FPGA Multipliers

Bogdan PASCA

projet Arénaire, ENS-Lyon/INRIA/CNRS/Université de Lyon, France

RAIM’11
February 7-10, 2011
Background & Context

Algorithmic techniques for reducing DSP count of large multipliers
  Karatsuba-Ofman algorithm
  Non-Standard tilings
  Squarers
  Truncated multipliers

Conclusions
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
- 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
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 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**
- 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**
What can we compute?

\[
x_2 x_1 x_0 \times
\]

\[
\begin{array}{cccc}
 l_2 & l_1 & l_0 & + \\
 u_2 & u_1 & u_0 & \\
 p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
\]

\[
\begin{align*}
l_0 & = y_0 \land x_0 \\
l_1 & = y_0 \land x_1 \\
l_2 & = y_0 \land x_2 \\
u_0 & = y_1 \land x_0 \\
u_1 & = y_1 \land x_1 \\
u_2 & = y_1 \land x_2
\end{align*}
\]
What can we compute?

\[
\begin{array}{c}
x_2 x_1 x_0 \times \\
y_1 y_0 \\
\hline
l_2 \quad l_1 \quad l_0 + \\
\hline
u_2 u_1 u_0 \\
\hline
p_4 p_3 p_2 p_1 p_0
\end{array}
\]

\[
\begin{align*}
l_0 &= y_0 \land x_0 \\
l_1 &= y_0 \land x_1 \\
l_2 &= y_0 \land x_2 \\
u_0 &= y_1 \land x_0 \\
u_1 &= y_1 \land x_1 \\
u_2 &= y_1 \land x_2
\end{align*}
\]
What can we compute?

\[
\begin{align*}
x_2 x_1 x_0 & \times \quad y_1 y_0 \\ l_2 & l_1 \quad l_0 + \\ u_2 u_1 u_0 & \quad p_4 p_3 p_2 p_1 p_0
\end{align*}
\]

\[
\begin{align*}
l_0 & = y_0 \land x_0 \\
l_1 & = y_0 \land x_1 \\
l_2 & = y_0 \land x_2 \\
u_0 & = y_1 \land x_0 \\
u_1 & = y_1 \land x_1 \\
u_2 & = y_1 \land x_2
\end{align*}
\]
What can we compute?

\[ x_2 x_1 x_0 \times \\
\frac{y_1 y_0}{l_2 l_1 l_0 +} \\
\frac{u_2 u_1 u_0}{p_4 p_3 p_2 p_1 p_0} \]

\[
\begin{align*}
  l_0 &= y_0 \land x_0 \\
  l_1 &= y_0 \land x_1 \\
  l_2 &= y_0 \land x_2 \\
  u_0 &= y_1 \land x_0 \\
  u_1 &= y_1 \land x_1 \\
  u_2 &= y_1 \land x_2
\end{align*}
\]
### Multiplication in logic is expensive

- \( n \times n \) bit \( \approx \left( n^2 \right) + n(n - 1) \) LUTs
  - partial products
  - adder tree
- \( 18 \times 18 \) bit \( \approx 324LUT + 306LUT = 630LUTs \)
- 1 DSP block = 8 LEs (size on FPGA layout)
Need of DSP blocks

Multiplication in logic is expensive

- $n \times n$ bit $\approx n^2 + n(n-1)\text{LUTs}$
  - partial products
  - adder tree
- $18 \times 18$ bit $\approx 324\text{LUT} + 306\text{LUT} = 630\text{LUTs}$
- 1 DSP block = 8 LEs (size on FPGA layout)

DSP blocks are a need in modern FPGAs

Bogdan PASCA FPGA Multipliers
DSP-Hungry Applications

- **FPGA floating point performance – a pencil and paper evaluation** \(^1\)
  - DSP-blocks are a scarce resource for accelerating DP apps.
- **Efficient reconfigurable design for pricing asian options** \(^2\)
  - LUTs 46%, RAM 4%, DSP 100% (192)
- **Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system** \(^3\)
  - a) LE 30%, DSP 86%, b) LE 52%, DSP 88%, c) LE 63%, DSP 100%
- **A temporal coding hardware implementation for spiking neural networks** \(^4\)
  - 16PE: LE 22%, RAM 3%, DSP 74% (100/136)

---

1. D. Strenski (HPCWire, 2007.)
2. Anson H.T. Tse, David B. Thomas, K. H. Tsoi, Wayne Luk (HEART’10)
3. H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano (HEART’10)
4. Marco Nuno-Maganda, Cesar Torres-Huitzil (HEART’10)
DSP-Hungry Applications

- **FPGA floating point performance – a pencil and paper evaluation**¹ → DSP-blocks are a scarce resource for accelerating DP apps.
- **Efficient reconfigurable design for pricing asian options**² → LUTs 46%, RAM 4%, DSP 100% (192)
- **Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system**³ → a) LE 30%, DSP 86%, b) LE 52%, DSP 88%, c) LE 63%, DSP 100%
- **A temporal coding hardware implementation for spiking neural networks**⁴ → 16PE: LE 22%, RAM 3%, DSP 74% (100/136)

Four recipes for saving DSPs

---

¹ D. Strenski (HPCWire, 2007.)
² Anson H.T. Tse, David B. Thomas, K. H. Tsoi, Wayne Luk (HEART’10)
³ H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano (HEART’10)
⁴ Marco Nuno-Maganda, Cesar Torres-Huitzil (HEART’10)
DSP-Hungry Applications

- **FPGA floating point performance – a pencil and paper evaluation**: DSP-blocks are a scarce resource for accelerating DP apps.

- **Efficient reconfigurable design for pricing asian options**: LUTs 46%, RAM 4%, **DSP 100%** (192)

- **Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system**: a) LE 30%, DSP 86%, b) LE 52%, DSP 88%, c) LE 63%, DSP 100%

- **A temporal coding hardware implementation for spiking neural networks**: 16PE: LE 22%, RAM 3%, **DSP 74%** (100/136)

---

1. D. Strenski (HPCWire, 2007.)
2. Anson H.T. Tse, David B. Thomas, K. H. Tsoi, Wayne Luk (HEART’10)
3. H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano (HEART’10)
4. Marco Nuno-Maganda, Cesar Torres-Huitzil (HEART’10)
DSP-Hungry Applications

- **FPGA floating point performance – a pencil and paper evaluation**¹
  → DSP-blocks are a scarce resource for accelerating DP apps.
- **Efficient reconfigurable design for pricing asian options**²
  → LUTs 46%, RAM 4%, DSP 100% (192)
- **Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system**³
  → a) LE 30%, DSP 86%, b) LE 52%, DSP 88%, c) LE 63%, DSP 100%
- **A temporal coding hardware implementation for spiking neural networks**⁴
  → 16PE: LE 22%, RAM 3%, DSP 74% (100/136)

---

¹ D. Strenski (HPCWire, 2007.)
² Anson H.T. Tse, David B. Thomas, K. H. Tsoi, Wayne Luk (HEART’10)
³ H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano (HEART’10)
⁴ Marco Nuno-Maganda, Cesar Torres-Huitzil (HEART’10)
DSP-Hungry Applications

- **FPGA floating point performance – a pencil and paper evaluation**
  → DSP-blocks are a scarce resource for accelerating DP apps.

- **Efficient reconfigurable design for pricing asian options**
  → LUTs 46%, RAM 4%, DSP 100% (192)

- **Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system**
  → a) LE 30%, DSP 86%, b) LE 52%, DSP 88%, c) LE 63%, DSP 100%

- **A temporal coding hardware implementation for spiking neural networks**
  → 16PE: LE 22%, RAM 3%, DSP 74% (100/136)

---

1 D. Strenski (HPCWire, 2007.)
2 Anson H.T. Tse, David B. Thomas, K. H. Tsoi, Wayne Luk (HEART’10)
3 H. Morisita, K. Inakagata, Y. Osana, N. Fujita, H. Amano (HEART’10)
4 Marco Nuno-Maganda. Cesar Torres-Huitzil (HEART’10)
classical binary multiplication
all sub-products can be properly located inside the diamond
rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
Perceiving Multiplications Visually

\[ XY = X_0 Y_0 + 2^3 X_0 Y_1 + 2^3 X_1 Y_0 + 2^{3+3} X_1 Y_1 \]

- classical binary multiplication
- all sub-products can be properly located inside the diamond
- rotate the diamond so to obtain a rectangle
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
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
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
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

Bogdan PASCA   FPGA Multipliers
Visual Interpretation

\[ X_1 \quad X_0 \]
\[ Y_0 \]
\[ Y_1 \]
Visual Interpretation

Bogdan PASCA  FPGA Multipliers
Visual Interpretation
Visual Interpretation

\[ X_1 \quad X_0 \]
\[ \quad Y_0 \quad Y_1 \]
\[ \quad Y_2 \]

\[ X_2 \quad X_1 \quad X_0 \]
\[ \quad Y_0 \quad Y_1 \quad Y_2 \]
Visual Interpretation
Visual Interpretation

\[ X_1 \quad X_0 \quad Y_0 \]

\[ X_1 \quad X_0 \quad Y_0 \]

\[ X_2 \quad X_1 \quad X_0 \quad Y_0 \]

\[ X_2 \quad X_1 \quad X_0 \quad Y_0 \]

\[ Y_1 \]

\[ Y_1 \]

\[ Y_2 \]
Visual Interpretation

X_1  X_0
Y_0
Y_1

X_2  X_1  X_0
Y_0
Y_1
Y_2

X_3  X_2  X_1  X_0
Y_0
Y_1
Y_2
Y_3
fairly trivial starting from the equation:

\[ XY = 2^{2k} X_1 Y_1 + 2^k (X_1 Y_1 + X_0 Y_0 - D_X D_Y) + X_0 Y_0 \]

34x34bit multiplier using Virtex-4 DSP48

- \( X_1 Y_1 + X_0 Y_0 - D_X D_Y \) is implemented inside the DSPs
- need to recover \( X_1 Y_1 \) with a subtraction
### Results

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>frequency (MHz)</th>
<th>slices</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogiCore</td>
<td>6</td>
<td>447</td>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>LogiCore</td>
<td>3</td>
<td>176</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>K-O-2</td>
<td>3</td>
<td>317</td>
<td>95</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table:** 34x34-bit multipliers on Virtex-4

Trade-off 1DSPs (>630 Logic Elements) for 138 Logic Elements

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>frequency (MHz)</th>
<th>slices</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogiCore</td>
<td>11</td>
<td>353</td>
<td>185</td>
<td>9</td>
</tr>
<tr>
<td>LogiCore</td>
<td>6</td>
<td>264</td>
<td>122</td>
<td>9</td>
</tr>
<tr>
<td>K-O-3</td>
<td>6</td>
<td>317</td>
<td>331</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table:** 51x51 multipliers on Virtex-4

Trade-off 3DSPs (>1890 Logic Elements) for 292 Logic Elements

---

5 On Virtex4 devices 1 slice = 2 Logic Elements
Non-Standard tilings

new multiplication algorithms
Non-standard tilings

- Optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- Classical decomposition may produce suboptimal results
  - Chunk size for X is 24
  - Chunk size for Y is 17
- Translate the operand decomposition into a tiling problem
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for $X$ is 24
  - chunk size for $Y$ is 17
- translate the **operand decomposition** into a **tiling** problem

\[
\sum \quad \times \\
\]

\[
\begin{array}{cccccc}
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

\[
\sum
\]
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for X is 24
  - chunk size for Y is 17
- translate the operand decomposition into a tiling problem
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for $X$ is 24
  - chunk size for $Y$ is 17
- translate the operand decomposition into a tiling problem

\[
\begin{array}{c}
\times \\
\hline
X_{5:3} \\
\sum
\end{array}
\]
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for X is 24
  - chunk size for Y is 17
- translate the operand decomposition into a tiling problem
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for X is 24
  - chunk size for Y is 17
- translate the operand decomposition into a tiling problem
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for $X$ is 24
  - chunk size for $Y$ is 17
- translate the operand decomposition into a tiling problem

$$2^{3+1} X_{3:1} Y_{4:3}$$
Non-standard tilings

- optimize use of rectangular multipliers on Virtex5 (25x18 signed)
- classical decomposition may produce suboptimal results
  - chunk size for X is 24
  - chunk size for Y is 17
- translate the operand decomposition into a tiling problem

\[ XY = 2^{3+1}X_{3:1}Y_{4:3} + 2^4X_{5:4}Y_{5:0} + X_{3:0}Y_{2:0} + 2^3X_0Y_{5:3} + 2^{1+5}X_{3:1}Y_5 \]
Performing a $53 \times 53$-bit multiplication on Virtex5

(a) standard tiling

(b) Logicore tiling

(c) proposed tiling

- **standard tiling** $\equiv$ 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
\[
XY = X_{0:23}Y_{0:16} + 2^{17}(X_{0:23}Y_{17:33}) + 2^{17}(X_{0:16}Y_{34:57}) + 2^{17}(X_{17:33}Y_{34:57}) + 2^{24}(X_{24:40}Y_{0:23}) + 2^{17}(X_{41:57}Y_{0:23}) + 2^{17}(X_{34:57}Y_{24:40}) + 2^{17}(X_{34:57}Y_{41:57}) + 2^{48}(X_{24:33}Y_{24:33})
\]

- \(X_{24:33}Y_{24:33}\) (10x10 multiplier) probably best implemented in LUTs.
- parenthesis makes best use of DSP48E internal adders (17-bit shifts)
58x58 multipliers on Virtex-5 (5vlx50ff676-3)\(^6\)

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>Freq.</th>
<th>REGs</th>
<th>LUTs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogiCore</td>
<td>14</td>
<td>440</td>
<td>300</td>
<td>249</td>
<td>10</td>
</tr>
<tr>
<td>LogiCore</td>
<td>8</td>
<td>338</td>
<td>208</td>
<td>133</td>
<td>10</td>
</tr>
<tr>
<td>LogiCore</td>
<td>4</td>
<td>95</td>
<td>208</td>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>Tiling</td>
<td>4</td>
<td>366</td>
<td>247</td>
<td>388</td>
<td>8</td>
</tr>
</tbody>
</table>

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

\(^6\)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*. 

\[
\begin{align*}
(2^kX_1 + X_0)^2 &= 2^{2k}X_2^2 + 2^{2k+1}X_1X_0 + X_2^0 \\
&= 2^{2k}X_2^2 + 2^{2k}X_1X_0 + 2^{2k}X_0X_1 + 2^{2k}X_0X_0 \\
&= 2^{2k}X_2^2 + 2^{2k}X_1X_0 + 2^{2k}X_0X_1 + 2^{2k}X_0^2
\end{align*}
\]
Squaring with $k = 17$ on a Virtex-4

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

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

\[
(2^{2k} X_2 + 2^k X_1 + X_0)^2 = 2^{4k} X_2^2 + 2^{2k} X_1^2 + X_0^2 + 2 \cdot 2^{3k} X_2 X_1 + 2 \cdot 2^{2k} X_2 X_0 + 2 \cdot 2^k X_1 X_0
\]
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
- the DSP48 of VirtexIV allow shifts of 17 so internal adders unused
\[ (2^k X_1 + X_0)^2 = 2^{34} X_1^2 + 2^{18} X_1 X_0 + X_0^2 \]

- shifts of 0, 18, 34 the previous equation
- 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

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>frequency</th>
<th>slices</th>
<th>DSPs</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogiCore</td>
<td>6</td>
<td>489</td>
<td>59</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>LogiCore</td>
<td>3</td>
<td>176</td>
<td>34</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Squarer</td>
<td>3</td>
<td>317</td>
<td>18</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LogiCore</td>
<td>18</td>
<td>380</td>
<td>279</td>
<td>16</td>
<td>53</td>
</tr>
<tr>
<td>LogiCore</td>
<td>7</td>
<td>176</td>
<td>207</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Squarer</td>
<td>7</td>
<td>317</td>
<td>332</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

- DSPs saved without any overhead
- Impressive **10 DSPs saved** for double precision squarer
the tiling technique can be extended to squaring

Issues
- darker squares are computed twice thus need be removed.
- thanks to symmetry diagonal multiplication of size $n$ should consume only $n(n + 1)/2$ LUTs instead of $n^2$. 
Truncated multipliers
Truncated multipliers

Classical technique

- reduce resources, delay, or power consumption
- controlled accuracy degradation
Truncated multipliers

Classical technique
- reduce resources, delay, or power consumption
- controlled accuracy degradation

\[ \sum \times \]

\[ \begin{array}{c}
A \\
B \\
\end{array} \]

\[ \begin{array}{c}
v \\
u \\
k \\
\end{array} \]

\[ \begin{array}{c}
n - k \\
d \\
k \\
\end{array} \]
Truncated multipliers

Classical technique

- reduce resources, delay, or power consumption
- controlled accuracy degradation

\[ \sum BA \]

remove some of the least-significant \( d \) columns

keep the error smaller than \( 2^k \)
Error budget

\[ E_{total} = E_{approx} + E_{round} \leq 2^k \]

- \( E_{round} \) - caused by rounding the \( n - d \)-bit result to \( n - k \) bits
  - use compensation bit to center the error
  - round to nearest bounds \( E_{round} \leq 2^{k-1} \)
Error budget

\[ E_{\text{total}} = E_{\text{approx}} + E_{\text{round}} \leq 2^k \]

- \( E_{\text{round}} \) – caused by rounding the \( n - d \)-bit result to \( n - k \) bits
  - use compensation bit to center the error
  - round to nearest bounds \( E_{\text{round}} \leq 2^{k-1} \)
- \( E_{\text{approx}} \) – caused by the truncation of the \( d \) columns
  \[
  \begin{cases}
  0 \leq E_{\text{approx}} \leq \sum_{i=1}^{d} i2^{i-1} \\
  E_{\text{approx}} < 2^{k-1}
  \end{cases}
  \rightarrow d = f(k)\]
**Tiling the truncated board**

- **Sol 1**: tile and discard columns (save additions)
  - waste DSPs
- **Sol 2**: use softcore multiplier (trade a DSP for logic)
- **Best**: tile with softcore multipliers so that $E_{approx} \leq 2^{k-1}$
  - use the extra precision for free
Tiling the truncated board

- **Sol 1**: tile and discard columns (save additions)
  - waste DSPs

- **Sol 2**: use softcore multiplier (trade a DSP for logic)

- **Best**: tile with softcore multipliers so that $E_{\text{approx}} \leq 2^{k-1}$
  - use the extra precision for free
Tiling the truncated board

- **Sol 1**: tile and discard columns (save additions)
  - waste DSPs
- **Sol 2**: use softcore multiplier (trade a DSP for logic)
- **Best**: tile with softcore multipliers so that $E_{approx} \leq 2^{k-1}$
  - use the extra precision for free
Mantissa Multipliers for SP, DP, QP, Virtex4 (left) and Virtex5 (right)

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Prec.</th>
<th>Latency, Freq.</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtex5</td>
<td>DP</td>
<td>6 cycles @ 414MHz</td>
<td>320LUT 302REG 5DSP</td>
</tr>
<tr>
<td></td>
<td>QP</td>
<td>20 cycles @ 334MHz</td>
<td>2497LUT 2321REG 19DSP</td>
</tr>
<tr>
<td></td>
<td>QP</td>
<td>14 cycles @ 245MHz</td>
<td>2249LUT 1576REG 19DSP</td>
</tr>
<tr>
<td>Virtex4</td>
<td>DP</td>
<td>11 cycles @ 368MHz</td>
<td>358sl. 7DSP</td>
</tr>
<tr>
<td></td>
<td>QP</td>
<td>21 cycles @ 368MHz</td>
<td>1735sl. 26DSP</td>
</tr>
</tbody>
</table>

- **Virtex4**
  - DP reduce DSPs from 10 to 7 while also reducing slice count
  - QP reduce DSPs from 49 to 26 at without any slice penalty
- **Virtex5**
  - DP reduce DSP from 6 to 5 for and roughly half the LUTs and REGs
  - QP reduce DSP from 34 to 19 at a small increase in logic resources.
Another point of view

\[(w_E, w_F) \text{ correctly rounded} \quad = \text{accuracy} \quad (w_E, w_F + 1) \text{ faithfully rounded} \]
\[\rightarrow \text{in FPGAs the extra bit comes for free}^* \]

- truncate multipliers when IEEE-754 compliance is not needed
  - function approximation by polynomial evaluation

\[\log_2(1 + x) \text{ (53-bit)}\]

<table>
<thead>
<tr>
<th></th>
<th>default</th>
<th>27 DSPs</th>
<th>23 DSPs</th>
<th>11 DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimized Horner</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimized Horner + truncated multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Bogdan PASCA FPGA Multipliers 28*
Another point of view

\[(w_E, w_F) = \text{accuracy} (w_E, w_F + 1)\]

- correctly rounded \(\rightarrow\) in FPGAs the extra bit comes for free*

- truncate multipliers when IEEE-754 compliance is not needed
  - function approximation by polynomial evaluation

\[\log_2(1 + x) \text{ (53-bit)}\]

<table>
<thead>
<tr>
<th></th>
<th>default</th>
<th>27 DSPs</th>
<th>23 DSPs</th>
<th>11* DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimized Horner</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimized Horner + truncated multipliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
save DSPs by exploiting the flexibility of the FPGA
- **Karatsuba-Ofman** reduces DSP cost at small price in logic elements
- **tiling** techniques adapt better to asymmetric DSPs
- dedicated **squarers** significantly reduce DSP count
- control accuracy and save DSPs using **truncated multipliers**
Thank you for your attention!

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

Questions?