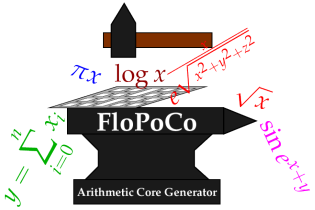
Multipliers and Square Root for FloPoCo

ANR EvaFlo Reunion 23-23 September 2009

Florent de Dinechin and Bogdan Pasca
projet Arénaire, LIP,
ENS-Lyon/CNRS/INRIA/Université de Lyon

Flopoco [1] – what, why, how ?



Not your neighbor's FPU

[1] Published at FPL09

What is FloPoCo ?

1. **Generator** of operators for FPGAs
2. **Framework** for developing arithmetic operators

- written in C++
- generates portable synthesizable VHDL
- open source
- now at version **1.15**

`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

Why FloPoCo?

VHDL/Verilog libraries are obsolete !

- too bulky
- inflexible pipelines
- high complexity code (”slippery when wet”)
- no real design space exploration

The only way to keep FPGAs in the FP cards !

- basic FP: FPGA faster than PC, but no match to GPGPU, Cell, ...

How?

Explore Flexibility

- mix and match FP and fixed-point
- generate economical operators for target frequency
- implement exotic arithmetic operators no available in processors

The tool for you is FloPoCo

- development/generator framework
- automatic pipeline synchronization infrastructure
- automatic test-bench generator
- fast synthesis scripts for Altera and Xilinx
- regression test-script

Large multipliers using fewer DSP blocks[2]

[2]Published at FPL'09

Large multipliers using embedded multipliers

"Large" - multiplier that consumes ≥ 2 embedded multipliers

Large multipliers using embedded multipliers

Let:

k - an **integer parameter**

X,Y - $2k$-bit **integers to multiply**.

Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - 2k-bit **integers to multiply**.

$$\begin{array}{lcccccccc} X & & 1 & 0 & 1 & 1 & 0 & 1 & \\ Y & & 1 & 1 & 0 & 1 & 0 & 0 & \times \\ \hline \end{array}$$

Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - 2k-bit **integers to multiply**.

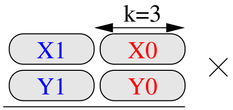
$$\begin{array}{lcccccc} & & & & \multicolumn{3}{c}{\overset{k=3}{\longleftrightarrow}} \\ X & 1 & 0 & 1 & 1 & 0 & 1 \\ Y & 1 & 1 & 0 & 1 & 0 & 0 \\ \hline \end{array} \times$$

Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - 2k-bit **integers to multiply**.

$$X = 2^k X_1 + X_0$$
$$Y = 2^k Y_1 + Y_0$$



Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - 2k-bit **integers to multiply**.

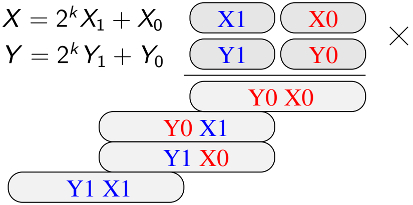
$$\begin{array}{l}
X = 2^k X_1 + X_0 \\
Y = 2^k Y_1 + Y_0
\end{array}
\qquad
\begin{array}{cccccccccccc}
 & & & & & & \boxed{1\ 0\ 1} & & & \boxed{1\ 0\ 1} & & \\
 & & & & & & \boxed{1\ 1\ 0} & & & \boxed{1\ 0\ 0} & & \times \\
\hline
 & & & & & & 0 & 1 & 0 & 1 & 0 & 0 \\
 & & & 0 & 1 & 0 & 1 & 0 & 0 & & & \\
 & & & 0 & 1 & 1 & 1 & 1 & 0 & & & \\
0 & 1 & 1 & 1 & 1 & 0 & & & & & &
\end{array}$$

Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - 2k-bit **integers to multiply**.

$X = 2^k X_1 + X_0$
$Y = 2^k Y_1 + Y_0$



$$Y0X0 \\ +2^k Y0X1 \\ +2^k Y1X0 \\ +2^{2k} Y1X1$$

If **k=embedded multiplier width** then
need **4** embedded multipliers for 2k-bit multiplication

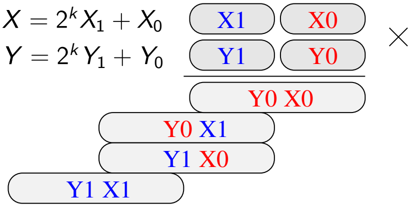
Large multipliers using embedded multipliers

Let:

k - an **integer parameter**
X,Y - $2k$-bit **integers to multiply**.

$X = 2^k X_1 + X_0$

$Y = 2^k Y_1 + Y_0$



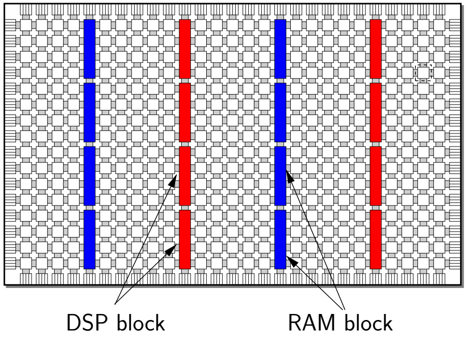
$$\begin{aligned} &Y0X0 \\ +&2^k Y0X1 \\ +&2^k Y1X0 \\ +&2^{2k} Y1X1 \end{aligned}$$

If **k=embedded multiplier width** then
need **4** embedded multipliers for 2k-bit multiplication

Generalization

$\forall\ p > 1$, numbers of size $p(k-1)+1$ to pk can be decomposed into p k-bit numbers $\Rightarrow$ architecture consuming p^2 embedded multipliers.

Today's FPGAs



- Small and fast memory blocks (Kbits)
  - example (Virtex4) : configurable $2^{16} \times 1$ to $2^{9} \times 36$ bits
- DSP blocks
  - 1 to 8 small multipliers (9x9, 18x18, 36x36 bits)
  - add/accumulate units
  - cascade possibility

The premise

DSP-blocks are a scarce resource when accelerating double precision floating-point applications [3]

we give

Three recipes for saving DSPs

[3]D. Strenski, FPGA floating point performance – a pencil and paper evaluation. HPCWire, Jan. 2007.

Karatsuba-Ofman algorithm

trading multiplications for additions

Prior work by Beuchat/Tisserand for Virtex II (Arenaire)

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

The Karatsuba-Ofman algorithm

Basic principle for two way splitting

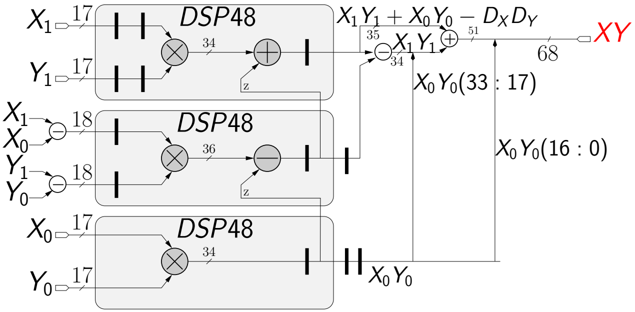
- split X and Y into two chunks:

$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- computation goal: $XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_0 + X_0 Y_1) + X_0 Y_0$
- precompute $D_X = X_1 - X_0$ and $D_Y = Y_1 - Y_0$
- make the observation: $X_1 Y_0 + X_0 Y_1 = X_1 Y_1 + X_0 Y_0 - D_X D_Y$
- XY requires only 3 DSP blocks $(X_1 Y_1, X_0 Y_0, D_X D_Y)$
- overhead: two k-bit and one $2k$-bit subtraction
- overhead $\ll$ DSP-block emulation

Implementation – 34x34bit multiplier on Virtex-4

$$XY = 2^{34}X_1Y_1 + 2^{17}(X_1Y_1 + X_0Y_0 - D_XD_Y) + X_0Y_0$$



- take advantage of DSP48 by cascading
- $X_1Y_1 + X_0Y_0 - D_XD_Y$ is implemented inside the DSPs
- need to recover X_1Y_1 with a subtraction

Results - 34x34bit multiplier on Virtex-4

	latency	freq.	slices	DSPs
LogiCore	6	447	26	4
LogiCore	3	176	34	4
K-O-2	3	317	95	**3**

Remarks

- trade-off one DSP-block for 69 slices*
- *frequency bottleneck of 317MHz caused by SRL16
- larger frequency with more slices (disable shift register extraction)

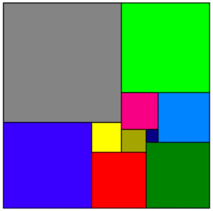
Results - 51x51bit multiplier on Virtex-4

	latency	freq.	slices	DSPs
LogiCore	11	353	185	9
LogiCore	6	264	122	9
K-O-3	6	317	331	**6**

Remarks:

- reduced DSP usage from 9 to 6
- overhead of $6k$ LUTs for the pre-subtractions
- overhead of the remaining additions difficult to evaluate (most may be implemented inside DSP blocks)

Non-standard tilings



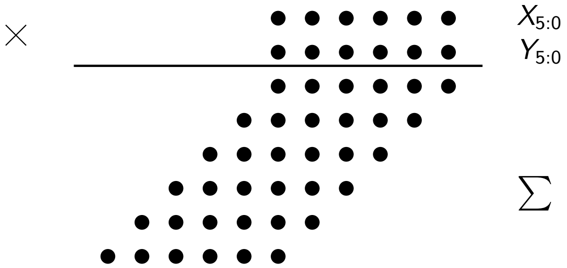
new multiplier family

From multiplication to tiling

- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

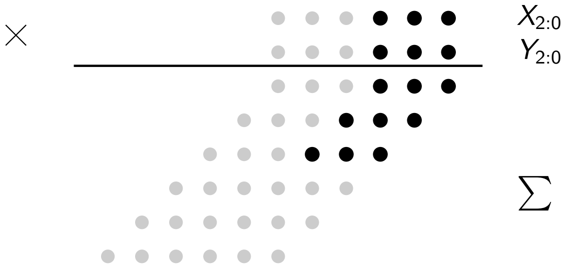
From multiplication to tiling



- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

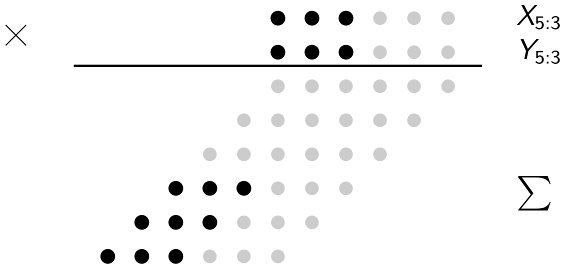
From multiplication to tiling



- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

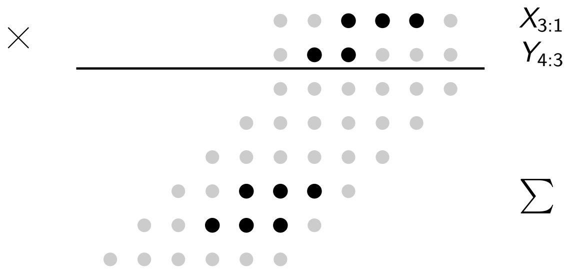
From multiplication to tiling



- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

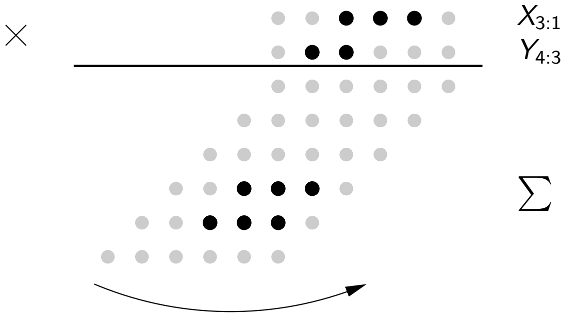
From multiplication to tiling



- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

From multiplication to tiling

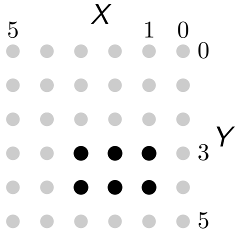


- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

From multiplication to tiling

$2^{3+1}X_{3:1}Y_{4:3}$

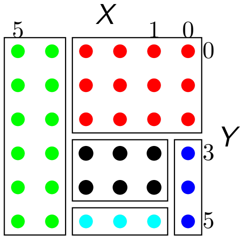


- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

From multiplication to tiling

$$\begin{aligned} XY =\ & 2^{3+1} X_{3:1} Y_{4:3} \\ & + 2^4 X_{5:4} Y_{5:0} \\ & + X_{3:0} Y_{2:0} \\ & + 2^3 X_0 Y_{5:3} \\ & + 2^{1+5} X_{3:1} Y_5 \end{aligned}$$



- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

$$tile\_contribution = 2^{upper\_right\_cornerX+Y} X_{projection} Y_{projection}$$

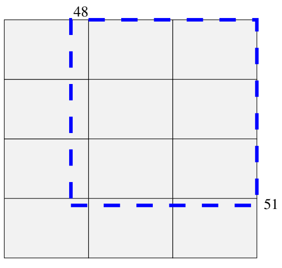
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5,6 (25x18 signed)
- classical decomposition may produce suboptimal results
- translate the operand decomposition into a tiling problem

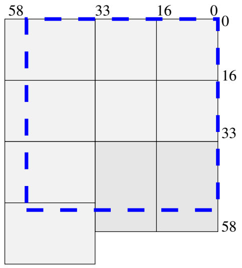
Tiling principle

- start-off with a **rectangle** of size $X.width \times Y.width$
- and **tiles** of size $P \times Q$ where:
  - $P \leq embeddedMultiplier.width1$ and $(P \leq 24)$
  - $Q \leq embeddedMultiplier.width2$ $(Q \leq 17)$
- **place tiles** so to fill-up the initial rectangle
- directly **translate** the placement **into an architecture**
- decide which multiplications are performed in LUTs

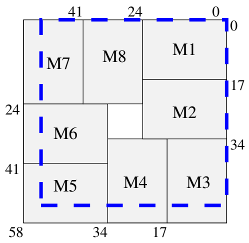
Tilings – 53 × 53-bit multiplication on Virtex5



(a) standard tiling



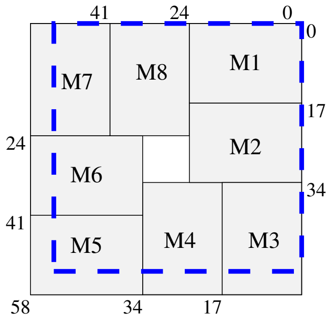
(b) Logicore tiling



(c) proposed tiling

- standard tiling ≡ classical decomposition (12 DSPs)
- Logicore 11.1 tiling uses 10 DSPs (4 DSPs used as 17x17-bit)
- **our proposed tiling** does it in **8 DSPs** and a few LUTs

Tiling Architecture - 53x53bit



$$
\begin{array}{rcl|l}
XY & = & X_{0:23}Y_{0:16} & (M1) \\
 & + & 2^{17}(X_{0:23}Y_{17:33} & (M2) \\
 & + & 2^{17}(X_{0:16}Y_{34:57} & (M3) \\
 & + & 2^{17}X_{17:33}Y_{34:57})) & (M4) \\
 & + & 2^{24}(X_{24:40}Y_{0:23} & (M8) \\
 & + & 2^{17}(X_{41:57}Y_{0:23} & (M7) \\
 & + & 2^{17}(X_{34:57}Y_{24:40} & (M6) \\
 & + & 2^{17}X_{34:57}Y_{41:57}))) & (M5) \\
 & + & 2^{48}X_{24:33}Y_{24:33} &
\end{array}
$$

- $X_{24:33}Y_{24:33}$ (10x10 multiplier) probably best implemented in LUTs.
- parenthesis makes best use of DSP48E internal adders (17-bit shifts)

Tiling Results

58x58 multipliers on Virtex-5 (5vlx50ff676-3)[4]

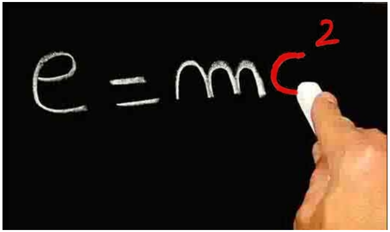
	latency	Freq.	REGs	LUTs	DSPs
LogiCore	14	440	300	249	10
LogiCore	8	338	208	133	10
LogiCore	4	95	208	17	10
Tiling	4	366	247	388	8

Remarks

- save 2 DSP48E for a few LUTs/REGs
- huge latency save at a comparable frequency
- good use of internal adders due to the 17-bit shifts

[4] Results for 53-bits are almost identical

Squarers



simple methods to save resources

Squarers

- appear in norms, statistical computations, polynomial evaluation...
- dedicated squarer saves as many DSP blocks as the Karatsuba-Ofman algorithm, but without its overhead*.

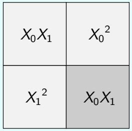
Squarers

- appear in norms, statistical computations, polynomial evaluation...
- dedicated squarer saves as many DSP blocks as the Karatsuba-Ofman algorithm, but without its overhead*.

Squaring with $k = 17$ on a Virtex-4

$\leq 34 - bit$

$$(2^k X_1 + X_0)^2 = 2^{2k} X_1^2 + 2 \cdot 2^k X_1 X_0 + X_0^2$$



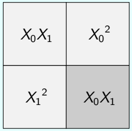
Squarers

- appear in norms, statistical computations, polynomial evaluation...
- dedicated squarer saves as many DSP blocks as the Karatsuba-Ofman algorithm, but without its overhead*.

Squaring with $k = 17$ on a Virtex-4

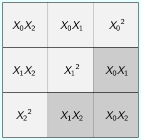
$\leq 34 - bit$

$$(2^k X_1 + X_0)^2 = 2^{2k} X_1^2 + 2 \cdot 2^k X_1 X_0 + X_0^2$$



$\leq 51 bit$

$$\begin{aligned}(2^{2k} X_2 + 2^k X_1 + X_0)^2 &= 2^{4k} X_2^2 + 2^{2k} X_1^2 + X_0^2 \\ &+ 2 \cdot 2^{3k} X_2 X_1 \\ &+ 2 \cdot 2^{2k} X_2 X_0 \\ &+ 2 \cdot 2^k X_1 X_0\end{aligned}$$



*However ...

$$(2^k X_1 + X_0)^2 = 2^{34} X_1^2 + 2^{18} X_1 X_0 + X_0^2$$

- shifts of 0, 18, 34 the previous equation
- shifts of 0, 18, 34, 35, 52, 68 for 3-way splitting
- the DSP48 of VirtexIV allow shifts of 17 so internal adders unused

*However ...

$$(2^k X_1 + X_0)^2 = 2^{34} X_1^2 + 2^{18} X_1 X_0 + X_0^2$$

- shifts of 0, 18, 34 the previous equation
- shifts of 0, 18, 34, 35, 52, 68 for 3-way splitting
- the DSP48 of VirtexIV allow shifts of 17 so internal adders unused

Workaround for $\leq$ 33-bit multiplications[a]

[a]same trick works for ≤ 50

- rewrite equation:

$$(2^{17} X_1 + X_0)^2 = 2^{34} X_1^2 + 2^{17} (2X_1) X_0 + X_0^2$$

- compute $2X_1$ by shifting X_1 by one bit before inputing into DSP48 block

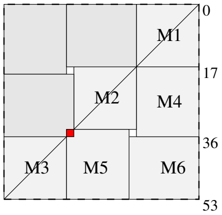
Results – 32-bit and 53-bit squarers on Virtex-4

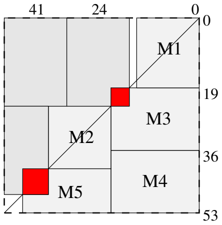
	latency	frequency	slices	DSPs	bits
LogiCore	6	489	59	4	**32**
LogiCore	3	176	34	4	
Squarer	3	317	18	**3**	
LogiCore	18	380	279	16	**53**
LogiCore	7	176	207	16	
Squarer	7	317	332	**6**	

- DSPs saved without much overhead
- impressive **10 DSPs saved** for double precision squarer

Squarers on Virtex5 using tilings

- the tiling technique can be extended to squaring
- squarer architectures for 53x53-bit



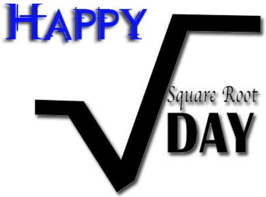


Issues

- red squares are computed twice thus need be subtracted.
- thanks to symmetry diagonal squares of size n should consume only $n(n+1)/2$ LUTs instead of n^2 .
- no implementation results … yet

Multiplicative Square-Root[5]

(joint work with Mioara Joldes and Guillaume Revy)



an unofficial holiday

1/1/01, 2/2/04, 3/3/09 ...

[5]Published at Sympa'13

Question of the day

Remember the DSPs and RAMs ?
For computing $\sqrt{x}$, most libraries don't use them

or sometimes we have plenty ...

How to make good use of them?

Algorithms for computing the square root

Two classes of algorithms:

- Digit recurrence (Pentium processors)
  - Basic operation: addition
  - Convergence: linear
- Newton/Raphson iterations (AMD, PowerPC, Itanium)
  - Basic operation: multiplication
  - Convergence: quadratic
- Piecewise polynomial approximation:
  - (unclear boundary with previous method)

Square-root using digit recurrence

- We will compute $S_j = \sum_{i=1}^{j} s_i \beta^{-i}$
- will have $S = S_n$
- we select $R_j = \beta^j (X - S_j^2)$
- The recurrence :

```
1: $R_0 = X - 1$
2: for $j \in \{1..n\}$ do
3:     $s_{j+1} = \mathrm{Sel}(\beta R_j, S_j)$    ("we guess" $s_{j+1}$)
4:     $R_{j+1} = \beta R_j - 2 s_{j+1} S_j - s_{j+1}^2 \beta^{-j-1}$
5: end for
```

Two remarks

- The blue term starts-off small and grows
- The correct rounding will be computed from the last R_j

The Matula apporach

The recurrence :

1: $R_{j+1} = \beta R_j - 2s_{j+1}S_j - s_{j+1}^2 \beta^{-j-1}$

- $\beta = 2^{17}$
- still to explore

Thank you Marc Daumas !

Polynomial approach

We want to compute the square root of a normalized FP number x

$$x = 2^e \times 1,f$$

If e is even, the square root is

$$\sqrt{x} = 2^{e/2} \times \sqrt{1,f}$$

else if e is odd, the square root is

$$\sqrt{x} = 2^{(e-1)/2} \times \sqrt{2 \times 1,f}$$

The problem is reduced at computing $\sqrt{z}$ for $z \in [1,4[$.

Piecewise polynomial approximation

First thought: cut $[1, 4[$ in pieces, obtain polynomials with Sollya
Drawbacks:

- length of $[1, 4[$ is 3, 25% of table is unused
- polynomials used on the left side of $[1, 4[$ are less precise

Solution:

- distinguish the two cases: odd and even exponent
- for **even** case ($z \in [1, 2]$) the chunks are two time smaller

The details – even branch

If e is even let:

$$\tau_{\text{even}}(x) = \sqrt{1+x}, x \in [0,1).$$

Split $[0,1[$ into 2^{k-1} intervals:

$$\left[\frac{i}{2^{k-1}}, \frac{i+1}{2^{k-1}}\right[, i \in 0...2^{k-1}-1$$

We approximate each $\tau_{\text{even}}(\frac{i}{2^{k-1}}+y)$ with:

$$p_i(y) = c_{0,i} + c_{1,i}y + \cdots + c_{d,i}y^d$$

such that:

$$|\tau_{\text{even}}(\frac{i}{2^{k-1}}+y) - p_i(y)| \leq 2^{-wF-2} \forall y \in [0, 1/2^{k-1}[.$$

The details – odd branch

If e is odd let:

$$\tau_{\text{odd}}(x) = \sqrt{2 + x}, x \in [0, 2).$$

Split $[0, 2[$ into 2^{k-1} intervals:

$$\left[\frac{i}{2^{k-2}}, \frac{i+1}{2^{k-2}}\right[, i \in 0 \ldots 2^{k-1} - 1$$

We approximate each $\tau_{\text{odd}}(\frac{i}{2^{k-2}} + y)$ with:

$$p_i(y) = c_{0,i} + c_{1,i}y + \cdots + c_{d,i}y^d$$

such that:

$$|\tau_{\text{odd}}(\frac{i}{2^{k-2}} + y) - p_i(y)| \leq 2^{-wF-2} \forall y \in [0, 1/2^{k-1}[.$$

Building the reduced argument

- **even case**: we have

$$\begin{aligned} z &= 1, f_{-1}...f_{-w_F} \\ &= 1 + 0, f_{-1}...f_{-k+1} + 2^{-k+1} 0, f_{-k}...f_{-w_F} \end{aligned}$$

- **odd case**: we have

$$\begin{aligned} z &= 2 \times 1, f \\ &= 1 f_{-1}, f_{-2}...f_{-w_F} \\ &= 2 + f_{-1}, f_{-2}...f_{-k+1} + 2^{-k+2} 0, f_{-k}...f_{-w_F} \end{aligned}$$

And we have a fix-point polynomial evaluation architecture, so:
The reduced argument is (by aligning the dot):

- **even case**: $y = 2^{-k+2} \times 0, 0 f_{-k}...f_{-w_F}$
- **odd case**: $y = 2^{-k+2} \times 0, f_{-k}...f_{-w_F} 0$

Faithful rounding

Denote by:

- $\tau(y)$ exact value of square root
- $p(y)$ approximation polynomial

Using **Sollya's** `fpminimax` we obtain:

$$\varepsilon_{\text{approx}} = |\tau(y) - p(y)| < 2^{-wF-2}.$$

Consider r – the value computed by the architecture before final rounding (not the same as $p(y)$ – we truncate at multiplier inputs)
We use **Gappa** to verify that:

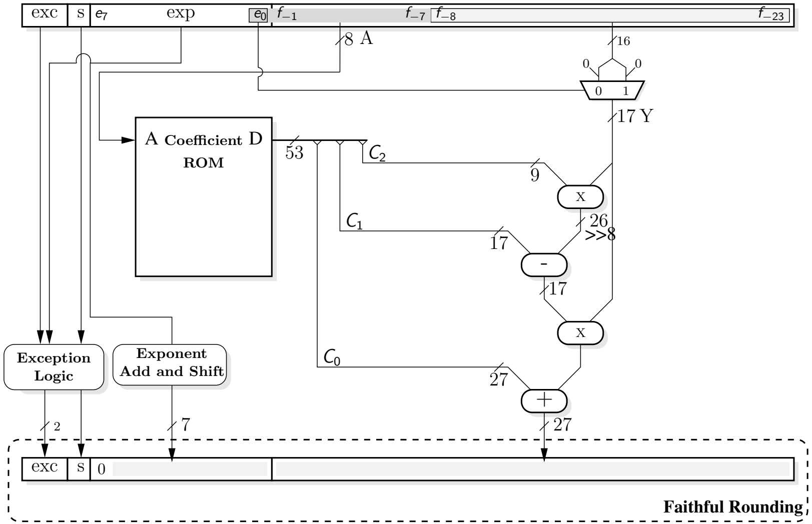
$$\varepsilon_{\text{trunc}} = |r - p(y)| < 2^{-wF-2}.$$

The **final truncation** causes an error of:

$$\varepsilon_{\text{final}} =< 2^{-wF-1}.$$

The sum of errors is smaller than $2^{-w_F} \rightarrow$ faithful rounding

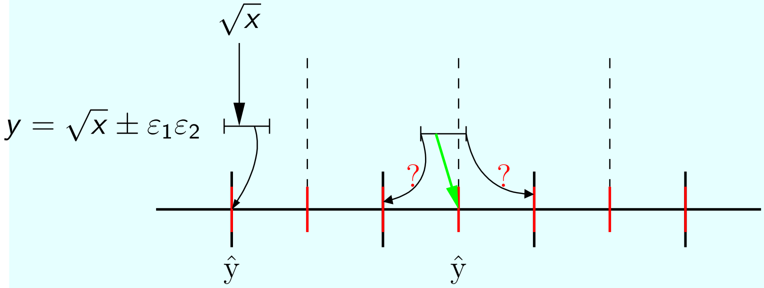
A glance at the architecture



Correct rounding

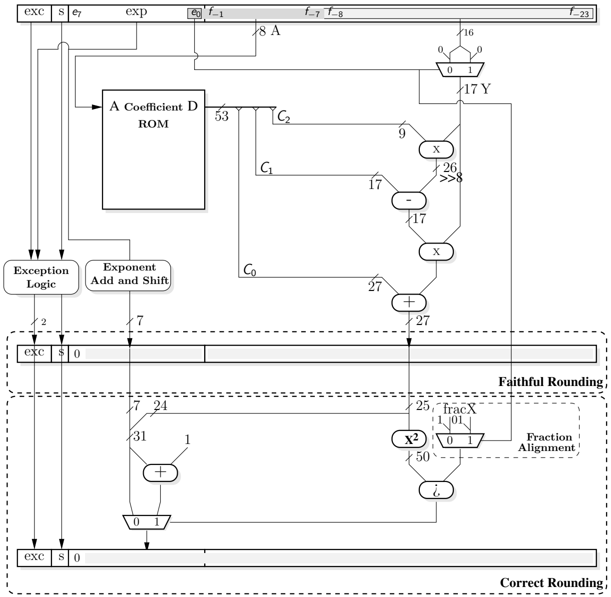
The technique of Jeannerod and Revy

In order to have the correct rounding of the square root on p bits, is enough to have a faithful rounding on $p+1$ bits



- have to round to p bits. Should we round up or down ?
- to decide we square $\hat{y}$ and compare it with x

A bird's eye view of the architecture



Other multiplicative approaches (Newton/Raphson)

Recurrence to compute $1/\sqrt{X}$:

$$Y_{n+1} = Y_n \times (3 - X \times Y_n^2)/2.$$

- 3 multiplications
- use table to initialize
- to get $\sqrt{X}$ still need to multiply by X
- here also correct rounding costs extra
- rare implementations
  we will soon humiliate them

Simple precision (32bit FP)

tool	precision	performance	cost (mult,mem)
CoreGen	0.5 ulp	28 cycles @ 353 MHz	464 sl.
FPLibrary	0.5 ulp	15 cycles @ 219 MHz	345 sl.
SRT@350MHz	0.5 ulp	26 cycles @ 353 MHz	412 sl.
SRT@200MHz	0.5 ulp	12 cycles @ 219 MHz	328 sl.
VFLOAT	$>$ 2 ulp	9 cycles @ $>$300 MHz	351 sl., (9, 3)
Poly_Faithful	1 ulp	5 cycles @ 339 MHz	79 sl., (2, 2)
Poly_Correct	0.5 ulp	12 cycles @ 237 MHz	241 sl., (5, 2)
Altera ($1/\sqrt{x}$)	?	19 cycles @ ?	350 ALM, (11, ?)

- polynomial approach gains latency and slices

Double precision (64bit FP)

tool	precision	performance	cost (mult,mem)
CoreGen	0.5 ulp	57 cycles @ 334 MHz	2061 sl.
FPLibrary	0.5 ulp	29 cycles @ 148 MHz	1352 sl.
SRT@300MHz	0.5 ulp	53 cycles @ 307 MHz	1740 sl.
SRT@200MHz	0.5 ulp	40 cycles @ 206 MHz	1617 sl.
VFLOAT	$>$ 2 ulp	17 cycles @ $>$200 MHz	1572 sl., (24, 116)
Poly_Faithful	1 ulp	*25 cycles @ 340 MHz*	*2700 sl.*, (24, 20)
Altera ($1/\sqrt{x}$)	?	32 cycles @ ?	900 ALM, (27, ?)

- multiplicative approaches less and less convincing
- we didn't even try for correct rounding

Conclusions

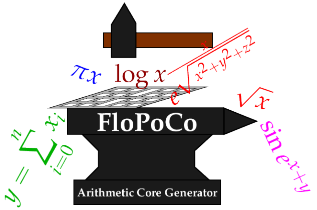
On one hand,

- DSP resources can be saved by exploiting the flexibility of the FPGA target
- flexible small granularity multipliers give best results for this techniques
- the place for this algorithms is in vendor tools

On the other hand, we are rather surprised:

- it seems difficult to effectively use DPSs to compute DP $\sqrt{x}$
- in SP, it works because the multiplications hold in 18bits

Try FloPoCo !



`http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`