Exploitation du parallélisme et de la taille des données dans la conception d'opérateurs :

Sub-Word Parallelism (SWP)

E. Casseau
INRIA / IRISA
Projet CAIRN

ARCHI 09
Ecole thématique
Pleumeur-Bodou 30 mars – 3 avril 2009

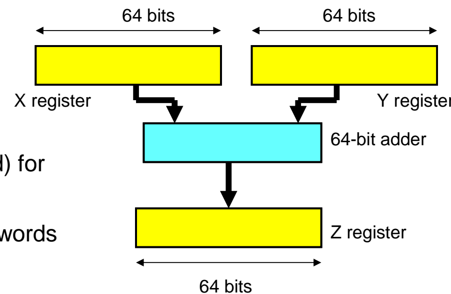
Outline

- Introduction to Sub-Word Parallelism (SWP)
- SWP extensions for general purpose processors
- Basic SWP operator design
  - adder
  - multiplier
- Towards a multimedia dedicated operator

Usual computing resources

- Conventional ALU :
  - word size : - 32 bits
    - 64 bits
    - …
- Basic units designed (optimized) for word sizes: 32/64 bits

$\Rightarrow$ max. efficiency with 32/64-bit words

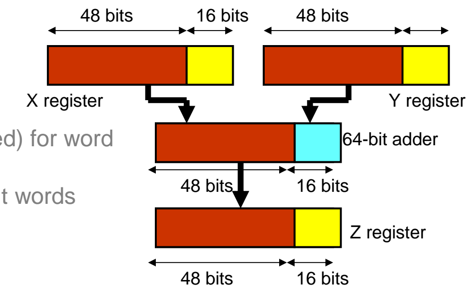


Usual computing resources

- Conventional ALU :
  - word size : - 32 bits
    - 64 bits
    - …
- Basic units designed (optimized) for word sizes: 32/64 bits

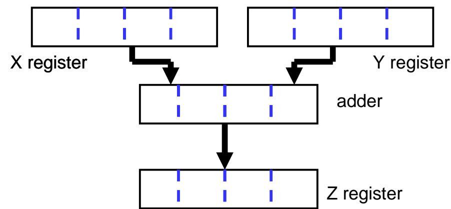
$\Rightarrow$ max. efficiency with 32/64-bit words

- Audio/video/… processing :
  - low precision data: 8/10/12/16-bits
  - conventional ALU
    - low efficiency : under utilization of processor resources
  - wasting: area, power, performance



SWP: Sub Word Parallelism

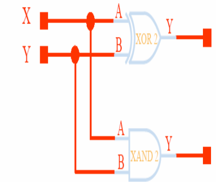
- Goal: efficiency improvement
- ? : avoid wasting of word size resources
- Sub Word Parallelism SWP:
  - subword : small data item contain within a word
  - multiple subwords are packed into one word
  - whole word is processed at the same time :
    - ⇒ simultaneous parallel processing on subwords



SWP: basic example

ADDER

- Ripple Carry adder
  - based on half adders and full adders



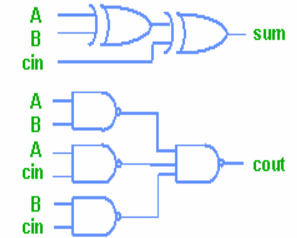
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth Table

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_i = (a_i \oplus b_i)c_{i-1} + a_i b_i$$

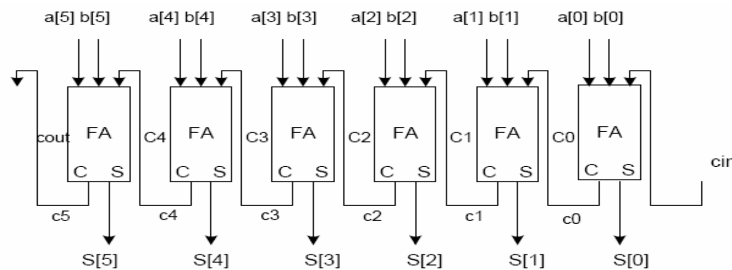
$$s_i = a_i \oplus b_i \oplus c_i$$



SWP: basic example

ADDER

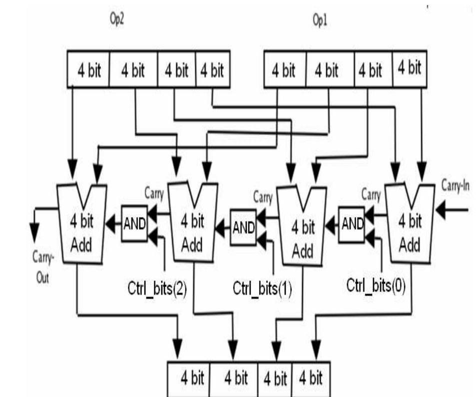
- Ripple Carry adder
  - based on half adder and full adder
  - N-bit full adders are required to add two N-bit operands



SWP: basic example

ADDER

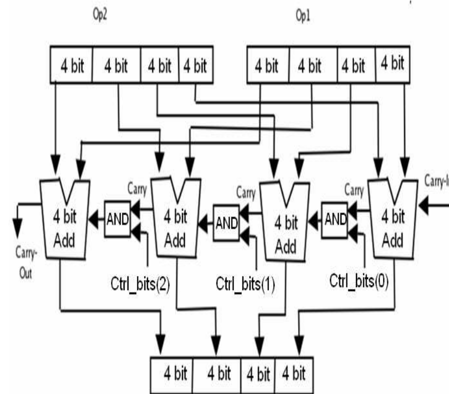
- 16-bit SWP enabled ripple carry adder can perform either (for example):
  - *Four 4-bit additions*
  - *Two 8-bit additions*
  - *One 16-bit additions*



SWP: basic example

ADDER

- 16-bit SWP enabled ripple carry adder can perform either (for example):
  - *Four 4-bit additions*
  - *Two 8-bit additions*
  - *One 16-bit additions*



- BUT: usually not so easy…
  - can SWP be applied to
    - the application ?
    - the operator(s) ?
  - complexity increase ?

SWP

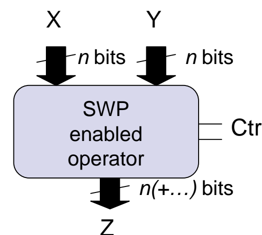
- SWP solves under utilization issues in processors (GPP, media processors, DSP)
- Rather than wasting word oriented datapath, use SWP
- Efficient and flexible solution for media applications
- More efficient use of memory as packed subwords move between memory and processor
- Example: 64-bit word size and 8-bit subword size:
  - 8 subwords a processed per computing cycle
  - only some portions of the application can utilize SWP
  - actual speedup : 8 ?

SWP

- SWP utilizes data level parallelism
  - *k m*-bit subwords are processed in parallel

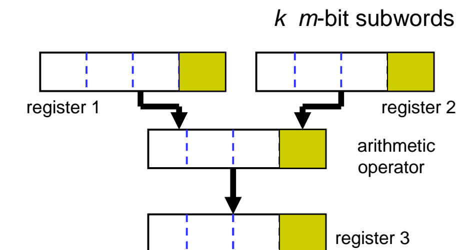
    $k \cdot m \leq n$ n : word size

- SWP is sometimes called *small scale SIMD*
- Applications : low precision data applications (audio, video, …)
- Subword sizes :
  - size of subwords in a word can be different
    but same subword sizes reduce complexity (operator design & use (data management))
  - more subwords leads to more parallelism but increases area & delay



SWP oriented computations

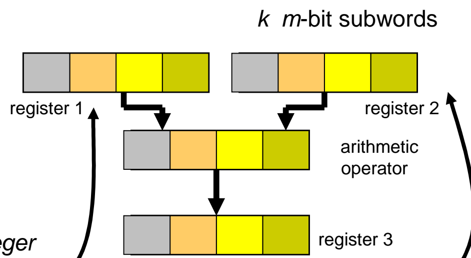
~~x = a + b;~~
~~y = c * d;~~
~~z = a – d;~~



SWP oriented computations

$$y = \sum_{i=1}^{N} a(i) \times b(i)$$

```
y = 0;
For (i =1, j<N+1, i+)
  y = y + a[i] . b[i];
```



Max. efficiency with $N = k \cdot l$, *l: integer*

@ A

a4	a3	a2	a1
a8	a7	a6	a5
		...	a9

@ B

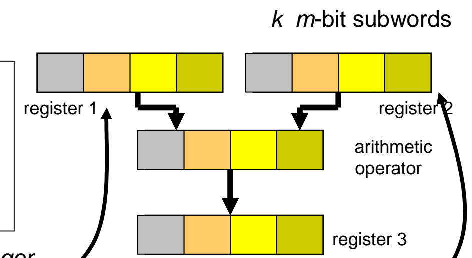
b4	b3	b2	b1
b8	b7	b6	b5
		...	b9

SWP oriented computations

$$\Delta = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} |Img_1(y,x) - Img_2(y,x)|$$

Block matching

```
diff = 0;
For (i =0, i<N, i+)
  For (i =0, j<N, j+)
    diff = diff + abs( a[i,j] - b[i,j]);
```



Max. efficiency with $N = k \cdot l$, *l: integer*

@ A

a03	a02	a01	a00
a07	a06	a05	a04
			...
a13	a12	a11	a10
a17	a16	a15	a14

@ B

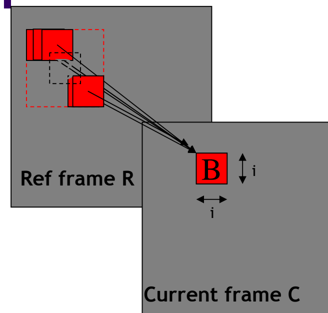
b03	b02	b01	b00
b07	b06	b05	b04
			...
b13	b12	b11	b10
b17	b16	b15	b14

SWP oriented computations

$$\Delta = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} |Img_1(y,x) - Img_2(y,x)|$$

Block matching

```
diff[0] = 0;
For (i =0, i<n, i+)
  For (i =j, j<n, j+)
    diff[i] = diff[i] + abs( a[i,j] - b[i,j]);
```



@ A

a03	a02	a01	a00
a07	a06	a05	a04
			...
a13	a12	a11	a10
a17	a16	a15	a14

@ B

b03	b02	b01	b00
b07	b06	b05	b04
			...
b13	b12	b11	b10
b17	b16	b15	b14

SWP primitives

- Subword parallel primitives are required to exploit data parallelism
- SWP primitives include :
  - basic arithmetic operations :
    - Add, Subtract, Multiply, etc.
  - data management :
    - data alignment before and after certain operation
    - subword arrangement
    - expansion and contraction of data
    - load multiple packed subwords from memory to registers
    - etc.

Loop coding with and without SWP

High-level language loop

```
Short x[200], y[200],z[200],w[200];
Int i;
For(i =0, i<200, i+) {
  Z[i] = x[i] + y[i];
  W[i] = x[i] – y[i];
  }
```

Without SWP Instructions

```
        ldi 199, Ri
Loop:   ldhs,ma    2(Raddrx),Rx
        ldhs,ma    2(Raddry), Ry
        add        Rx, Ry, Rz
        sub        Rx, Ry, Rw
        sths,ma    Rz, 2(Raddrz)
        sths,ma    Rw,2(Raddrw)
        addibf,<   -1, Ri, loop
```

With SWP Instructions

```
        ldi 49, Ri
Loop:   ldds,ma    8(Raddrx),Rx
        ldds,ma    8(Raddry), Ry
        hadd       Rx, Ry, Rz
        hsub       Rx, Ry, Rw
        stds,ma    Rz, 8(Raddrz)
        stds,ma    Rw,8(Raddrw)
        addibf,<   -1, Ri, loop
```


Multimedia extensions for GPP

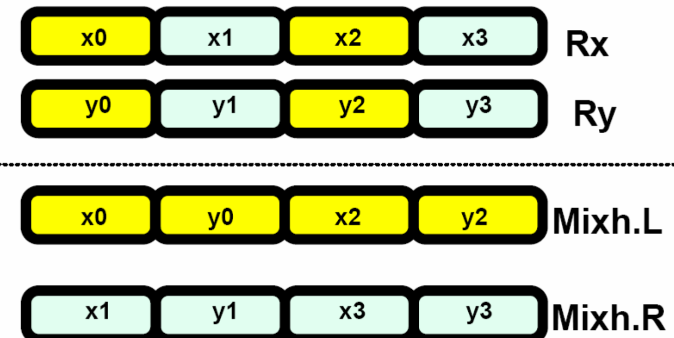
- **1990's :** optimization of **image-processing** programs
  ⇒ SWP …
- To take full advantages of SWP, SWP-dedicated instructions are required
- Instruction sets including SWP instructions :
  - MAX-1 (1994) and MAX-2 (1996) added to HP's PA-RISC
  - MMX added to Intel Pentium (1997)(4x16bits) (~~floating point unit~~)
    puis SSE, SSE-2,3,4 added to IA-32, IA-64 (floating point unit)
  - VIS (1995) added to Sun's Sparc V9 (UltraSparc, SPARC64)
  - Altivec added to Motorola's PowerPC (PPC G4 1999)

Example: MAX-2 instruction set

- MAX-2 is multimedia acceleration extensions implemented in PA_RISC-2.0 (64 bits).
- MAX-2 supports **subword sizes of 16-bits**
  - Parallel Add (modulo or saturation)
  - Parallel Subtract (modulo or saturation)
  - Parallel Shift Right (1,2 or 3 bits) and Add } multiplies subwords by
  - Parallel Shift Left (1,2 or 3 bits) and Add } integer/fractional data
  - Parallel Average
  - Parallel Shift Right (n bits)
  - Parallel Shift Left (n bits)
  - Mix
  - Permute

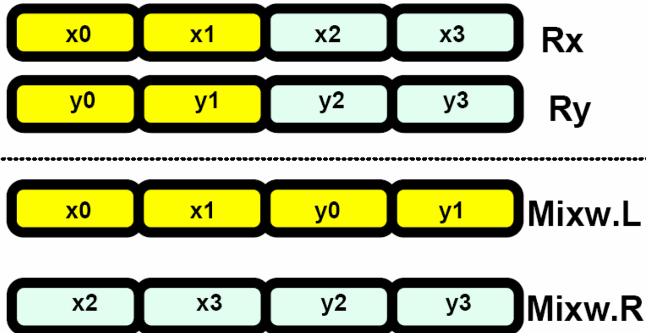
MAX-2

- MIX interleaves subwords from 2 source registers



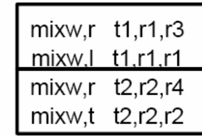
MAX-2

• MIX of larger (32-bit) subwords



MAX-2

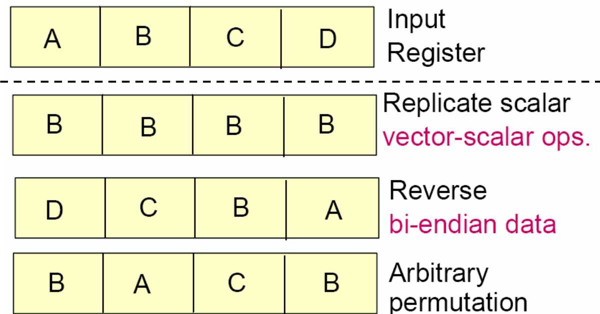
- Matrix transpose : 4x4 matrices: 2 steps, 8 instructions



- n.n matrices : n.log(n) instructions

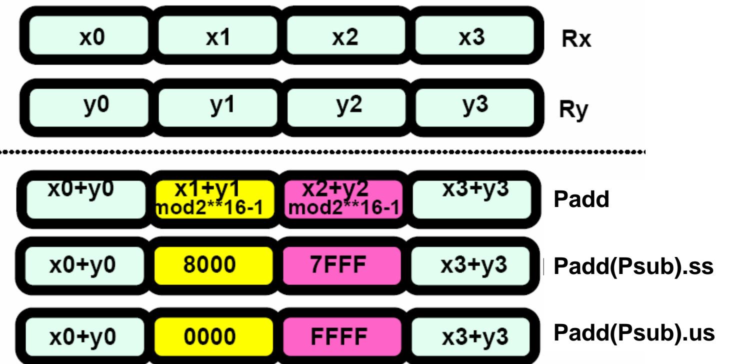
MAX-2

• Permute instruction



MAX-2

• Parallel subword Add/Sub: with modulo arithmetic or signed saturation or unsigned saturation



MAX-2

$$\Delta = \sum_{y=0}^{N-1} \sum_{x=0}^{N-1} |Img_1(y,x) - Img_2(y,x)|$$

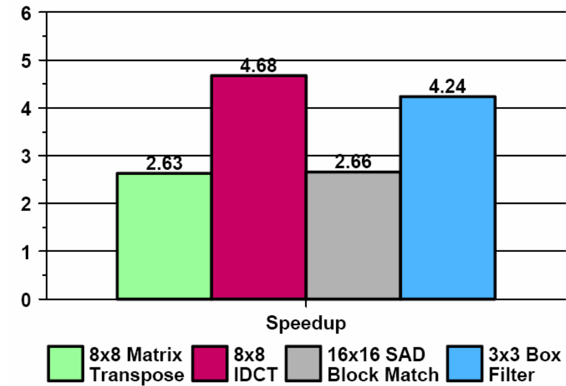
- SAD using saturation arithmetic

r1 [85 | 20 | 100 | 5] r2 [45 | 211 | 0 | 33]

Instruction	Operands	Result				
hsub,us	r1, r2, r3	r3	40	0	100	0
hsub,us	r2, r1, r4	r4	0	191	0	28
hadd	r3, r4, r5	r5	40	191	100	28

MAX-2

- Execution time speedup (PA8000 with MAX-2 vs without MAX-2)



SWP in DSP chips: examples

- TigerSHARC (Analog Devices)
  - combine VLIW and SIMD
  - each of the two datapaths can processes :
    - Eight 8-bit operations
    - Four 16-bit operations
    - Two 32-bit operations
- TMS320C64x (Texas Instruments)
  - fixed point DSP
  - 2 clusters
    - dual 16-bit and quad 8-bit SIMD additions and comparisons
    - dual 16-bit and quad 8-bit SIMD multiplications

SWP multimedia operator design

- Conventional subword sizes :
  - uniform arithmetic relation with subword sizes:
    - 8, 16, 32-bits etc. (MAX-2, MMX, Altivec, ...)
  - *complexity of operators* is less **but** *under utilization* of resources for multimedia applications:
    - pixel's sizes: 8, 10, 12 and sometimes 16-bits
- $\Rightarrow$ multimedia oriented subword sizes : 8, 10, 12, 16
  - no uniform arithmetic relation with subword sizes
  - *complexity of operators* is increased **but** *resource utilization* for multimedia applications is better

SWP multimedia operator design

- Synopsys SIMD IPs :
  - *VHDL/Verilog* signed or unsigned SIMD *adder*, SIMD *adder with carry*, SIMD *multiplier*

Parameter	Values	Description
width	≥ 2, must be a multiple of $2^{no_confs-1}$	Word length
no_confs	≥ 2	Number of configurations

no_confs : number of possible configurations

conf : 2^{conf} partitions of size ($width/2^{conf}$)

$\Rightarrow$ subword sizes : uniform arithmetic relation 2 / 4 / 8 / 16 / 32 …

```
Example: width = 32, no_confs = 3
    conf = 0:
      z[31: 0] = a[31: 0] + b[31: 0]
    conf = 1:
      z[31:16] = a[31:16] + b[31:16]
      z[15: 0] = a[15: 0] + b[15: 0]
    conf = 2:
      z[31:24] = a[31:24] + b[31:24]
      z[23:16] = a[23:16] + b[23:16]
      z[15: 8] = a[15: 8] + b[15: 8]
      z[ 7: 0] = a[ 7: 0] + b[ 7: 0]
```


SWP multimedia operator design

- $\Rightarrow$ multimedia oriented subword sizes : 8, 10, 12, 16
  - "good" choice : *word size = 40-bits*
    - supported subword sizes: 8, 10, 12, 16-bits
    - gives better efficiency for different pixel sizes

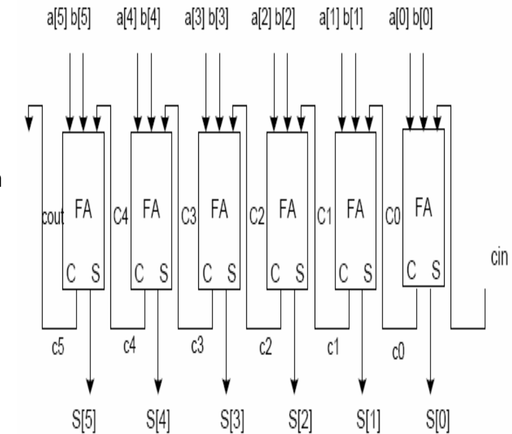
	conventional(8/16/32)		dedicated(8/10/12/16/40)	
	#	use ratio%	#	use ratio%
a(8) OP b(8)	4	100	5	100
a(10) OP b(10)	*2*	*62*	4	100
a(12) OP b(12)	*2*	*75*	3	90
a(16) OP b(16)	2	100	2	80
a(32) OP b(32)	1	100	*1*	*80*
a(40) OP b(40)	0	X	1	100

ADD architectures

- Adders are used in:
  - addition
  - subtraction
  - multiplication
  - division
- Different types of adders:
  - ripple carry adder
  - carry look ahead adder
  - carry save adder
  - conditional sum adder
- Speed of the processing system heavily depends upon these fundamental units.

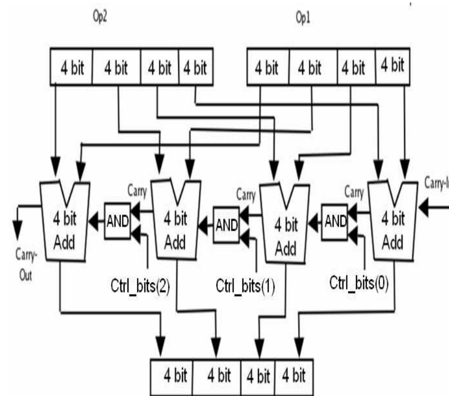
Ripple Carry Adder (RCA)

- Conventional way of adding two numbers.
- N-bit full adders are required to add two N-bit operands
- Slowest adder (carry ripples from the LSB to MSB)
- Takes minimum area
- RCA is used when
  - minimum hardware is required
  - speed is not critical
- Speed is linear with word length O(N)



SWP enabled Ripple Carry Adder

- This example: 16-bit SWP enabled ripple carry. It can perform either:
  - *Four 4-bit additions*
  - *Two 8-bit additions*
  - *One 16-bit additions*



Carry Look Ahead adder (CLA)

- Speed of RCA get worst when number of bits increases
- Remedy
  - use Carry look ahead adder
  - CLA calculate carries in advance
- Carry is calculated using:
  - carry generate logic
  - carry propagate logic
- Generation of all carries simultaneously using CLA generator

Case	x_i	y_i	$x_i + y_i$	c_{i+1}	Comment
1	0	0	0	0	kill (stop) carry-in
2	0	1	1	c_i	propagate carry-in
	1	0	1	c_i	propagate carry-in
3	1	1	2	1	generate carry-out

Case 1 (Kill): $k_i = x_i' y_i' = (x_i + y_i)'$

Case 2 (Propagate): $p_i = x_i \oplus y_i$

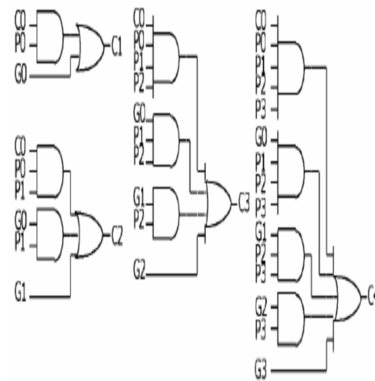
Case 3 (Generate): $g_i = x_i y_i$

Then

$$c_{i+1} = g_i + p_i c_i = x_i y_i + (x_i \oplus y_i) c_i$$

Carry Look Ahead adder

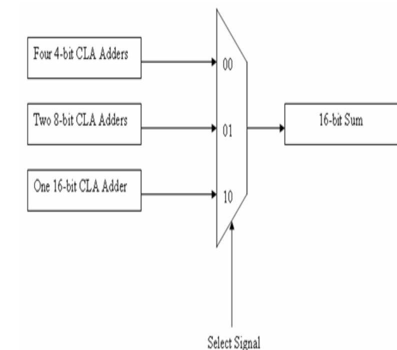
- Must generate carry when
  - Ai = Bi = 1
  - Gi=Ai Bi
- Carry propagate:
  - Pi=Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Ci+1=Gi+ Pi Ci
  - Si=Ci xor Pi
- Re-express the carry logic
  - C1 = G0 + P0C0
  - C2 = G1 + P1C1
  - = G1 + P1(G0 + P0C0)
  - = G1 + P1G0 + P0P1C0



Carry logic gets costly with the increase in word length.

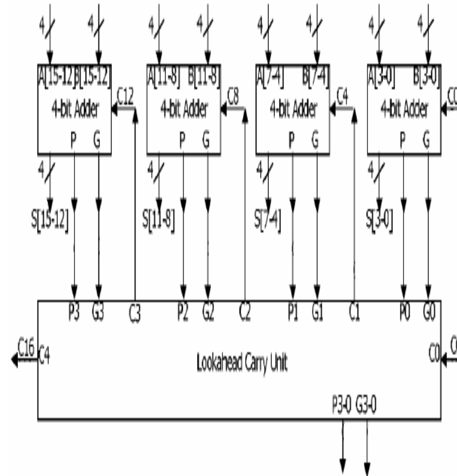
SWP enabled CLA adder

- Implementation
  - Multiple subword sizes can not be easily combined :
    - Separate implementation of blocks
    - Sharing of components between blocks is performed by the synthesis tool
- This example: SWP enabled 16-bit CLA which performs one of the following operation:
  - *Four 4-bit additions*
  - *Two 8-bit additions*
  - *One 16-bit additions*

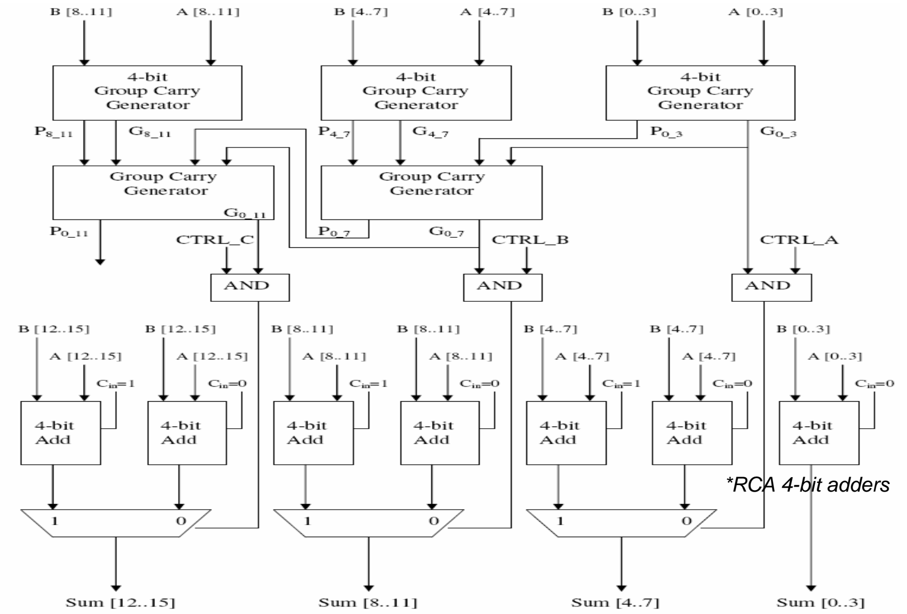


Group CLA adder

- Disadvantage of CLA:
  - carry logic gets more complicated for more than 4-bits
- Remedy:
  - implement CLA adders as 4-bit modules.
- Each 4-bit adder gives group propagate (PG) and generate (GG) signal:
  - PG = P3.P2.P1.P0
  - GG = G3 + P3G2 + P3.P2.G1 + P3.P2.P1.G0

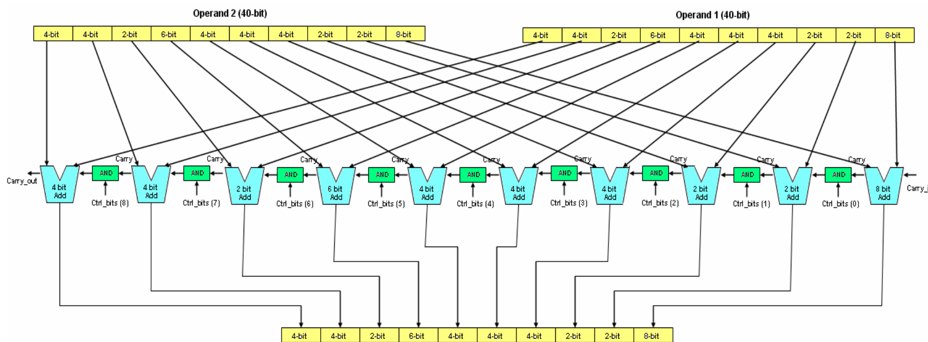


SWP enabled group CLA adder



SWP multimedia ADD operator

- Adders between the control logic can be:
  - ripple carry adder RCA
  - carry look ahead adder CLA
- Selected subword size determines the control bits :
  - propagate/block the carry



SWP multimedia ADD operator

		90 nm CMOS ASIC			130 nm CMOS ASIC			FPGA VirtexII		
		Nand Gates	CP (ns)	Gates x CP (norm)	Nand Gates	CP (ns)	Gates x CP (norm)	CLBs	CP (ns)	Gates x CP (norm)
40-bit RCA ADD	Simple	291	2.69	1	281	8.10	1	82	25.2	1
	SWP	345	4.31	1.90	347	10.1	1.54	78	27.6	1.04
	Overhead (%)	19	60	90	23	25	54	-5	10	4

		90 nm CMOS ASIC			130 nm CMOS ASIC			FPGA VirtexII		
		Nand Gates	CP (ns)	Gates x CP (norm)	Nand Gates	CP (ns)	Gates x CP (norm)	CLBs	CP (ns)	Gates x CP (norm)
40-bit Group CLA ADD	Simple	372	2.31	1	372	4.11	1	74	15.7	1
	SWP	444	1.92	0.99	463	4.52	1.37	81	15.2	1.06
	Overhead (%)	19	-17	-1	24	10	37	9	-3	6

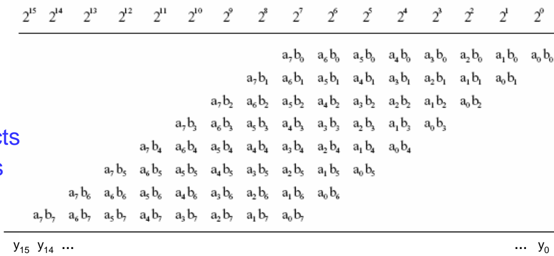
Subword sizes: 8,10,12,16(40) bits

Synthesis tools:
Synopsys / Mentor Graphics Precision RTL

- Compared to SWP RCA:
  - area of SWP CLA is more
  - CP of SWP CLA is less
  - efficiency of SWP CLA is less

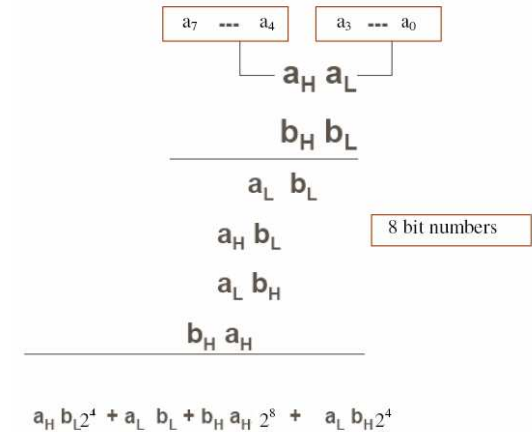
Multiplication

- Input:
  - N-bit multiplier
  - M-bit multiplicand
- Partial Products:
  - generation of partial products
  - left shift the partial products
- Final Product:
  - addition of shifted partial products
  - (N+M) bit final product



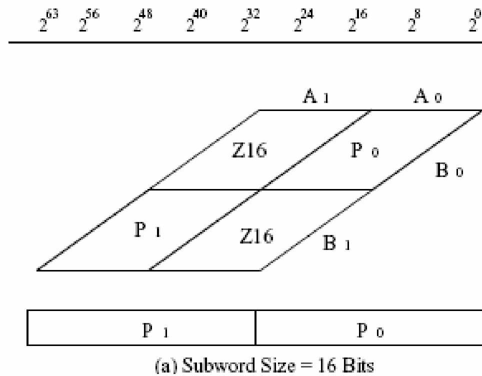
Implementation principle of a basic SWP multiplier

- Word size: 8 bits
- Subword size: 4-bits
- Multiplier
  - a_low & a_high = 4-bits
- Multiplicand
  - b_low & b_high = 4-bits
- Subword size = 4
  - 1st partial product
  - 4th partial product
- Word size = 8
  - addition of all PPs



Implementation principle of a basic SWP multiplier

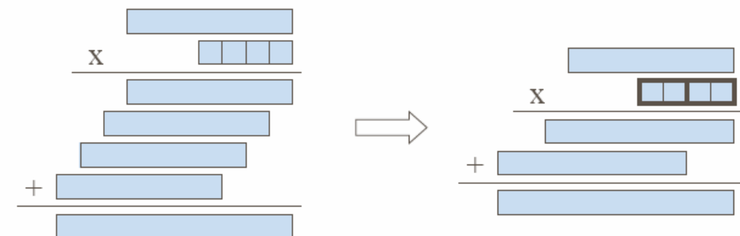
- Word size: 8 bits
- Subword size: 4-bits
- Multiplier
  - a_low & a_high = 4-bits
- Multiplicand
  - b_low & b_high = 4-bits
- Subword size = 4
  - 1st partial product
  - 4th partial product
- Word size = 8
  - addition of all PPs



(a) Subword Size = 16 Bits

Booth recoding

- Basic idea: reduce the number of partial products to reduce the number of accumulations
- Radix-2 : partial product = (multiplicand) x {0, 1}
- Radix-4 : partial product = (multiplicand) x {00, 01, 10, 11}
  - Instead of multiplying with single bit, we multiply with two bits hence making partial products half



SWP enabled multiplier

- SWP enabled *booth multiplier* design :
  - Multiple subword sizes can not be easily combined :
    - separate implementation of blocks
    - blocks share components when possible

- **Synthesis results** for both ASIC and FPGA technologies
  - Basic multiplier :

	Nand gates (CLB)	CP	Nand gates x CP
without SWP	x1	x1	1
with SWP	x1,7	x1,05	**1,8**

  - Booth multiplier :

	Nand gates (CLB)	CP	Nand gates x CP
without SWP	x0,7	x1,6	1,12
with SWP	x1,2	x1,65	**2**

SWP multiplier by Krithivasan & Schulte

[ ] S. Krithivasan, M.J. Schulte, *Multiplier Architectures for Media Processing,* Asilomar Conference on signals, systems and computers, vol.2, pp 2193-2197, 2003.

- Avoids the detection an suppression of carries across subword boundaries

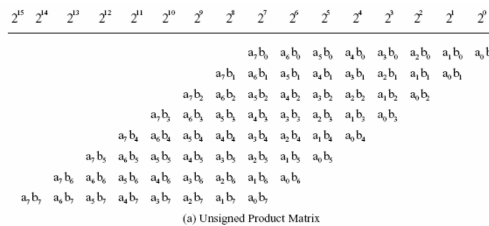
- Designed to perform in parallel:
  - One 32 X 32
  - Two 16 X 16
  - Four 8 X 8

- Supports operands in both:
  - unsigned
  - 2's complement

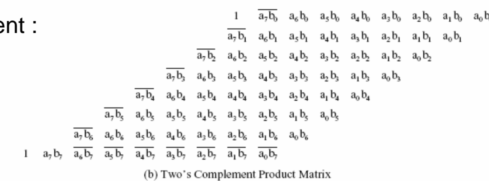
SWP multiplier by Krithivasan & Schulte

Ex.: word size: 8-bits, no subword

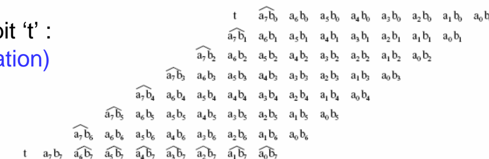
- Fig (a) shows PPs for unsigned :


(a) Unsigned Product Matrix

- Fig (b) shows PPs for 2's complement :
  - 2n-2 PPs bits are inverted
  - '1' added in column n
  - '1' added in column 2n-1

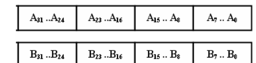

(b) Two's Complement Product Matrix

- Fig (c) supports both using control bit 't' :
  - t = '1' (2's complement multiplication)
  - t = '0' (unsigned multiplication)



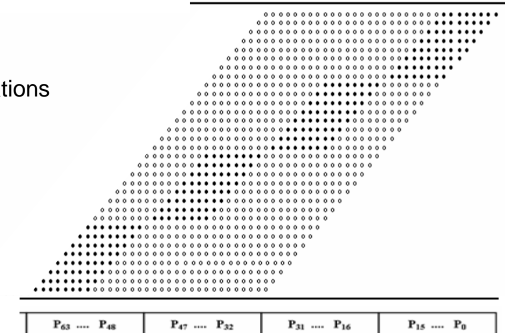
SWP multiplier by Krithivasan & Schulte

Ex.: word size : 32-bits, subword: 8-bit



- When subword size is 8:
  - four 8 X 8 unsigned multiplications
  - lot of PP are set to '0'

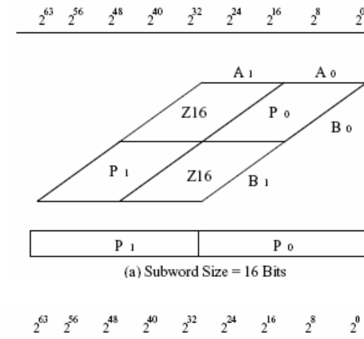
- To set unwanted PPs to '0'
  - AND gates are required



SWP multiplier by Krithivasan & Schulte

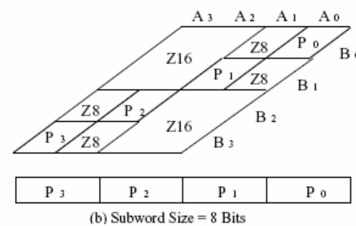
Ex.: word size : 32-bits, subwords: 8,16 bits

- Fig (a) : two 16 X 16 unsigned multiplications
  - Z16 regions are set to zero



(a) Subword Size = 16 Bits

- Fig (b) : four 8 X 8 unsigned multiplications
  - Z16 and Z8 regions are set to zero

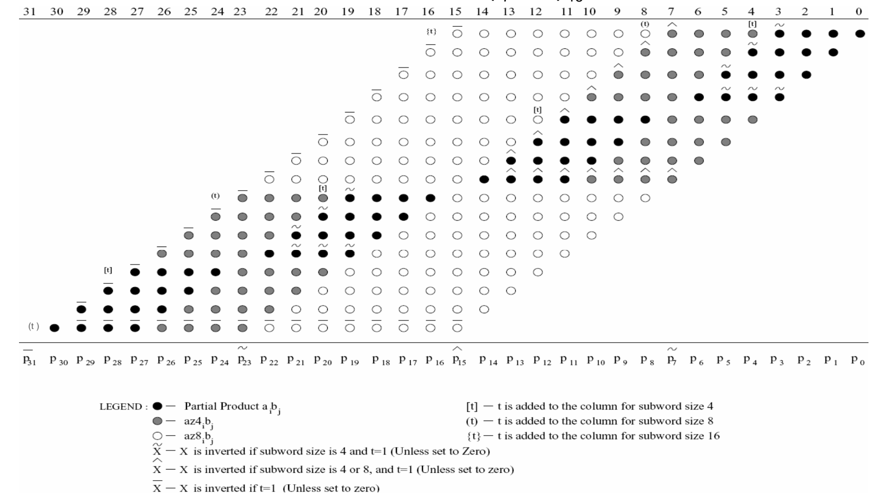


(b) Subword Size = 8 Bits

SWP multiplier by Krithivasan & Schulte

Ex.: word size : 16-bits
subwords: 4,8 bits

- e.g. when doing two 8X8 :
  - 't' are added to column 8 and 24
  - PPs bits formed by a_7 or b_7 and a_{15} or b_{15} are inverted
  - Product bits p_7 and p_{15} are inverted.



SWP multiplier

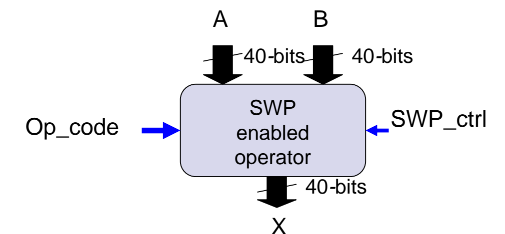
- based on Krithivasan & Schulte's multiplier but word size is 40-bits and subword sizes are 8,10,12,16-bits

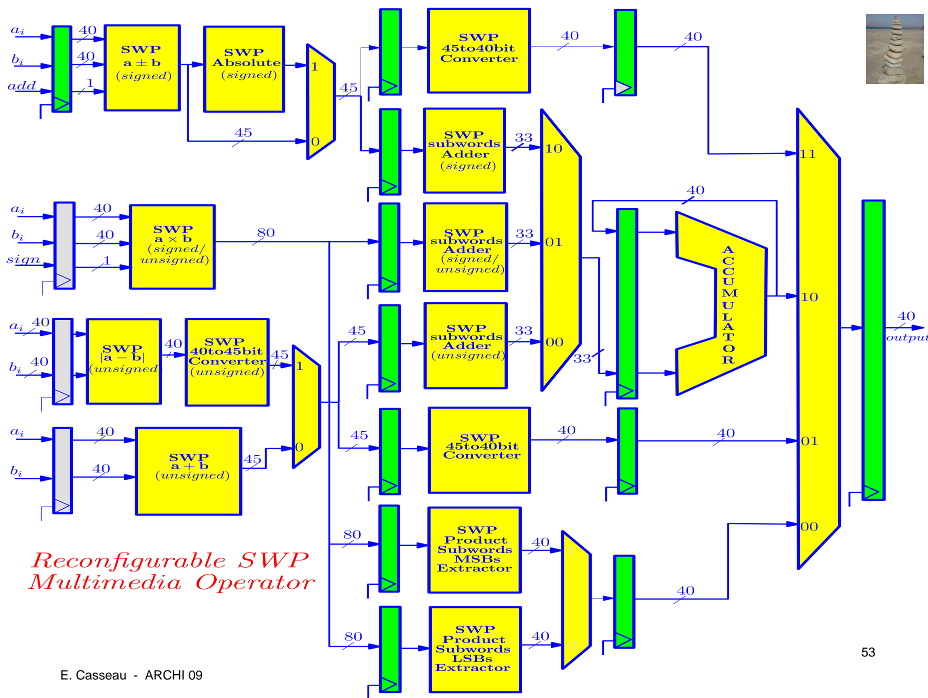
		90 nm CMOS ASIC			130 nm CMOS ASIC			FPGA VirtexII		
		Nand Gates	CP (ns)	Gates x CP (norm)	Nand Gates	CP (ns)	Gates x CP (norm)	CLBs	CP (ns)	CLBs x CP (norm)
40-bit Mult	Simple	14518	6.07	1	10532	14.0	1	917	19.7	1
	SWP	15099	7.38	1.26	11081	15.0	1.13	1505	21.4	1.78
	Overhead (%)	4	22	26	5	7	13	64	9	78

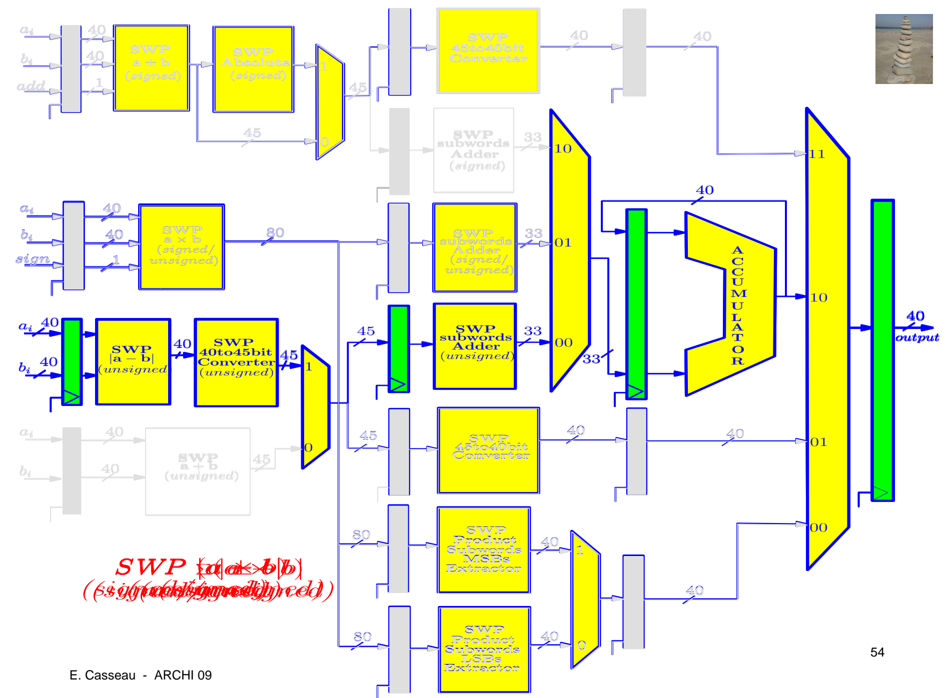
- SWP control logic is small compared to multiplier complexity
  - maximum area overhead on ASIC is 5%
  - maximum CP overhead on ASIC is 22%
- Coordination between ASIC and FPGA results:
  - in FPGA resources are CLBs rather than gates (AND, NOT…)

Reconfigurable SWP operator

- Operator word size : 40-bits
  - subword sizes : 8, 10, 12, 16-bits.
- Basic arithmetic operations:
  - $(a_i \pm b_i)$ Signed data
  - $|a_i \pm b_i|$ Signed data
  - $(a_i \times b_i)$ Signed/unsigned data
  - $|a_i - b_i|$ Unsigned data
  - $(a_i + b_i)$ Unsigned data
- Complex operations:
  - $\sum(a_i \pm b_i)$ Signed data
  - $\sum |a_i \pm b_i|$ Signed data
  - $\sum(a_i \times b_i)$ Signed/unsigned data
  - $\sum |a_i - b_i|$ Unsigned data
  - $\sum(a_i + b_i)$ Unsigned data
- Combination of complex operations:
  - $\sum|(a_i + b_i)| + \sum|(a_i - b_i)|$ Signed data
  - $\sum(a_i + b_i) + \sum(a_i - b_i)$ Signed data
  - $\sum(a_i + b_i) + \sum|(a_i - b_i)|$ Unsigned data
  - $\sum|(a_i + b_i)| + \sum|(a_i - b_i)| + \sum(a_i + b_i) + \sum(a_i - b_i)$ Signed data
  - etc.

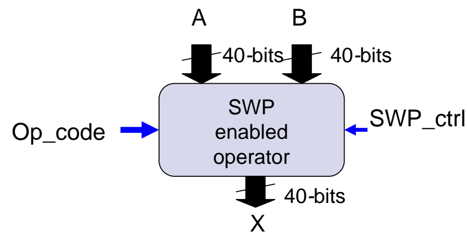






Reconfigurable SWP operator

- Operator word size : 40-bits
  - subword sizes : 8, 10, 12, 16-bits.

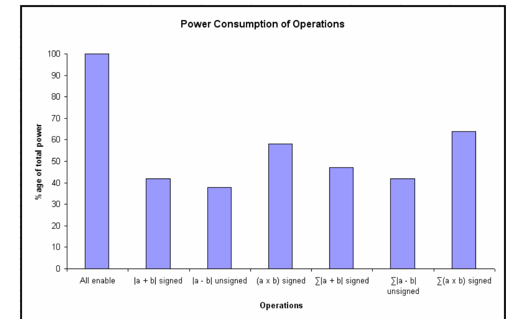


- **Synthesis results**

	Nand gates (CLBs)	CP	
130nm tech.	30.000	7 ns	(mult.:15.000 gates)
90 nm tech.:	31.000	10 ns	(mult.:11.000 gates)
FPGA VirtexII	2.800	17 ns	(mult.:1.500 CLBs)

Reconfigurable SWP operator

Operation Type	Data Format	Power (mW)	%age of total power
All enable	--	6.93	100
|a + b|	signed	2.9	42
|a - b|	unsigned	2.6	38
(a x b)	signed	4.0	58
∑|a + b|	signed	3.3	47
∑|a - b|	unsigned	2.9	42
∑(a x b)	signed	4.4	64



- Clock period = 10 ns
- ASIC tech :130nm
- Maximum power consumed by ∑(a x b) operation (64 % of total power)

References

- J. Fridman, *Sub-Word Parallelism in Digital Signal Processing* IEEE Signal Processing Magazine, pp 27-35, March 2000.
- S. Krithivasan and M.J. Schulte, *Multiplier Architectures for Media Processing,* Thirty seventh Asilomar Conference on signals, systems and computers, vol.2, pp 2193-2197, 2003.
- P Corsonello, S Perri, M.A Iachino1 and G Cocorullo *Variable Precision Arithmetic Circuits for FPGA Based Multimedia Processors,* IEEE Transactions on very large scale integration (VLSI) systems, VOL. 12, No.9, September 2004.
- A. Danysh, D. Tan, *Architecture and Implementation of a Vector/SIMD Multiply-Accumulate Unit*, IEEE Computer Society, volume 54, Issue 3, pp 284–293, March 2005.
- J. Wakerly, *Digital Design*, 3rd Edition, Prentice Hall, Upper Saddle River, NJ, 2000.
- M. O. Cheema, O. Hammami, *Customized SIMD Unit Synthesis for System on Programmable Chip – A Foundation for HW/SW Partitioning with Vectorization,* IEEE Design Automation conference, , pp 54– 60, Jan 2006.
- C. Brunelli, P.Salmela, J.Takala and J.Nurmi, *A Flexible Multiplier for Media Processing*, IEEE workshop on Design and Implementation, pp 70-74, Nov 2005.

Acknowledgment

- Design and syntheses have been performed by Shafqat Khan, PhD student, INRIA/IRISA