Pipelined FPGA Adders

Bogdan Pasca

joint work with:
Florent de Dinechin and Hong Diep Nguyen

FPL, Aug. 31st - Sep. 2nd, 2010
Why a paper on integer addition on FPGAs in 2010?
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

Goals

- provide new and exotic arithmetic operators
- highly complex operators (floating-point elementary functions)
- specialized operators (constant multipliers, squarers)
- ... (currently about ∞ operators in about 50 classes)
- optimized for a wide range of contexts
- adapt to the application’s precision needs
- match target hardware
- run at target frequency
- minimize resource consumption

Method:

assemble sub-components built for the same target frequency
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

Goals

- provide new and exotic arithmetic operators
  - highly complex operators (floating-point elementary functions)
  - specialized operators (constant multipliers, squarers)
- optimized for a wide range of contexts
- adapt to the application’s precision needs
- match target hardware
- run at target frequency
- minimize resource consumption
in an easy to use open-source framework.
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

- Goals
  - provide new and exotic arithmetic operators
    - highly complex operators (floating-point elementary functions)
    - specialized operators (constant multipliers, squarers)
    - ... (currently about $\infty$ operators in about 50 classes)
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

- Goals
  - provide new and exotic arithmetic operators
    - highly complex operators (floating-point elementary functions)
    - specialized operators (constant multipliers, squarers)
    - ... (currently about $\infty$ operators in about 50 classes)
  - optimized for a wide range of contexts
    - adapt to the application’s precision needs
    - match target hardware
    - run at target frequency
    - minimize resource consumption
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

Goals

- provide new and exotic arithmetic operators
  - highly complex operators (floating-point elementary functions)
  - specialized operators (constant multipliers, squarers)
  - ... (currently about $\infty$ operators in about 50 classes)
- optimized for a wide range of contexts
  - adapt to the application's precision needs
  - match target hardware
  - run at target frequency
  - minimize resource consumption
- in an easy to use open-source framework.
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

- Goals
  - provide new and exotic arithmetic operators
    - highly complex operators (floating-point elementary functions)
    - specialized operators (constant multipliers, squarers)
    - ... (currently about $\infty$ operators in about 50 classes)
  - optimized for a wide range of contexts
    - adapt to the application’s precision needs
    - match target hardware
    - run at target frequency
    - minimize resource consumption
  - in an easy to use open-source framework.

- Method:
  - assemble sub-components built for the same target frequency
Why a paper on integer addition on FPGAs in 2010?

The FloPoCo project

- Goals
  - provide new and exotic arithmetic operators
    - highly complex operators (floating-point elementary functions)
    - specialized operators (constant multipliers, squarers)
    - ... (currently about $\infty$ operators in about 50 classes)
  - optimized for a wide range of contexts
    - adapt to the application’s precision needs
    - match target hardware
    - run at target frequency
    - minimize resource consumption
  - in an easy to use open-source framework.

- Method:
  - assemble sub-components built for the same target frequency

Addition is the most useful of these components
flopoco -frequency=400  FPExp 11 52  (double precision)

(...)
| | | | | |---Entity IntAdder_16_f325_slice_SRL_noBUFFER_uid10_Alternative:
| | | | | | Not pipelined

(...)
| | | | |---Entity IntAdder_52_f400_slice_SRL_noBUFFER_uid16_Classical:
| | | | | Pipeline depth = 1
| | | |---Entity IntCompressorTree_52_2.uid15:
| | | | Pipeline depth = 1
| | |---Entity IntTruncMultiplier_30_34_35.signed:
| | | Pipeline depth = 4
| |---Entity IntAdder_42_f400_slice_SRL_BUFFER_uid17_Classical:
| | Pipeline depth = 1
|---Entity PolynomialEvaluator.d2:
| | Pipeline depth = 15
|---Entity FunctionEvaluator:
| Pipeline depth = 17
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER_uid18_Classical:
| Pipeline depth = 2
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER_uid19.Classical:
| Pipeline depth = 2
| |---Entity IntAdder_85_f325_slice_SRL_noBUFFER_uid22.Classical:
| | Pipeline depth = 1
|---Entity IntCompressorTree_85_3_uid21:
| | Pipeline depth = 2
|---Entity IntMultiplier_47.48.uid20:
| Pipeline depth = 6
|---Entity IntAdder_57_f400_slice_SRL_noBUFFER_uid23.Classical:
| Pipeline depth = 2
|---Entity IntAdder_65_f400_slice_SRL_noBUFFER_uid24.Classical:
| Pipeline depth = 2
Entity FPExp.11.52.400:
Pipeline depth = 52
flopoco -frequency=400 FPExp 11 52 (double precision)

(...)
| | | | |---Entity IntAdder_16_f325_slice_SRL_noBUFFER_uid10_Alternative:
| | | | | Not pipelined
(...)
| | | | |---Entity IntAdder_52_f400_slice_SRL_noBUFFER_uid16_Classical:
| | | | | Pipeline depth = 1
| | | | |---Entity IntCompressorTree_52_2_uid15:
| | | | | Pipeline depth = 1
| | | | |---Entity IntTruncMultiplier_30_34_35_signed:
| | | | | Pipeline depth = 4
| | | | |---Entity IntAdder_42_f400_slice_SRL_BUFFER_uid17_Classical:
| | | | | Pipeline depth = 1
| |---Entity PolynomialEvaluator_d2:
| | Pipeline depth = 15
|---Entity FunctionEvaluator:
| Pipeline depth = 17
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER_uid18_Classical:
| Pipeline depth = 2
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER_uid19_Classical:
| Pipeline depth = 2
| | |---Entity IntAdder_85_f325_slice_SRL_noBUFFER_uid22_Classical:
| | | Pipeline depth = 1
| |---Entity IntCompressorTree_85_3_uid21:
| | Pipeline depth = 2
|---Entity IntMultiplier_47_48_uid20:
| Pipeline depth = 6
|---Entity IntAdder_57_f400_slice_SRL_noBUFFER_uid23_Classical:
| Pipeline depth = 2
|---Entity IntAdder_65_f400_slice_SRL_noBUFFER_uid24_Classical:
| Pipeline depth = 2
Entity FPExp_11_52_400:
Pipeline depth = 52
flopoco -frequency=400 FPExp 11 52 (double precision)

(...)
| | | | |---Entity IntAdder_16_f325_slice_SRL_noBUFFER_uid10_Alternative:  
| | | | | Not pipelined

(...)
| | | | |---Entity IntAdder_52_f400_slice_SRL_noBUFFER_uid16_Classical:  
| | | | | Pipeline depth = 1
| | | | |---Entity IntCompressorTree_52_2.uid15:  
| | | | | Pipeline depth = 1
| | | | |---Entity IntTruncMultiplier_30_34.35_signed:  
| | | | | Pipeline depth = 4
| | | | |---Entity IntAdder_42_f400_slice_SRL_BUFFER_uid17_Classical:  
| | | | | Pipeline depth = 1
| | |---Entity PolynomialEvaluator_d2:  
| | | Pipeline depth = 15
|---Entity FunctionEvaluator:  
| | Pipeline depth = 17
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER_uid18_Classical:  
| Pipeline depth = 2
|---Entity IntAdder_48_f400_slice_SRL_noBUFFER.uid19.Classical:  
| Pipeline depth = 2
| | |---Entity IntAdder_85_f325_slice_SRL_noBUFFER_uid22.Classical:  
| | | Pipeline depth = 1
| |---Entity IntCompressorTree_85_3.uid21:  
| | | Pipeline depth = 2
|---Entity IntMultiplier_47.48.uid20:  
| | Pipeline depth = 6
|---Entity IntAdder_57_f400_slice_SRL_noBUFFER.uid23.Classical:  
| | Pipeline depth = 2
|---Entity IntAdder_65_f400_slice_SRL_noBUFFER.uid24.Classical:  
| | Pipeline depth = 2
Entity FPExp_11.52.400:  
Pipeline depth = 52
flopoco -frequency=200  FPExp 11 52  (double precision)

(...)
| | | | |---Entity IntAdder_16_f200_slice_SRL_noBUFFER_uid10_Alternative:
| | | | | Not pipelined
(...)
| | | | |---Entity IntAdder_52_f200_slice_SRL_noBUFFER_uid16_Alternative:
| | | | | Not pipelined
| | | | |---Entity IntCompressorTree_52.2.uid15:
| | | | | Not pipelined
| | | | |---Entity IntTruncMultiplier_30.34.35.signed:
| | | | Pipeline depth = 2
| | | |---Entity IntAdder_42_f200_slice_SRL_BUFFER_uid17_Classical:
| | | | Not pipelined
| | |---Entity PolynomialEvaluator_d2:
| | | Pipeline depth = 7
|---Entity FunctionEvaluator:
| Pipeline depth = 9
|---Entity IntAdder_48_f200_slice_SRL_noBUFFER_uid18_Alternative:
| Pipeline depth = 1
|---Entity IntAdder_48_f200_slice_SRL_noBUFFER_uid19_Alternative:
| Pipeline depth = 1
| | |---Entity IntAdder_85_f200_slice_SRL_noBUFFER_uid22_Alternative:
| | | Pipeline depth = 1
| | |---Entity IntCompressorTree_85.3_uid21:
| | | Pipeline depth = 1
|---Entity IntMultiplier_47.48_uid20:
| Pipeline depth = 5
|---Entity IntAdder_57_f200_slice_SRL_noBUFFER_uid23_Alternative:
| Pipeline depth = 1
|---Entity IntAdder_65_f200_slice_SRL_noBUFFER_uid24_Alternative:
| Pipeline depth = 1
Entity FPExp.11.52.200:
Pipeline depth = 23
How to generate "best" addition operator from user specifications
How to generate "best" addition operator from user specifications

- elementary block resource estimation
- selection among possible implementations
signal R, X, Y: std_logic_vector(w-1 downto 0);
signal Cin: std_logic;
begin
...
R <= X + Y + Cin;
...
end architecture;
signal R, X, Y: std_logic_vector(w-1 downto 0);
signal Cin: std_logic;
begin
...
l: fast_but_large_adder port map(X, Y, Cin, R); --pipeline depth=0
...
end architecture;
signal R, X, Y: std_logic_vector(w-1 downto 0);
signal Cin: std_logic;
begin
...
l: fast_pipelined_adder port map(X, Y, Cin, R); --pipeline depth=X
...
end architecture;
large number of VLSI studies [Mul89, EL04, DP96, US93]
Q: Why not reuse them?

[XY98]: low-latency unpipelined adders

[MJE97]: Compression tree using a (naive) final pipelined adder

Other more recent papers on accumulation or compression trees

But no comprehensive study on pipelined FPGA addition

FPGA Adders: Performance Evaluation and Optimal Design.

[MJE97] Peiro Marcos Martinez, Valls Javier, and Boemo Eduardo.
On the design of FPGA-based Multioperand Pipeline Adders.
large number of VLSI studies [Mul89, EL04, DP96, US93]
Q: Why not reuse them?
- most hypotheses false for FPGA
- new architecture → new rules
- some ideas can still be recycled

Literature

[XY98]: low-latency unpipelined adders
[MJE97]: Compression tree using a (naive) final pipelined adder

Other more recent papers on accumulation or compression trees

But no comprehensive study on pipelined FPGA addition

large number of VLSI studies [Mul89, EL04, DP96, US93]
Q: Why not reuse them?
- most hypotheses *false* for FPGA
- new architecture → *new rules*
- some ideas can still be recycled

Q: What about FPGA literature?
- [XY98]: low-latency *unpipelined* adders
- [MJE97]: Compression tree using a (naive) final pipelined adder
- Other more recent papers on accumulation or compression trees
large number of VLSI studies [Mul89, EL04, DP96, US93]
Q: Why not reuse them?
- most hypotheses false for FPGA
- new architecture → new rules
- some ideas can still be recycled

Q: What about FPGA literature?
- [XY98]: low-latency unpipelined adders
- [MJE97]: Compression tree using a (naive) final pipelined adder
- Other more recent papers on accumulation or compression trees

But no comprehensive study on pipelined FPGA addition

FPGA Adders: Performance Evaluation and Optimal Design.

[MJE97] Peiro Marcos Martinez, Valls Javier, and Boemo Eduardo.
On the design of FPGA-based Multioperand Pipeline Adders.
Ripple-Carry-Adder

signal R, X, Y: std_logic_vector(w-1 downto 0);
signal Cin: std_logic;
begin
...
R <= X + Y + Cin;
...
end architecture;

\[ T = \text{delay} \cdot 1 + (w - 1) \cdot \text{delay} \cdot \text{Cout} + \max(\text{delay} \cdot \text{Cout}, \text{delay} \cdot S) \]

- \( w \)-bit adder: connect \( w \) FA in series
signal R, X, Y: std_logic_vector(w-1 downto 0);
signal Cin: std_logic;
begin
...
R <= X + Y + Cin;
...
end architecture;

- \( w \)-bit adder: connect \( w \) FA in series
- \( T = delay_1 + (w - 1)delay_{C_{out}} + \max(delay_{C_{out}}, delays_{S}) \)
Pipeline a $w$-bit addition: How To?

- Break carry-propagation path
- \[ T_{\text{RCA}} \alpha < \frac{1}{f} \]
- \[ \alpha = \frac{1}{f} - \text{delay} \max(\text{delay}_{\text{Cout}}, \text{delay}_{S}) \]
- Synchronize I/O using registers to reduce register overhead.

Diagram:
- Inputs: \( X_7, Y_7, X_6, Y_6, X_5, Y_5, X_4, Y_4, X_3, Y_3, X_2, Y_2, X_1, Y_1, X_0, Y_0 \)
- Outputs: \( C_{out}, C_{in} \)
- Intermediate outputs: \( S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0 \)
Pipeline a $w$-bit addition: How To?

- **break carry-propagation path** such that $T_{RCA\alpha} < 1/f$
- $\alpha = \frac{1/f - \text{delay}_1 - \max(\text{delay}_{\text{cout}}, \text{delays}_5)}{\text{delay}_{\text{cout}}} + 1$
- synchronize I/O using registers $\rightarrow$ **register overhead**
Pipeline a $w$-bit addition: How To?

- **Break carry-propagation path** such that $T_{RCA\alpha} < 1/f$
- $\alpha = \frac{1/f - \text{delay}_1 - \max(\text{delay}_{Cout}, \text{delays}_S)}{\text{delay}_{Cout}} + 1$
- Synchronize I/O using registers → **register overhead**
Classical Pipelined Addition Architecture

- encountered in literature [EL04, MJE97]
- named also *traditional*
Classical Pipelined Addition Architecture

- encountered in literature [EL04, MJE97]
- named also traditional

How many resources does it take?
heterogeneous FPGA: **LUTs**, **Registers**, RAM, DSP-blocks etc
Q: What to count?
- Depends on the designer needs
- In our case we prefer to count the **atomic entities**
  - Slices on VirtexII, Spartan3, Virtex4 (should be half-slices)
  - LUTs, Registers on Virtex5
Resource Estimation Techniques

heterogeneous FPGA: LUTs, Registers, RAM, DSP-blocks etc

Q: What to count?
- Depends on the designer needs
- In our case we prefer to count the atomic entities
  - Slices on VirtexII, Spartan3, Virtex4 (should be half-slices)
  - LUTs, Registers on Virtex5

Case Study:
We evaluate the resources required by the classical architecture
Case Study: Classical Architecture

$k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L =$
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L = w +$
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L = w + 2(k - 3)\alpha +$
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L = w + 2(k - 3)\alpha + (k - 2)\alpha +$
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L = w + 2(k - 3)\alpha + (k - 2)\alpha + 2\beta$
Case Study: Classical Architecture

- **k** chunks, \( w \)-bit addition: \( w = (k - 1)\alpha + \beta \)
- \( L = w + 2(k - 3)\alpha + (k - 2)\alpha + 2\beta = w + (3k - 8)\alpha + 2\beta \)
Case Study: Classical Architecture

- \( k \) chunks, \( w \)-bit addition: \( w = (k - 1)\alpha + \beta \)
- \( L = w + 2(k - 3)\alpha + (k - 2)\alpha + 2\beta = w + (3k - 8)\alpha + 2\beta \)
- \( R = (3k - 8)\alpha + 2\beta + \alpha + 2\alpha + k - 1 \)
Case Study: Classical Architecture

- $k$ chunks, $w$-bit addition: $w = (k - 1)\alpha + \beta$
- $L = w + 2(k - 3)\alpha + (k - 2)\alpha + 2\beta = w + (3k - 8)\alpha + 2\beta$
- $R = (3k - 8)\alpha + 2\beta + \alpha + 2\alpha + k - 1$
- $2S = w + (3k - 8)\alpha + 2\beta + 3\alpha + k - 1 - \alpha$
- perform additions as soon as possible
- propagate carry-out bits for diagonal addition
- lower LUT and register consumption than first version

Details and formulas in the paper
1. **operator latency** is a problem
1. **operator latency is a problem**
   - pipeline depth of previous architectures increase linearly with $k$
   - $k$ increases when $f$ or $w$ increase
1. **operator latency is a problem**
   - pipeline depth of previous architectures increase linearly with $k$
   - $k$ increases when $f$ or $w$ increase
   - 128-bit addition @ 400MHz, Virtex4 (-12) takes 3 cycles
1. **operator latency is a problem**
   - pipeline depth of previous architectures increase linearly with $k$
   - $k$ increases when $f$ or $w$ increase
   - 128-bit addition @ 400MHz, Virtex4 (-12) takes 3 cycles

2. **no SRL are available**
   - register count explodes, even for the alternative architecture
basic idea:
- perform a carry-select addition

\[ Y_{k-1} X_{k-1} \]

\[ \begin{array}{cccc}
Y_3 & X_3 & 1 & \\
Y_2 & X_2 & 1 & \\
Y_1 & X_1 & 1 & \\
Y_0 & X_0 & \text{cin} & \\
\end{array} \]

\[ \begin{array}{cccc}
R_{k-1} & R_3 & R_2 & R_1 & R_0
\end{array} \]
basic idea:
- perform a **carry-select addition**
- use dedicated fast-carry lines to speed-up carry-bit computation
knowing $C_{i-1}$, $C_i^0$ and $C_i^1$ we compute $C_i$

$$C_i = f(C_{i-1}, C_i^0, C_i^1)$$

<table>
<thead>
<tr>
<th>$C_{i-1}$</th>
<th>$C_i^0$</th>
<th>$C_i^1$</th>
<th>$C_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
knowing \( C_{i-1}, C_i^0 \) and \( C_i^1 \) we compute \( C_i \)

\[
C_i : \text{Sum}_i = C_{i-1} + C_i^0 + C_i^1
\]

<table>
<thead>
<tr>
<th>( C_{i-1} )</th>
<th>( C_i^0 )</th>
<th>( C_i^1 )</th>
<th>( C_i )</th>
<th>( \text{Sum}_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Short-Latency Architecture: Carry-Adder-Cell

- knowing $C_{i-1}$, $C_i^0$ and $C_i^1$ we compute $C_i$

\[ C_i : \neg C_i : Sum_i = C_{i-1} + C_i^0 + C_i^1 + 2 \]

<table>
<thead>
<tr>
<th>$C_{i-1}$</th>
<th>$C_i^0$</th>
<th>$C_i^1$</th>
<th>$C_i$</th>
<th>$\neg C_i$</th>
<th>$Sum_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- portable VHDL that uses fast-carry lines

```
op0 <= "0" & C4_0 & "0" & C3_0 & "0" & C2_0 & "0" & C1_0;
op1 <= "1" & C4_1 & "1" & C3_1 & "1" & C2_1 & "1" & C1_1;
--perform the short carry additions
rawCarrySum <= op0_d1 + op1_d1 + C0;
```
for some values of \((w, f)\) discard second register level

\[ Y_{k-1} \quad X_{k-1} \]

\[ Y_3 \quad X_3 \quad 1 \]

\[ Y_2 \quad X_2 \quad 1 \]

\[ Y_1 \quad X_1 \quad 1 \]

\[ Y_0 \quad X_0 \quad \text{cin} \]

\[ R_{k-1} \quad R_3 \quad R_2 \quad R_1 \quad R_0 \]

CAC CAC CAC CAC

greedy algorithm for determining chunk sizes

ensure for each chunk:

delay CAC + delay routing + delay sum < \(\frac{1}{f}\)

save some registers + 1 cycle count

if no solution, insert second register level
for some values of \((w, f)\) discard second register level

\[
Y_{k-1} \quad X_{k-1} \quad Y_3 \quad X_3 \quad Y_2 \quad X_2 \quad Y_1 \quad X_1 \quad Y_0 \quad X_0 \quad c_{in}
\]

\[
R_{k-1} \quad R_3 \quad R_2 \quad R_1 \quad R_0
\]

greedy algorithm for determining chunk sizes

- ensure for each chunk: \(\text{delay}_{CAC} + \text{delay}_{routing} + \text{delay}_{sum} < 1/f\)
for some values of \((w, f)\) discard second register level

- greedy algorithm for determining chunk sizes
  - ensure for each chunk: \(delay_{CAC} + delay_{routing} + delay_{sum} < 1/f\)
Short-Latency Architecture: Optimization

- for some values of \((w, f)\) discard second register level

- greedy algorithm for determining chunk sizes
  - ensure for each chunk: \(delay_{CAC} + delay_{routing} + delay_{sum} < 1/f\)
for some values of \((w, f)\) discard second register level

\[
Y_{k-1} X_{k-1} \ldots R_{k-1}
\]

\[
Y_3 X_3 1 \quad Y_2 X_2 1 \quad Y_1 X_1 1 \quad Y_0 X_0 c_{in}
\]

greedy algorithm for determining chunk sizes
- ensure for each chunk: \(\text{delay}_{\text{CAC}} + \text{delay}_{\text{routing}} + \text{delay}_{\text{sum}} < 1/f\)
- save some registers + 1 cycle count
- if no solution, insert second register level
Adder Generator
VHDL

128 Optimiz. Metric Virtex4 400MHz SRL

Classical

Alternative

Short-Latency

Largest mismatch 4%: good approximation formulas
**Reality Check**

**Adder Generator VHDL**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>SRL</th>
<th>Results</th>
<th>L</th>
<th>R</th>
<th>S</th>
<th>Estimations</th>
<th>L</th>
<th>R</th>
<th>S</th>
<th>Mismatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical</td>
<td>N</td>
<td>128</td>
<td>573</td>
<td>309</td>
<td>128</td>
<td>573</td>
<td>300</td>
<td>0</td>
<td>0</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>318</td>
<td>292</td>
<td>198</td>
<td>318</td>
<td>292</td>
<td>194</td>
<td>0</td>
<td>0</td>
<td>2%</td>
</tr>
<tr>
<td>Alternative</td>
<td>N</td>
<td>222</td>
<td>392</td>
<td>216</td>
<td>223</td>
<td>393</td>
<td>207</td>
<td>0.4%</td>
<td>0.2%</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>352</td>
<td>199</td>
<td>183</td>
<td>352</td>
<td>199</td>
<td>177</td>
<td>0</td>
<td>0</td>
<td>3%</td>
</tr>
<tr>
<td>Short-Latency</td>
<td>N</td>
<td>288</td>
<td>264</td>
<td>216</td>
<td>293</td>
<td>263</td>
<td>214</td>
<td>2%</td>
<td>0.3%</td>
<td>0.9%</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>416</td>
<td>136</td>
<td>216</td>
<td>421</td>
<td>137</td>
<td>211</td>
<td>2%</td>
<td>0.7%</td>
<td>2%</td>
</tr>
</tbody>
</table>

Largest mismatch 4%: good approximation formulas
### Reality Check

**Optimiz. Metric**
- 128
- Generator

**Adder**
- Virtex4
- 400MHz
- SRL

**VHDL**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>SRL</th>
<th>Results</th>
<th></th>
<th>Estimations</th>
<th></th>
<th>Mismatch</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>L</td>
<td>R</td>
<td>S</td>
<td>L</td>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>Classical</td>
<td>N</td>
<td>128</td>
<td>573</td>
<td>309</td>
<td>128</td>
<td>573</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>318</td>
<td>292</td>
<td>198</td>
<td>318</td>
<td>292</td>
<td>194</td>
</tr>
<tr>
<td>Alternative</td>
<td>N</td>
<td>222</td>
<td>392</td>
<td>216</td>
<td>223</td>
<td>393</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>352</td>
<td>199</td>
<td>183</td>
<td>352</td>
<td>199</td>
<td>177</td>
</tr>
<tr>
<td>Short-Latency</td>
<td>N</td>
<td>288</td>
<td>264</td>
<td>216</td>
<td>293</td>
<td>263</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>416</td>
<td>136</td>
<td>216</td>
<td>421</td>
<td>137</td>
<td>211</td>
</tr>
</tbody>
</table>

Largest mismatch 4%: good approximation formulas
## Synthesis Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>200</td>
<td>Spartan3</td>
<td>Y</td>
<td>SLICE</td>
<td>62</td>
<td>4</td>
<td>62</td>
<td>4</td>
<td>76</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-5)</td>
<td>N</td>
<td></td>
<td>110</td>
<td></td>
<td>84</td>
<td></td>
<td>64</td>
<td></td>
<td>41%</td>
</tr>
<tr>
<td>64bit</td>
<td>450</td>
<td>Virtex4</td>
<td>Y</td>
<td>SLICE</td>
<td>96</td>
<td>2</td>
<td>81</td>
<td>2</td>
<td>109</td>
<td>2</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-12)</td>
<td>N</td>
<td></td>
<td>113</td>
<td></td>
<td>82</td>
<td></td>
<td>110</td>
<td></td>
<td>27%</td>
</tr>
<tr>
<td>128bit</td>
<td>450</td>
<td>Virtex4</td>
<td>Y</td>
<td>SLICE</td>
<td>247</td>
<td>5</td>
<td>230</td>
<td>5</td>
<td>258</td>
<td>2</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-12)</td>
<td>N</td>
<td></td>
<td>516</td>
<td></td>
<td>369</td>
<td></td>
<td>258</td>
<td></td>
<td>50%</td>
</tr>
<tr>
<td>128bit</td>
<td>450</td>
<td>Virtex5</td>
<td>Y</td>
<td>REG</td>
<td>322</td>
<td>4</td>
<td>232</td>
<td>4</td>
<td>143</td>
<td>2</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-3)</td>
<td>N</td>
<td></td>
<td>718</td>
<td></td>
<td>525</td>
<td></td>
<td>267</td>
<td></td>
<td>63%</td>
</tr>
</tbody>
</table>

- proposed architectures gain (for examples):
  - 15% less slices, 56% less registers when SRL
  - 50% less slices, 63% less registers when NO SRL

$^1$ compared to the classical implementation
Conclusions

- binary addition pervasive in FPGA applications
- adder operator generator implemented in FloPoCo
- classical pipeline, plus two novel pipelined addition architectures
- selection of the best implementation given the specifications by accurate resource estimation
- performance and resource improvements for operators using it
Conclusions

- binary addition pervasive in FPGA applications
- adder operator generator implemented in FloPoCo
- classical pipeline, plus two novel pipelined addition architectures
- selection of the best implementation given the specifications by accurate resource estimation
- performance and resource improvements for operators using it

In 2010 we can still improve elementary arithmetic operations on FPGA!
Future Work

- applicable to Altera targets, but need different resource estimation formulas
- low-latency architecture could be improved if synthesis tools were more clever
- implement similar resource evaluation for larger components
Future Work

- applicable to Altera targets, but need different resource estimation formulas
- low-latency architecture could be improved if synthesis tools were more clever
- implement similar resource evaluation for larger components

On-going work on ”pipelining made easy” in FloPoCo
Thank you for your attention!

- try it at http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/
  New release 2.1.0 from Aug. 10, 2010
- or, come to us and ask for a demo


