FPGA-specific arithmetic pipeline design using FloPoCo

Bogdan Pasca, Arénaire

CARAMEL, 17/02/2011
Outline

FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

Conclusion
FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

Conclusion
Field Programmable Gate Array

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

<table>
<thead>
<tr>
<th>Year</th>
<th>1995</th>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>XC4010</td>
<td>XC6VHX565T</td>
</tr>
<tr>
<td>Capacity (K LE)</td>
<td>1</td>
<td>500</td>
</tr>
<tr>
<td>DSPs</td>
<td>-</td>
<td>1K</td>
</tr>
<tr>
<td>Bock RAM</td>
<td>-</td>
<td>2K (18Kb)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>10</td>
<td>600</td>
</tr>
<tr>
<td>FPAdder ((w_E = 6, w_F = 9)^1)</td>
<td>28%</td>
<td>0.05%</td>
</tr>
<tr>
<td>FPMultiplier ((w_E = 6, w_F = 9))</td>
<td>44%</td>
<td>*(^2)</td>
</tr>
<tr>
<td>FPDivider ((w_E = 6, w_F = 9))</td>
<td>46%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

\(^1\)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

FPGAs are now large enough to implement complex datapaths

<table>
<thead>
<tr>
<th>Year</th>
<th>1995</th>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPGA</strong></td>
<td>XC4010</td>
<td>XC6VHX565T</td>
</tr>
<tr>
<td>Capacity (K LE)</td>
<td>1</td>
<td>500</td>
</tr>
<tr>
<td>DSPs</td>
<td>-</td>
<td>1K</td>
</tr>
<tr>
<td>Bock RAM</td>
<td>-</td>
<td>2K (18Kb)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>10</td>
<td>600</td>
</tr>
<tr>
<td><strong>FPAdder</strong> ($w_E = 6$, $w_F = 9$)</td>
<td>28%</td>
<td>0.05%</td>
</tr>
<tr>
<td><strong>FPMultiplier</strong> ($w_E = 6$, $w_F = 9$)</td>
<td>44%</td>
<td>*2</td>
</tr>
<tr>
<td><strong>FPDivider</strong> ($w_E = 6$, $w_F = 9$)</td>
<td>46%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

---

2. Multiplications are usually implemented using DSPs on modern FPGAs
So, are FPGAs any good at floating-point in 2011?

Today’s basic operations: +, −, ×

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.

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)
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 ...

---

Peak figures for double-precision floating-point exponential.

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

(Bogdan Pasca, Arénaire)
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)
Useful operators that would not be economical in a processor

- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($x \sqrt{x^2 + y^2}$, polynomials, ...)
- Compound functions ($\log_2(1 \pm 2^x), e^{-Kt}$, ...)
- 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.
Useful operators that would not be economical in a processor

- Elementary functions (sine, exponential, logarithm...)
- Algebraic functions ($\sqrt{x^2 + y^2}$, 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.
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.
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

FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

Conclusion
A question of granularity

FloPoCo

performance

productivity

low

abstraction

high

FPGA primitives

C–like arithmetic datapath

system builder

loop management

Bogdan Pasca, Arénaire

FPGA-specific arithmetic pipeline design using FloPoCo
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

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
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
The FloPoCo recipe for high productivity

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

flopoco FPPipeline expr.in 8 23

/* sum of squares */
r = sqr(x) + sqr(y) + sqr(z);
output r;
A few results for floating-point sum-of-squares on Virtex4:

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>area</td>
<td>performance</td>
</tr>
<tr>
<td>LogiCore classic</td>
<td>1282 slices, 20 DSP</td>
<td>43 cycles @ 353 MHz</td>
</tr>
<tr>
<td>FloPoCo compiler</td>
<td>1047 slices, 9 DSP</td>
<td>36 cycles @ 357 MHz</td>
</tr>
<tr>
<td>FloPoCo custom</td>
<td>453 slices, 9 DSP</td>
<td>11 cycles @ 368 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LogiCore classic</td>
<td>3942 slices, 48 DSP</td>
<td>52 cycles @ 279 MHz</td>
</tr>
<tr>
<td>FloPoCo compiler</td>
<td>3354 slices, 18 DSP</td>
<td>49 cycles @ 348 MHz</td>
</tr>
<tr>
<td>FloPoCo custom</td>
<td>1845 slices, 18 DSP</td>
<td>16 cycles @ 362 MHz</td>
</tr>
</tbody>
</table>

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

\[\text{Assembling floating-point operators}\]
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

Bogdan Pasca, Arénaire
FPGA-specific arithmetic pipeline design using FloPoCo
Inside FloPoCo

FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

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

- VHDL generation is “print-based”

```vhdl
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!
FloPoCo is **not** a C++-based HDL, but more of a **mix**

- VHDL generation is “print-based”
  ```vhdl
  vhdl << "SoS <= EA(wE-1 downto 0) & Fraction;" ;
  ```
  - easy to port existing work (FPLibrary)
  - easy learning curve for the VHDL-literate
  - at least the expressive power of VHDL!

- Many helper functions help doing the prints

  **Example:** VHDL signal declaration
  ```vhdl
  vhdl << declare("SoS", wE+wF+g)
  << " <= EA(wE-1 downto 0) & Fraction;" ;
  ```
FloPoCo class hierarchy

**Signal**
- width
- cycle
- lifeSpan

**Operator**
- signalList
- vhdl
- outputVHDL()
- emulate()
- buildStandardTestCases()

**Targets**
- StratixII
- Virtex4

**Shifters**

**IntAddder**
- size

**FPAdder**
- wE
- wF

**Collision**
- wE
- wF

**TestBench**
Pipeline made easy

\[ \begin{align*}
X & \\
Y & \\
Z & \\
\text{unpack} & \\
M_X & = 1 + w_F \\
M_Y & = 1 + w_F \\
M_Z & = 1 + w_F \\
\text{squerer} & \\
\text{squerer} & \\
\text{squerer} & \\
\text{sort} & \\
\text{sort} & \\
\text{sort} & \\
\text{shifter} & \\
\text{shifter} & \\
\text{shifter} & \\
\text{add} & \\
4 + w_F + g & \\
\text{normalize/pack} & \\
W_E + w_F + g & \\
R & 
\end{align*} \]
Notion of current cycle during VHDL output
Each signal has an active cycle
Pipeline made easy

Bogdan Pasca, Arénaire  
FPGA-specific arithmetic pipeline design using FloPoCo
Pipeline made easy

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

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

Bogdan Pasca, Arénaire  
FPGA-specific arithmetic pipeline design using FloPoCo
Pipeline made easy

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

\[
\text{SoS} \leq \text{EA}_d6(\text{wE-1 downto 0}) \& \text{Fraction}_d1;
\]

(and transparently declare and build the needed shift registers)
Managing the current cycle:
\[ n = \text{getCycle}(); \]
\[ \text{setCycle}(n); \]
\[ \text{nextCycle}(); \]
\[ \text{syncCycleWithSignal}("EA"); \]

Frequency-directed pipelining:
\[ \text{manageCriticalPath(} \]
\[ \quad \text{target->adderDelay}(n) \text{);} \]
// 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

\[ x \times a_2 \times a_1 \times a_0 + a_2 x^2 \times a_1  + a_0 \]
Multiple Path Designs

```cpp
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);
olist.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);
olist.push_back(fpm);
inPortMap (fpm, "X", "X2");
inPortMap (fpm, "Y", "a2");
outPortMap(fpm, "R", "a2x2");
vhdl << instance(fpm, "fpMultiplier_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, "fpMultiplier_a1x");

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

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

Bogdan Pasca, Arénaire

FPGA-specific arithmetic pipeline design using FloPoCo
Operator data-flow

Constructor

(recursively calling constructors of all sub-components to know their pipeline depth)

C++

structure information

produces

vhd stream
(combinatorial circuit)

signal list
(bitwidth, etc)

subcomponent list

pipeline information

cycle value for each signal

lifeSpan value for each signal

is used

Operator.outputVHDL()

lifeSpan computation
(first pass on vhdl stream)

generation of VHDL declarations

generation of VHDL code for registers

generation of VHDL architecture code
(second pass on vhdl stream, delaying right-hand side signals)

VHDL

C++
Pipeline made easy

Correct-by-construction pipelines, and more

- Conceptually simple
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
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**
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

```vhdl
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);
signal EX : std_logic_vector(7 downto 0);
signal EY : std_logic_vector(7 downto 0);
signal EZ : std_logic_vector(7 downto 0);
signal DEXY : std_logic_vector(8 downto 0);
signal DEY : std_logic_vector(8 downto 0);
signal DEXZ : std_logic_vector(8 downto 0);
signal XltY, XltY_d1, XltY_d2, XltY_d3, XltY_d4, XltY_d5 : std_logic;
signal YltZ, YltZ_d1, YltZ_d2, YltZ_d3, YltZ_d4, YltZ_d5 : std_logic;
signal XltZ, XltZ_d1, XltZ_d2, XltZ_d3, XltZ_d4, XltZ_d5 : std_logic;
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);
signal EB, EB_d1 : std_logic_vector(7 downto 0);
signal EC, EC_d1 : std_logic_vector(7 downto 0);
signal fullShiftValB, fullShiftValB_d1 : std_logic_vector(7 downto 0);
signal fullShiftValC, fullShiftValC_d1 : std_logic_vector(7 downto 0);
signal shiftedOutB : std_logic;
signal shiftValB, shiftValB_d1, shiftValB_d2, shiftValB_d3, shiftValB_d4 :
    std_logic_vector(4 downto 0);
signal shiftedOutC : std_logic;
signal shiftValC, shiftValC_d1, shiftValC_d2, shiftValC_d3, shiftValC_d4 :
    std_logic_vector(4 downto 0);
signal mX : std_logic_vector(23 downto 0);
signal mY2 : std_logic_vector(47 downto 0);
signal mY : std_logic_vector(23 downto 0);
signal mZ2 : std_logic_vector(47 downto 0);
signal mZ : std_logic_vector(23 downto 0);
signal mZ2 : std_logic_vector(47 downto 0);
signal X2t, X2t_d1 : std_logic_vector(27 downto 0);
signal Y2t, Y2t_d1 : std_logic_vector(27 downto 0);
signal Z2t, Z2t_d1 : std_logic_vector(27 downto 0);
signal MA, MA_d1, MA_d2, MA_d3 : std_logic_vector(27 downto 0);
signal MB, MB_d1 : std_logic_vector(27 downto 0);
signal MC, MC_d1 : std_logic_vector(27 downto 0);
signal shiftedB : std_logic_vector(55 downto 0);
```
One slide on Targets

Purpose

- target-optimal architectures
- frequency-directed pipeline

Try to avoid:

```cpp
if(target == "StratixII"){
    // output some VHDL
}
else if(target == "Virtex4"){
    // output other VHDL
}
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.
Test against the mathematical specification!
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](http://www.mpfr.org))
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 \( e^x \)**

```cpp
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

```c
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

FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

Conclusion
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 */
}

Bogdan Pasca, Arénaire
FPGA-specific arithmetic pipeline design using FloPoCo
Matrix-multiply Kernel

\[ Y \times X \]

Zero

\[ \text{pipeline size (m)} \]

\[ \overset{H_2}{\rightarrow} \]

\[ \overset{H_1}{\rightarrow} \]

\[ \overset{I}{\rightarrow} \]

\[ \overset{0}{\rightarrow} \]

\[ \overset{j}{\rightarrow} \]

\[ \overset{N-1}{\rightarrow} \]

\[ \overset{i}{\rightarrow} \]

\[ \overset{\text{tile band}}{\rightarrow} \]

\[ \overset{\text{tile slice}}{\rightarrow} \]

\[ \overset{\text{step N-1}}{\rightarrow} \]

\[ \overset{\text{step 2}}{\rightarrow} \]

\[ \overset{\text{step 1}}{\rightarrow} \]

\[ \overset{\text{step 0}}{\rightarrow} \]
### Some results

<table>
<thead>
<tr>
<th>Application</th>
<th>FPGA</th>
<th>Precision ((w_E, w_F))</th>
<th>Latency (cycles)</th>
<th>Freq. (MHz)</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>REG</td>
<td>LUT</td>
</tr>
<tr>
<td>Matrix-Matrix Multiply</td>
<td>Virtex5(-3)</td>
<td>(5,10)</td>
<td>11</td>
<td>277</td>
<td>320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(8,23)</td>
<td>15</td>
<td>281</td>
<td>592</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(10,40)</td>
<td>14</td>
<td>175</td>
<td>978</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11,52)</td>
<td>15</td>
<td>150</td>
<td>1315</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(15,64)</td>
<td>15</td>
<td>189</td>
<td>1634</td>
</tr>
<tr>
<td>N=128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>StratixIII</td>
<td>(5,10)</td>
<td>12</td>
<td>276</td>
<td>399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(9,36)</td>
<td>12</td>
<td>218</td>
<td>978</td>
</tr>
</tbody>
</table>
## Some results

<table>
<thead>
<tr>
<th>Application</th>
<th>FPGA</th>
<th>Precision ( (w_E, w_F) )</th>
<th>Latency (cycles)</th>
<th>Freq. (MHz)</th>
<th>Resources</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Matrix Multiply</td>
<td>Virtex5(-3)</td>
<td>( (5,10) )</td>
<td>11</td>
<td>277</td>
<td>320</td>
<td>526</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (8,23) )</td>
<td>15</td>
<td>281</td>
<td>592</td>
<td>864</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (10,40) )</td>
<td>14</td>
<td>175</td>
<td>978</td>
<td>2098</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (11,52) )</td>
<td>15</td>
<td>150</td>
<td>1315</td>
<td>2122</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (15,64) )</td>
<td>15</td>
<td>189</td>
<td>1634</td>
<td>4036</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>N=128</td>
<td>Strati×III</td>
<td>( (5,10) )</td>
<td>12</td>
<td>276</td>
<td>399</td>
<td>549</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( (9,36) )</td>
<td>12</td>
<td>218</td>
<td>978</td>
<td>2098</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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

FPGAs and floating-point

Datapath design using FloPoCo

Inside FloPoCo

Back-end for HLS

Conclusion
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.
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?
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. Creț, 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 *FPL 2007.*


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

\[ \lambda \]

exp. difference / swap

\[ e_x \quad m_x \quad m_y \quad c/f \quad +/- \quad e_x - e_y \]

1-bit shift

\[ \rho + 1 \]

\[ |m_x - m_y| \]

\[ e_x \quad p + 1 \]

LZC/shift

\[ e_z \]

\[ m_z, r \]

\[ c/f \]

shift

\[ 2p + 2 \]

\[ m_y \]

\[ e_x \]

\[ p + 1 \]

\[ m_z \]

\[ e_z \]

\[ p + 1 \]

prenorm (2-bit shift)

\[ m_z, r \]

far path

close path

rounding, normalization and exception handling

\[ z \]