Large multipliers with fewer DSP blocks

FPL09

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

Outline

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**.

X	1	0	1	1	0	1	$\times$
Y	1	1	0	1	0	0	

Large multipliers using embedded multipliers

Let:

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

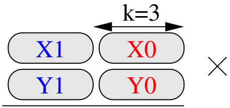
	k=3 (low 3 bits)						
X	1	0	1	1	0	1	$\times$
Y	1	1	0	1	0	0	

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**.

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

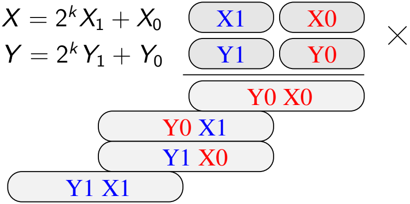
$$\begin{array}{cccccccccccc}
 & & & & & & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & & & & 1 & 1 & 0 & 1 & 0 & 0 \\
\hline
 & & & & & & 0 & 1 & 0 & 1 & 0 & 0 \\
 & & & 0 & 1 & 0 & 1 & 0 & 0 & & & \\
 & & & 0 & 1 & 1 & 1 & 1 & 0 & & & \\
0 & 1 & 1 & 1 & 1 & 0 & & & & & &
\end{array} \times$$

Large multipliers using embedded multipliers

Let:

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

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



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

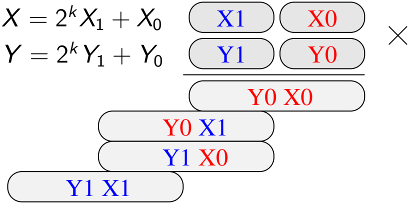
If **k=embedded multiplier width** then
need **4** embedded multipliers for 2k-bit multiplication

Large multipliers using embedded multipliers

Let:

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

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



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

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.

Embedded multipliers and more ...

Xilinx - DSP-block evolution

VirtexII-Pro

- starts-off as a 18×18 signed multiplier (k=17)

VirtexIV – DSP48

- multiplier followed by a 48-bit adder/subtracter unit
- adder/subtracter inputs are cascadable
- optional registers present

VirtexV – DSP48E

- asymmetrical multiplier of 18×25 signed
- 3-operand adder/subtracter, one input coming from global routing

Embedded multipliers and more ...

Xilinx - DSP-block evolution

VirtexII-Pro

- starts-off as a 18×18 signed multiplier (k=17)

VirtexIV – DSP48

- multiplier followed by a 48-bit adder/subtracter unit
- adder/subtracter inputs are cascadable
- optional registers present

VirtexV – DSP48E

- asymmetrical multiplier of 18×25 signed
- 3-operand adder/subtracter, one input coming from global routing

Embedded multipliers and more ...

Xilinx - DSP-block evolution

VirtexII-Pro

- starts-off as a 18×18 signed multiplier (k=17)

VirtexIV – DSP48

- multiplier followed by a 48-bit adder/subtracter unit
- adder/subtracter inputs are cascadable
- optional registers present

VirtexV – DSP48E

- asymmetrical multiplier of 18×25 signed
- 3-operand adder/subtracter, one input coming from global routing

Embedded multipliers and more ...

Xilinx - DSP-block evolution

VirtexII-Pro

- starts-off as a 18×18 signed multiplier (k=17)

VirtexIV – DSP48

- multiplier followed by a 48-bit adder/subtracter unit
- adder/subtracter inputs are cascadable
- optional registers present

VirtexV – DSP48E

- asymmetrical multiplier of 18×25 signed
- 3-operand adder/subtracter, one input coming from global routing

Embedded multipliers and more ...

Altera - DSP-block evolution

Stratix, Stratix-II

- four 18×18 multipliers (may function as unsigned)
- two levels of adders
- can perform:
  - eight 9x9 products
  - four 18x18 products independently or
  - one 36x36-bit product or
  - one 18x18-bit complex product

Stratix III, IV

- two Stratix-II DSPs are coupled to form the new DSP
- neighboring half-DSPs may be cascaded
- less flexible, limited DSP bandwidth: half-DSP cannot be split into 4 independent 18x18-bit products

Embedded multipliers and more ...

Altera - DSP-block evolution

Stratix, Stratix-II

- four 18×18 multipliers (may function as unsigned)
- two levels of adders
- can perform:
  - eight 9x9 products
  - four 18x18 products independently or
  - one 36x36-bit product or
  - one 18x18-bit complex product

Stratix III, IV

- two Stratix-II DSPs are coupled to form the new DSP
- neighboring half-DSPs may be cascaded
- less flexible, limited DSP bandwidth: half-DSP cannot be split into 4 independent 18x18-bit products

Embedded multipliers and more ...

Altera - DSP-block evolution

Stratix, Stratix-II

- four 18×18 multipliers (may function as unsigned)
- two levels of adders
- can perform:
  - eight 9x9 products
  - four 18x18 products independently or
  - one 36x36-bit product or
  - one 18x18-bit complex product

Stratix III, IV

- two Stratix-II DSPs are coupled to form the new DSP
- neighboring half-DSPs may be cascaded
- less flexible, limited DSP bandwidth: half-DSP cannot be split into 4 independent 18x18-bit products

The premise

DSP-blocks are a scarce resource when accelerating double precision floating-point applications [1]

we give

Three recipes for saving DSPs

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

Karatsuba-Ofman algorithm

trading multiplications for additions

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:

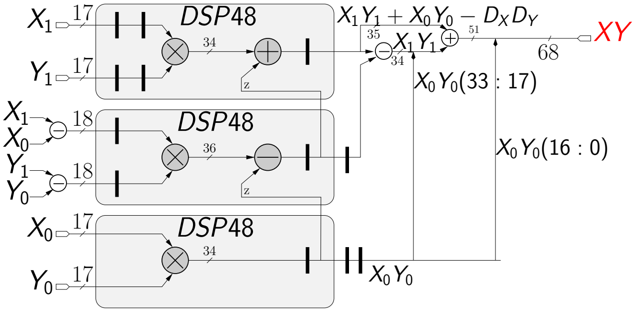
$$X = 2^k X_1 + X_0 \quad \text{and} \quad Y = 2^k Y_1 + Y_0$$

- 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

fairly trivial starting from the equation:

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



- $X_1Y_1 + X_0Y_0 - D_XD_Y$ 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)

Three way splitting

Consider X and Y of size $3k$:

$$X = 2^{2k} X_2 + 2^k X_1 + X_0$$
$$Y = 2^{2k} Y_2 + 2^k Y_1 + Y_0$$

Three way splitting

Consider X and Y of size $3k$:

$$X = 2^{2k}X_2 + 2^kX_1 + X_0$$
$$Y = 2^{2k}Y_2 + 2^kY_1 + Y_0$$

Compute the products:

$$P_{22} = X_2Y_2 \qquad P_{21} = (X_2 - X_1) \times (Y_2 - Y_1)$$
$$P_{11} = X_1Y_1 \qquad P_{10} = (X_1 - X_0) \times (Y_1 - Y_0)$$
$$P_{00} = X_0Y_0 \qquad P_{20} = (X_2 - X_0) \times (Y_2 - Y_0)$$

Three way splitting

Consider X and Y of size $3k$:

$$X = 2^{2k} X_2 + 2^k X_1 + X_0$$
$$Y = 2^{2k} Y_2 + 2^k Y_1 + Y_0$$

Compute the products:

$$P_{22} = X_2 Y_2 \qquad P_{21} = (X_2 - X_1) \times (Y_2 - Y_1)$$
$$P_{11} = X_1 Y_1 \qquad P_{10} = (X_1 - X_0) \times (Y_1 - Y_0)$$
$$P_{00} = X_0 Y_0 \qquad P_{20} = (X_2 - X_0) \times (Y_2 - Y_0)$$

The product XY uses **6 DSPs** instead of 9:

$$\begin{aligned} XY &= 2^{4k} P_{22} \\ &+ 2^{3k}(P_{22} + P_{11} - P_{21}) \\ &+ 2^{2k}(P_{22} + P_{11} + P_{00} - P_{20}) \\ &+ 2^{k}(P_{11} + P_{00} - P_{10}) \\ &+ P_{00} \end{aligned}$$

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)
- the results for K-O-3* are obtained with ISE 9.2i and could not be reproduced with ISE 11.1.

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)
- the results for K-O-3* are obtained with ISE 9.2i and could not be reproduced with ISE 11.1.

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)
- the results for K-O-3* are obtained with ISE 9.2i and could not be reproduced with ISE 11.1.

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)
- the results for K-O-3* are obtained with ISE 9.2i and could not be reproduced with ISE 11.1.

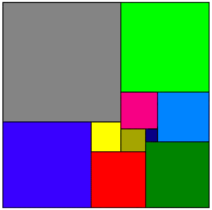
Results – 51x51bit multiplier on Virtex-4

	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)
- the results for K-O-3* are obtained with ISE 9.2i and could not be reproduced with ISE 11.1.

Non-standard tilings



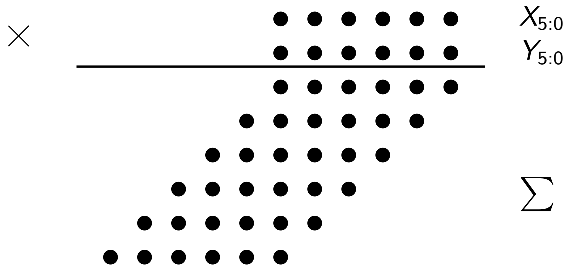
new multiplier family

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 $XY = \sum tile\_contribution$

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

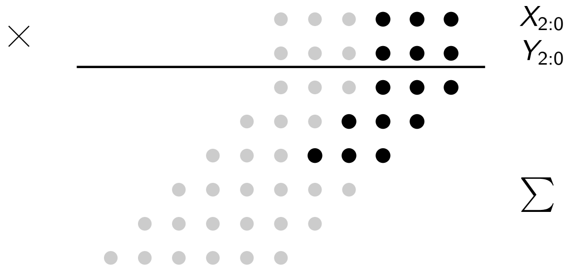
From multiplication to tiling



- 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 $XY = \sum tile\_contribution$

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

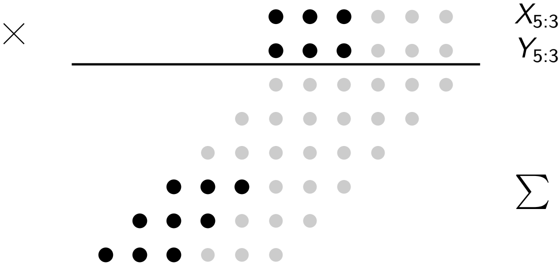
From multiplication to tiling



- 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 $XY = \sum tile\_contribution$

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

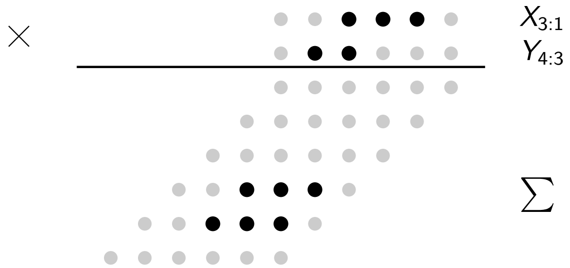
From multiplication to tiling



- 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 $XY = \sum tile\_contribution$

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

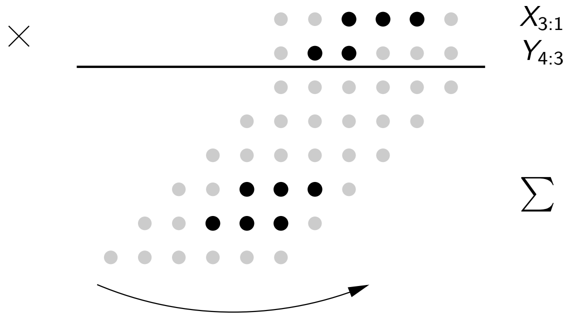
From multiplication to tiling



- 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

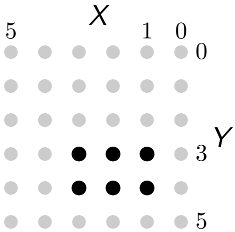


- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

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

From multiplication to tiling

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

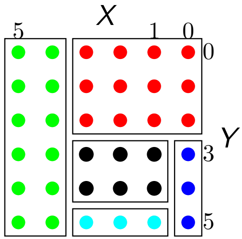


- classical binary multiplication
- all subproducts can be properly located inside the diamond
- create a rectangle by forgetting the shifts
- fill rectangle with tiles
- translate the tiling into an architecture $XY = \sum tile\_contribution$

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

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 $XY = \sum tile\_contribution$

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

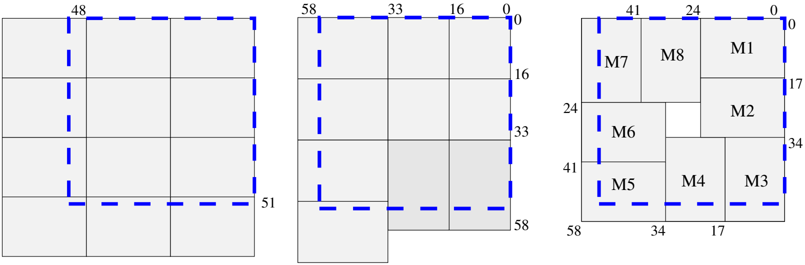
Non-standard tilings

- 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

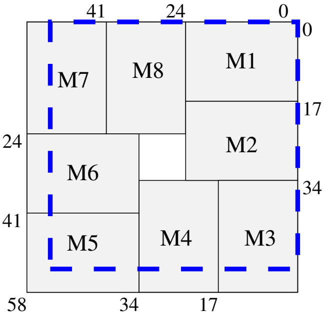
Tilings – 53 × 53-bit multiplication on Virtex5



(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



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

- $X_{24:33}Y_{24:33}$ (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)[2]

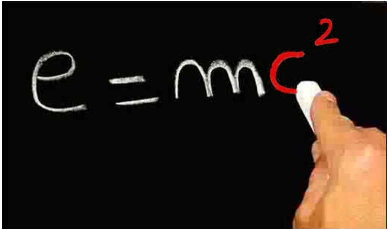
	latency	Freq.	REGs	LUTs	DSPs
LogiCore	14	440	300	249	10
LogiCore	8	338	208	133	10
LogiCore	4	95	208	17	10
Tiling	4	366	247	388	8

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

[2]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*.

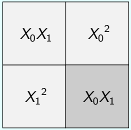
Squarers

- 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

$\leq 34 - bit$

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



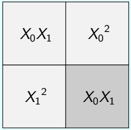
Squarers

- 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

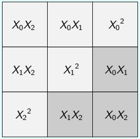
$\leq 34 - bit$

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



$\leq 51 bit$

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



*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

- 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

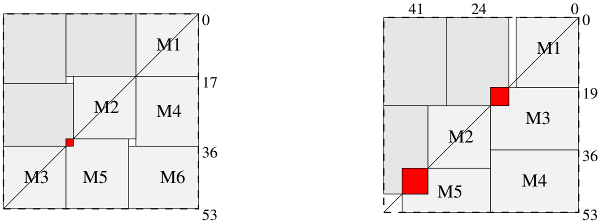
Results – 32-bit and 53-bit squarers on Virtex-4

	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

Summary

- Karatsuba-Ofman reduces DSP cost from 4 to 3, 9 to 6, 16 to 10 at small price
- introduced original family of multipliers and squarer architectures for Virtex-5 using the concept of tiling
- dedicated squarers save a huge number of DSPs (10 DSPs for DP)

Conclusions

- 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

Conclusions

- 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

Future work

- obtain results for Altera targets
- currently working at the generic implementation of the tilling algorithm

Conclusions

- 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

Future work

- obtain results for Altera targets
- currently working at the generic implementation of the tilling algorithm

Squarers and Karatsuba found in FloPoCo

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

Thank you for your attention !

Questions ?