExpectedOutput("R", svRU);
    mpfr_clears(x, ru, rd, NULL);
}
```



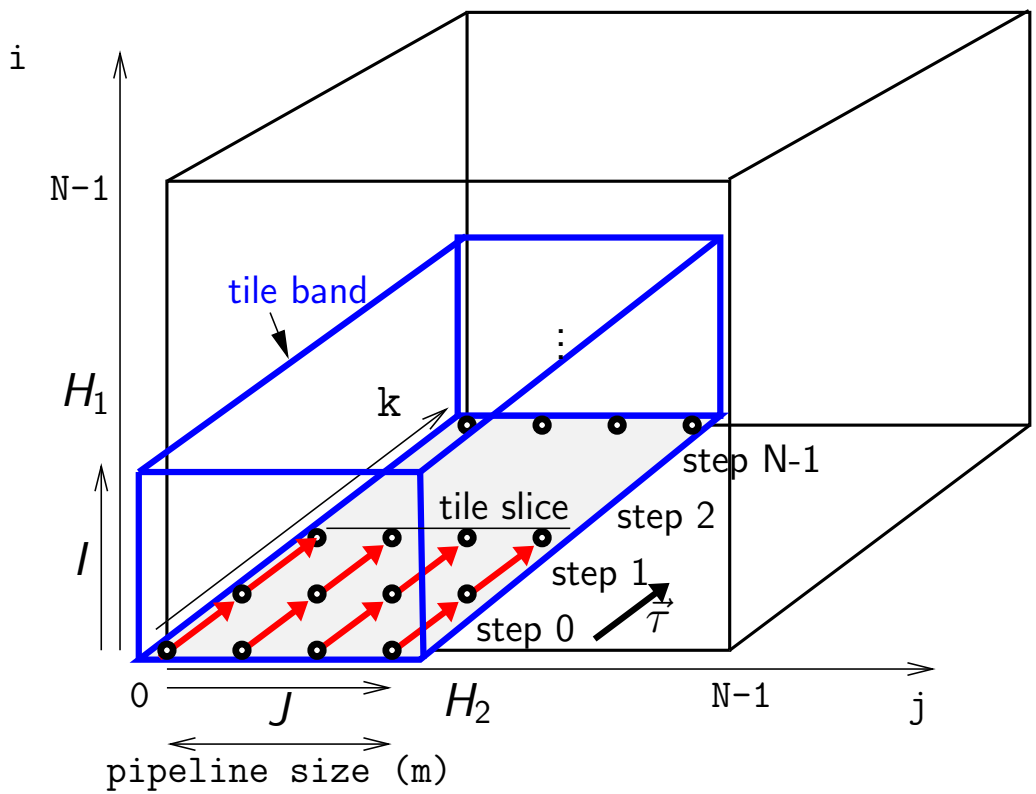
Operator-specific random test-cases

```
TestCase* FPExp::buildRandomTestCase(int i){
    TestCase *tc;
    tc = new TestCase(this);
    mpz_class x;
    mpz_class normalExn = mpz_class(1)<<(wE+wF+1);
    mpz_class bias = ((1<<(wE-1))-1);
    /* Fill inputs */
    if ((i & 7) == 0) { //fully random
       x = getLargeRandom(wE+wF+3);
    }
    else{
        mpz_class e = (getLargeRandom(wE+wF)%(wE+wF+2)) - wF-3;
        e = bias + e;
        mpz_class sign = getLargeRandom(1);
        x  = getLargeRandom(wF)
                    + (e << wF)
                    + (sign<<(wE+wF))
                    + normalExn;
    }
    tc->addInput("X", x);
    /* Get correct outputs */
    emulate(tc);
    return tc;
}
```

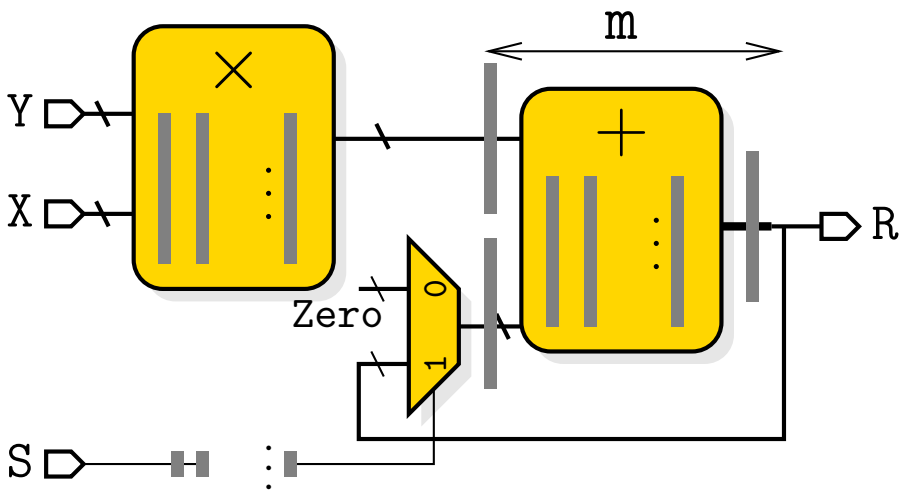
Back-end for HLS

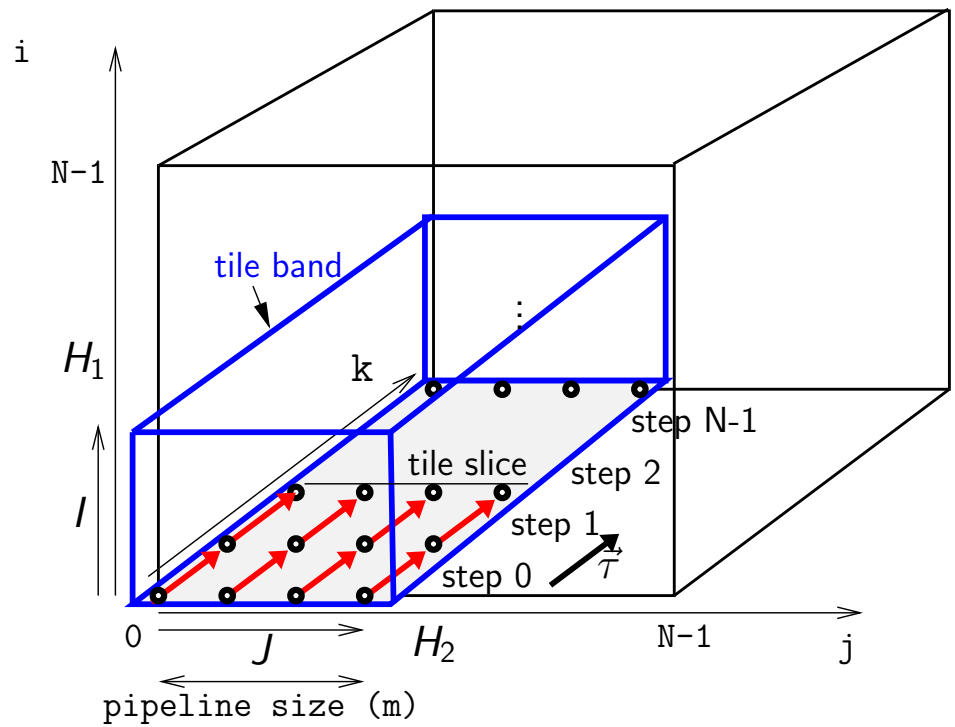
Matrix-multiply scenario

```
typedef float fl; /* basically any format */
void mmm(fl* a, fl* b, fl* c, int N) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k]*b[k][j]; <- FloPoCo Kernel
}
```



Matrix-multiply Kernel





Some results

Application	FPGA	Precision (w_E, w_F)	Latency (cycles)	Freq. (MHz)	Resources REG	LUT	DSPs
Matrix-Matrix Multiply N=128	Virtex5(-3)	(5,10)	11	277	320	526	1
		(8,23)	15	281	592	864	2
		(10,40)	14	175	978	2098	4
		(11,52)	15	150	1315	2122	8
		(15,64)	15	189	1634	4036	8
	StratixIII	(5,10)	12	276	399	549	2
		(9,36)	12	218	978	2098	4

Some results

Application	FPGA	Precision (w_E, w_F)	Latency (cycles)	Freq. (MHz)	Resources REG	LUT	DSPs
Matrix-Matrix Multiply N=128	Virtex5(-3)	(5,10)	11	277	320	526	1
		(8,23)	15	281	592	864	2
		(10,40)	14	175	978	2098	4
		(11,52)	15	150	1315	2122	8
		(15,64)	15	189	1634	4036	8
	StratixIII	(5,10)	12	276	399	549	2
		(9,36)	12	218	978	2098	4

★ efficiency: 99% for matrix-multiply, 94% for Jacobi 1D.

Conclusion

I don't like SUVs

In a Pentium

the choice is between

- an integer SUV, or
- a floating-point SUV.

I don't like SUVs

In a Pentium

the choice is between

- an integer SUV, or
- a floating-point SUV.

In an FPGA

- If all I need is a bicycle, I have the possibility to build a bicycle
- FloPoCo helps me to build the bicycle I need
- (and I'm usually faster to destination)

I don't like SUVs

In a Pentium

the choice is between

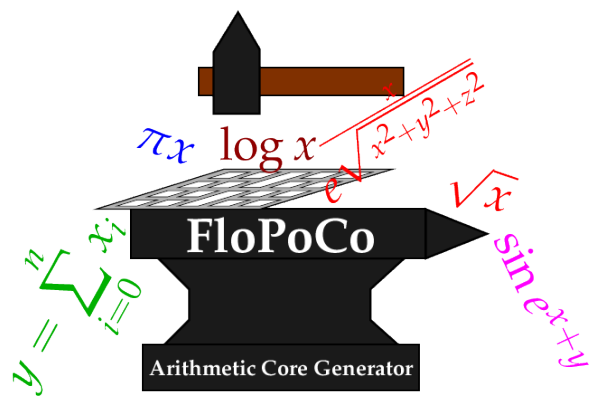
- an integer SUV, or
- a floating-point SUV.

In an FPGA

- If all I need is a bicycle, I have the possibility to build a bicycle
- FloPoCo helps me to build the bicycle I need
- (and I'm usually faster to destination)

A virgin land

Most of the arithmetic literature addresses the construction of SUVs.



Floating-Point Cores, but not only

After two years, release 2.2.0:

- 12 floating-point operators
- one meta operator (datapath compiler)
- 4 operators for the Logarithm Number System
- 16 non-trivial (pipelined) integer operators (shifters, LZC, etc)
- Two fixed-point arbitrary function generators (DEMO)

```
flopoco HOTBM "exp(x*x)" 15 15 4
flopoco FunctionEvaluator "exp(x*x)/4" 15 15 1
```

Perspectives

- Finetune target-directed optimizations
- Add an ASIC target?
- Explore using this infrastructure to assemble larger pipelines
- Endless list of operators (how about modular multipliers? ...)
- Explore direct interface to some C-to-hardware tool?

The real open question

Floating-point is for the lazy. Fixed-point is always more efficient.

Should we not work instead at assisting people in the floating- to fixed-point conversion of their applications?

Current state of the answer:

- Probably we should.
- But nobody wants that. People want floating-point!
- So we do it for large operators (e.g. the FP logarithm), but it is hidden to the user.

Thank you for your attention

F. de Dinechin and B. Pasca, **Floating-point exponential functions for DSP-enabled FPGAs.** In *FPT 2010*.

F. de Dinechin, H.D. Nguyen, B. Pasca,**Pipelined FPGA adders.** In *FPL 2010*.

F. de Dinechin, M. Joldes, B. Pasca, and G. Revy,**Multiplicative square root algorithms for FPGAs.** In *FPL 2010*.

F. de Dinechin, M. Joldes, and B. Pasca, **Automatic generation of polynomial-based hardware architectures for function evaluation.** In *ASAP 2010*.

F. de Dinechin, C. Klein and B. Pasca, **Generating high-performance custom floating-point pipelines.** In *FPL 2009*.

F. de Dinechin and B. Pasca, **Large multipliers with fewer DSP blocks.** In *FPL 2009*.

F. de Dinechin, B. Pasca, O. Creţ, and R. Tudoran, **An FPGA-specific approach to floating-point accumulation and sum-of-products**. In *FPT 2008*.

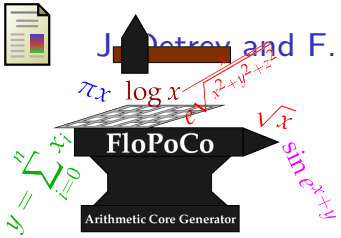
N. Brisebarre, F. de Dinechin, and J.-M. Muller, **Integer and floating-point constant multipliers for FPGAs.** In *ASAP 2008*.

J. Detrey and F. de Dinechin. **Floating-point trigonometric functions for FPGAs.** In *FPL2007*.

J. Detrey and F. de Dinechin. **Parameterized floating-point logarithm and exponential functions for FPGAs.** *Microprocessors and Microsystems*, 31(8):537–545, Jun. 2007. Elsevier.

J. Detrey, F. de Dinechin, and X. Pujol. **Return of the hardware floating-point elementary function.** In *Arith 2007*.

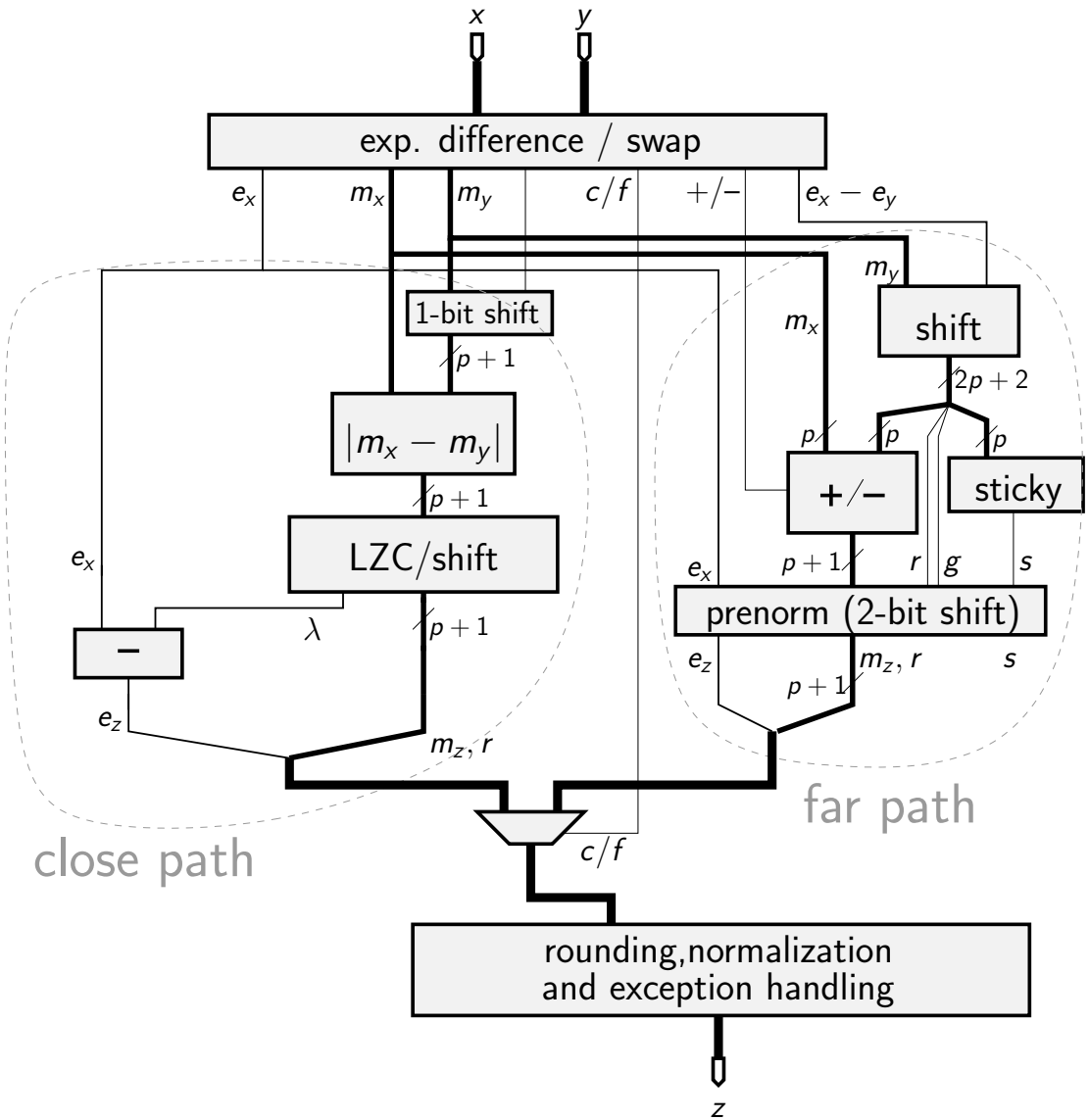
J. Detrey and F. de Dinechin, **Table-based polynomials for fast hardware function evaluation.** In *ASAP 2005*.



http://flopoco.gforge.inria.fr/

vs.

A floating-point adder



back