Topic 4

Arithmetic Circuits

Peter Cheung
Department of Electrical & Electronic Engineering
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/
E-mail: p.cheung@imperial.ac.uk

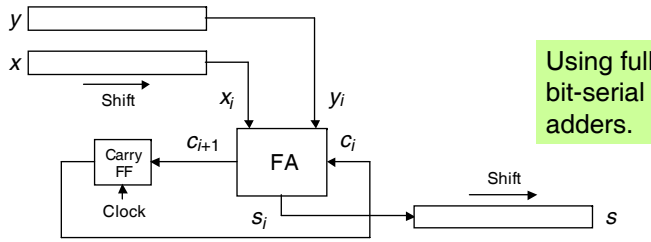
About this Topic

- Comparison of adder architectures on FPGAs
- Multiple operands addition
- Basic multipliers
- Booth recoding multipliers
- Fixed point vs Floating Point
- Floating point Unit architectures
- Example: FIR and IIR filter implementations
- References
  - "Computer Arithmetic", B. Parhami, OUP
  - "Computer Arithmetic Algorithms", I. Koren, AK Peters

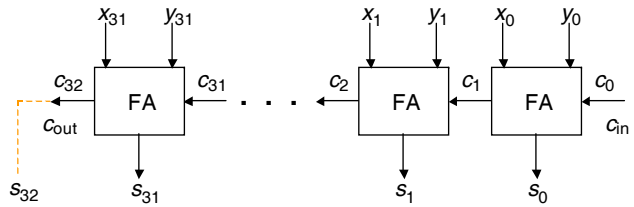
Different adder architectures

- Revision on last year's digital electronics II course (http://www.ee.ic.ac.uk/hp/staff/dmb/courses/dig2/5_Adder.pdf)
- Common adder architectures are:
  - Ripple carry adder
  - Carry lookahead adder
  - Carry skip (or carry select) adder
  - Carry save adder
  - Parallel prefix adder (Brent & Kung's)

Basic Ripple Carry Adder



(a) Bit-serial adder.



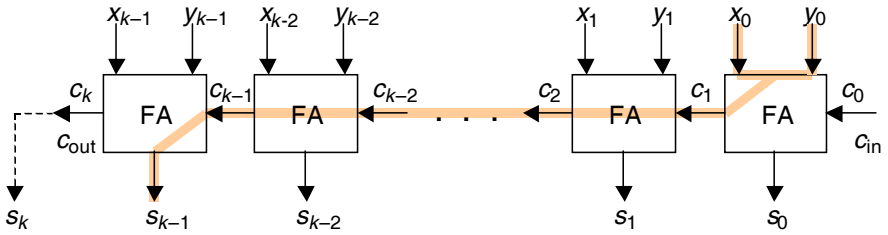
(b) Ripple-carry adder.

Using full-adders in building bit-serial and ripple-carry adders.

Source: Parhami

Critical Path Through a Ripple-Carry Adder

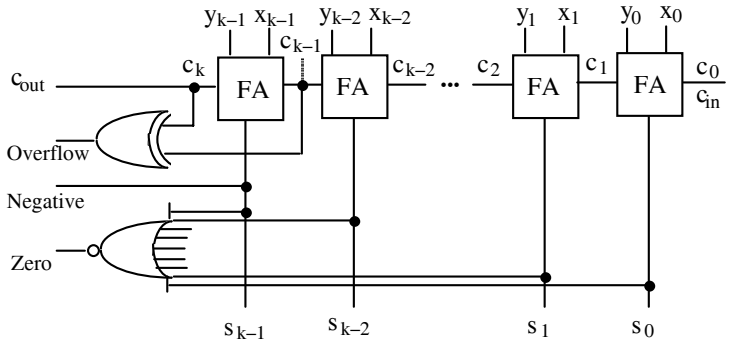
$$T_{\text{ripple-add}} = T_{FA}(x,y \rightarrow c_{out}) + (k-2) \times T_{FA}(c_{in} \rightarrow c_{out}) + T_{FA}(c_{in} \rightarrow s)$$



Critical path in a *k*-bit ripple-carry adder.

Source: Parhami

Adder Conditions and Exceptions



Two's-complement adder with provisions for detecting conditions and exceptions.

$$\text{overflow}_{\text{2's-compl}} = x_{k-1}\, y_{k-1}\, s_{k-1}' \vee x_{k-1}'\, y_{k-1}'\, s_{k-1}$$

$$\text{overflow}_{\text{2's-compl}} = c_k \oplus c_{k-1} = c_k\, c_{k-1}' \vee c_k'\, c_{k-1}$$

Source: Parhami

Saturating Adders

Saturating (saturation) arithmetic:

When a result's magnitude is too large, do not wrap around; rather, provide the most positive or the most negative value that is representable in the number format

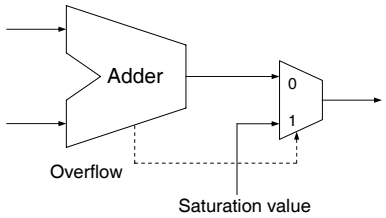
Example – In 8-bit 2's-complement format, we have:
120 + 26 → 18 (wraparound); 120 $+_{sat}$ 26 → 127 (saturating)

Saturating arithmetic in desirable in many DSP applications

Designing saturating adders

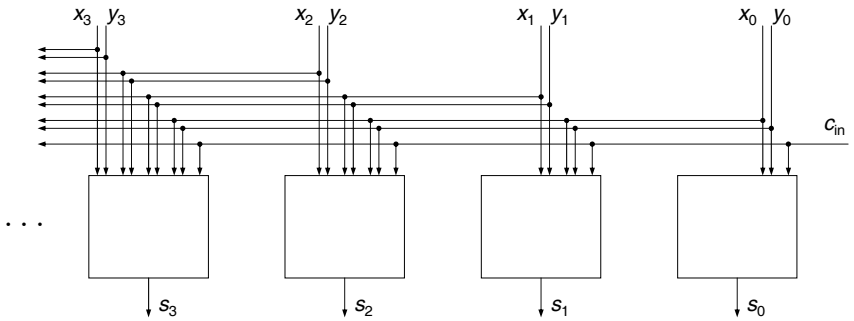
Unsigned (quite easy)

Signed (only slightly harder)



Source: Parhami

Full Carry Lookahead



Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits

Source: Parhami

Unrolling the Carry Recurrence

Recall the *generate g*, *propagate p* signals:

Signal	Radix r	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

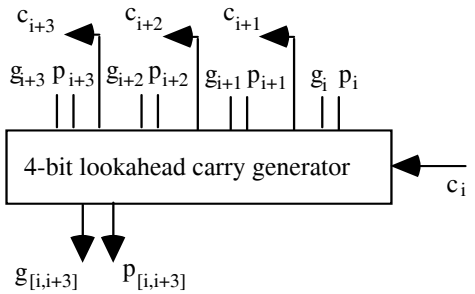
$$
\begin{aligned}
c_i &= g_{i-1} + c_{i-1}\, p_{i-1} \\
&= g_{i-1} + (g_{i-2} + c_{i-2}\, p_{i-2})\, p_{i-1} \\
&= g_{i-1} + g_{i-2} p_{i-1} + c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} + g_{i-2} p_{i-1} + g_{i-3} p_{i-2} p_{i-1} + c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} + g_{i-2} p_{i-1} + g_{i-3} p_{i-2} p_{i-1} + g_{i-4} p_{i-3} p_{i-2} p_{i-1} + c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}
$$

Source: Parhami

Carry-Lookahead Adder Design

Block *generate* and *propagate* signals

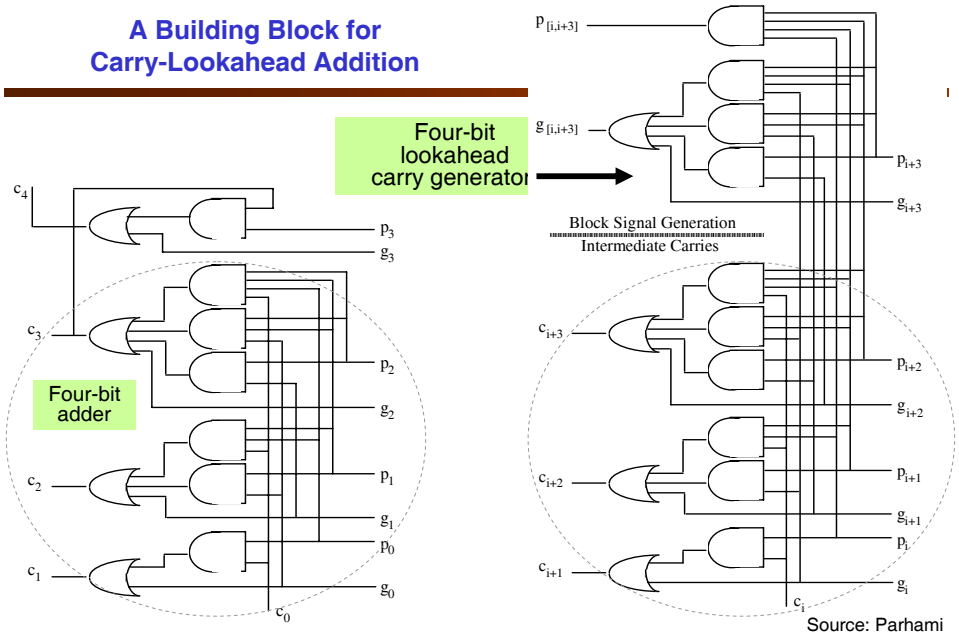
$$
\begin{aligned}
g_{[i,i+3]} &= g_{i+3} + g_{i+2} p_{i+3} + g_{i+1} p_{i+2} p_{i+3} + g_i\, p_{i+1} p_{i+2} p_{i+3} \\
p_{[i,i+3]} &= p_i\, p_{i+1} p_{i+2} p_{i+3}
\end{aligned}
$$



Schematic diagram of a 4-bit lookahead carry generator.

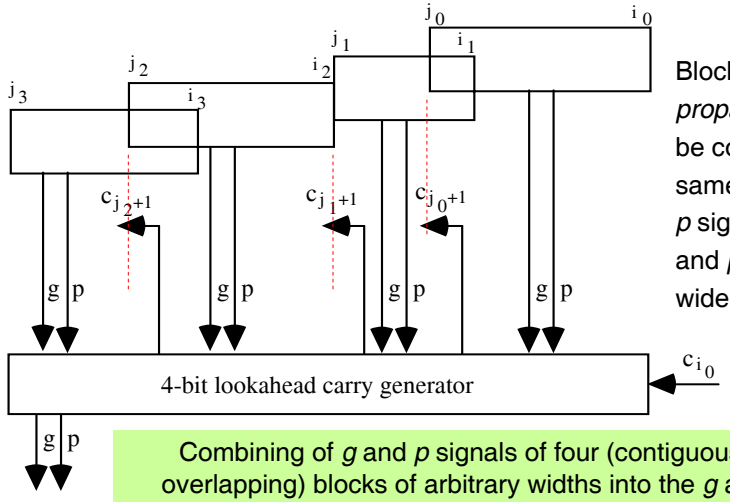
Source: Parhami

A Building Block for Carry-Lookahead Addition



Source: Parhami

Combining Block *g* and *p* Signals

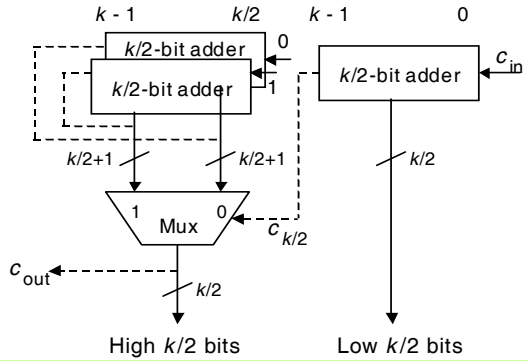


Block *generate* and *propagate* signals can be combined in the same way as bit *g* and *p* signals to form *g* and *p* signals for wider blocks

Combining of *g* and *p* signals of four (contiguous or overlapping) blocks of arbitrary widths into the *g* and *p* signals for the overall block $[i_0, j_3]$.

Source: Parhami

Carry-Select Adders

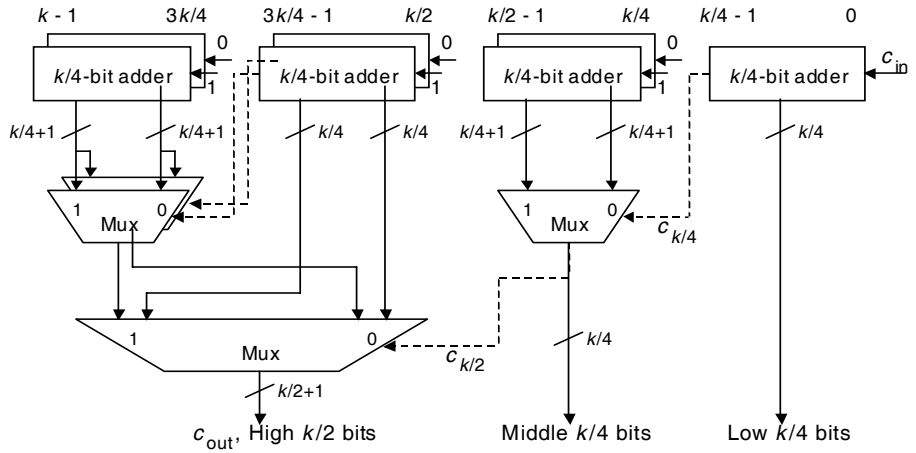


Carry-select adder for k-bit numbers built from three $k/2$-bit adders.

$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$
$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

Source: Parhami

Multilevel Carry-Select Adders



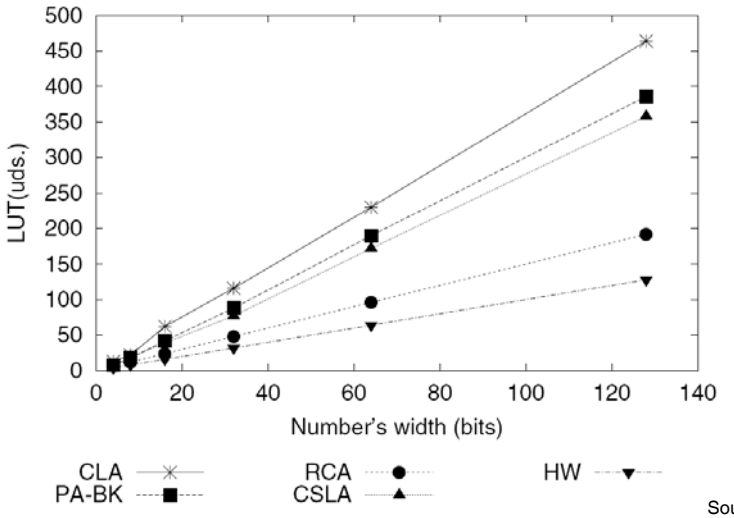
Two-level carry-select adder built of $k/4$-bit adders.

Source: Parhami

Comparison between adders on modern FPGAs

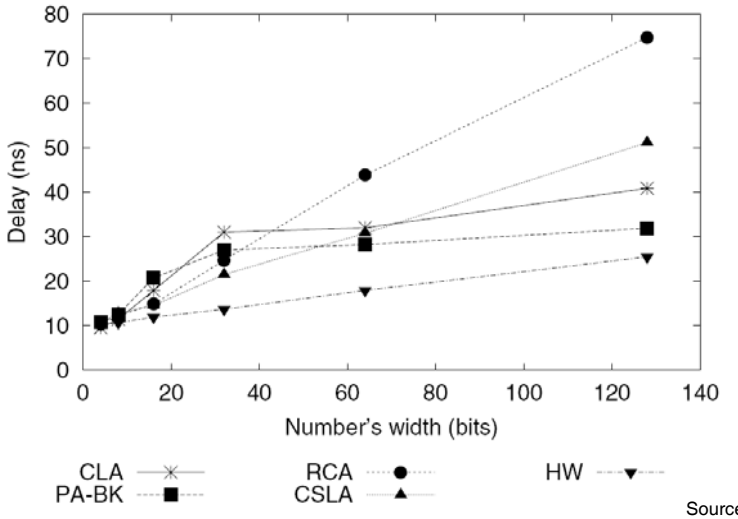
- Sacristan, Rodella & Diaz, "*Comparison of addition structures synthesis over commercial FPGAs*", International Conf. on Design & Test, 2006 Page(s):413 - 417
- Compare ripple carry adder (RCA), carry lookahead adder (CLA), carry select adder (CSLA), Brent&Kung parallel prefix adder (PA-BK) and finally not specifying any structure and let the synthesis tool decide!
- Use Altera Stratix II and Xilinx Virtex-4 (not latest, but pretty recent).
- Result summary:
  - Mostly as expected, faster means larger
  - Surprising, synthesis tools does the best: both fast and small!!
  - Morale – at low level, difficult to beat modern synthesis tools
- Results shown in the next four slides.

Results for Stratix II – Area



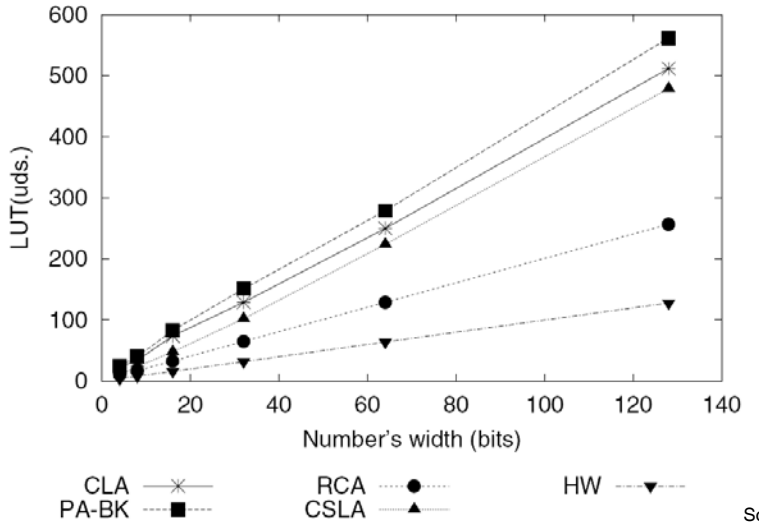
Source: Sacristan

Results for Stratix II – Delay



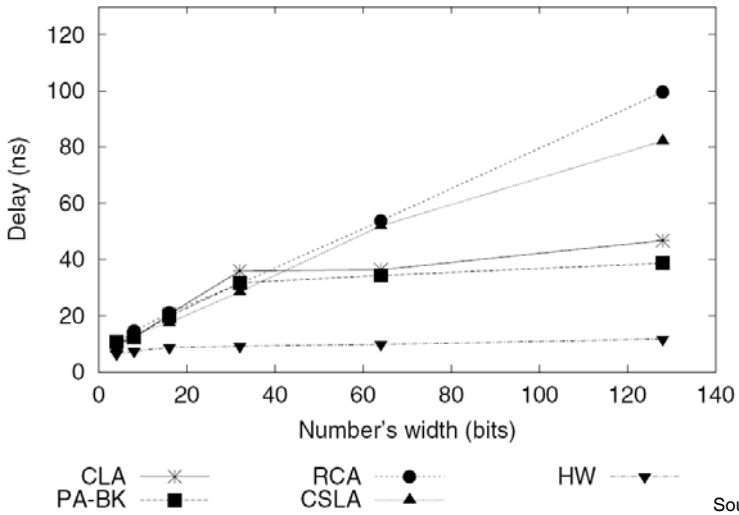
Source: Sacristan

Results for Virtex 4 – Area



Source: Sacristan

Results for Virtex-4 – Delay



Source: Sacristan

Multipliers and DSP Blocks

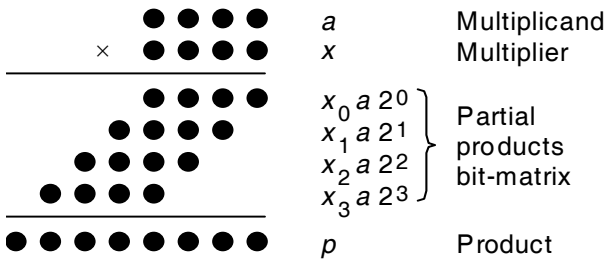
- Remember that both Altera and Xilinx FPGAs have embedded multipliers with accumulators etc.
- This part of the lecture will look at some of the common multiplier hardware (i.e. what such embedded multiplier circuits might look like).
- We will also consider application of FPGA embedded multiplier for FIR Filter implementations.
- Topics to cover are:
  - Basic multipliers
  - Booth recoded multipliers
  - Array multipliers
  - FIR Filter Compiler

Multiplication of two 4-bit unsigned numbers

Notation:

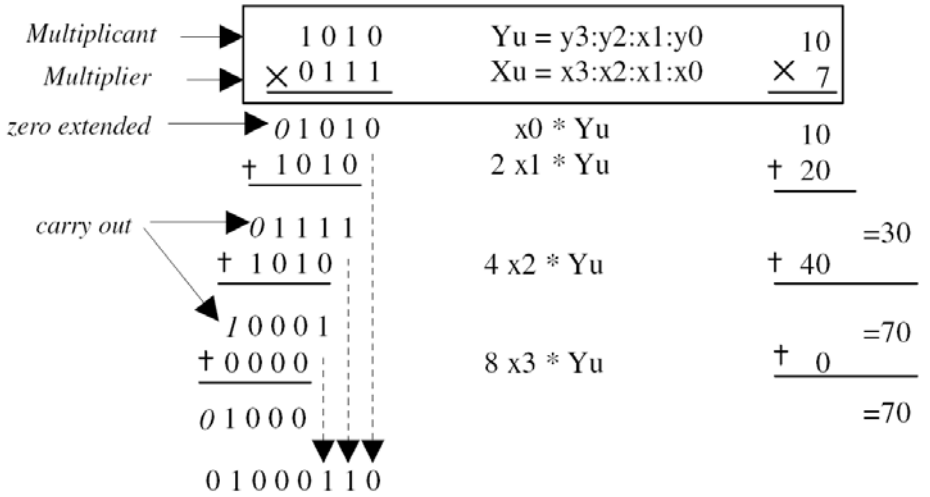
a	Multiplicand	$a_{k-1}a_{k-2} \ldots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \ldots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \ldots p_3p_2p_1p_0$

Initially, we assume unsigned operands

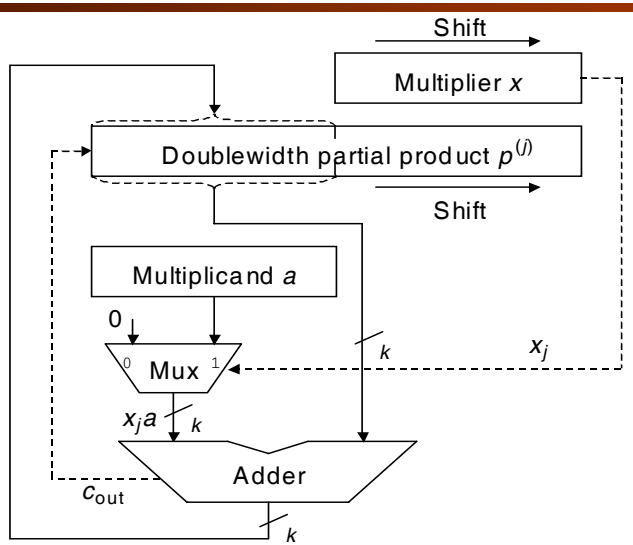


Source: Parhami

An example

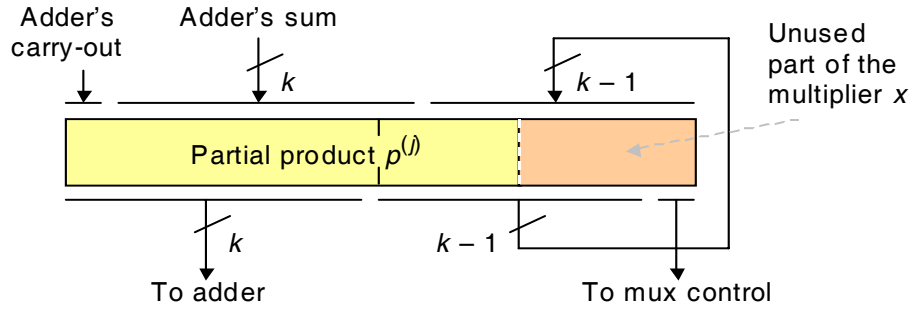


Basic Sequential Multipliers



Source: Parhami

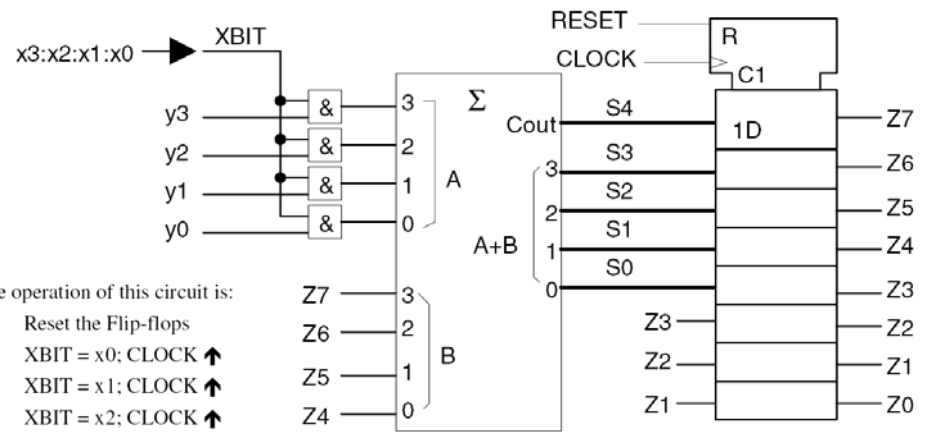
Performing Add and Shift in One Clock Cycle



Combining the loading and shifting of the double-width register holding the partial product and the partially used multiplier.

Source: Parhami

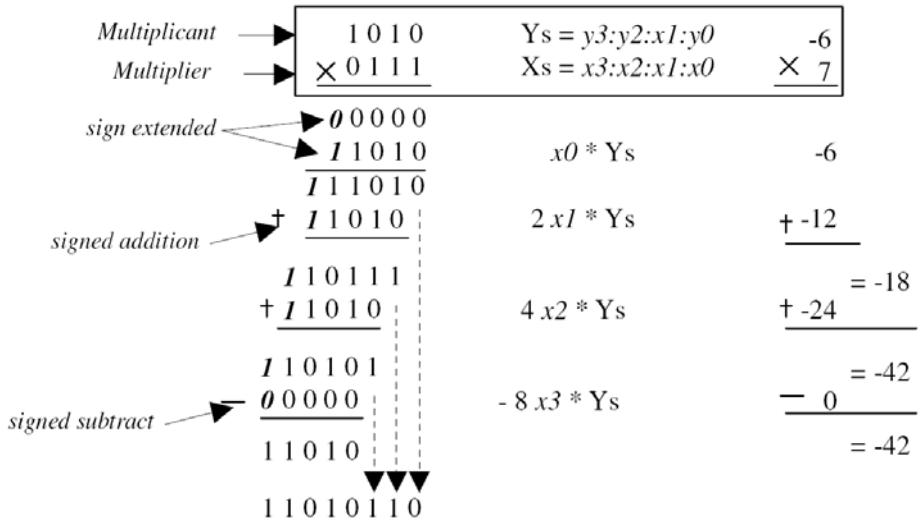
Example of a detail 4x4 unsigned sequential multiplier



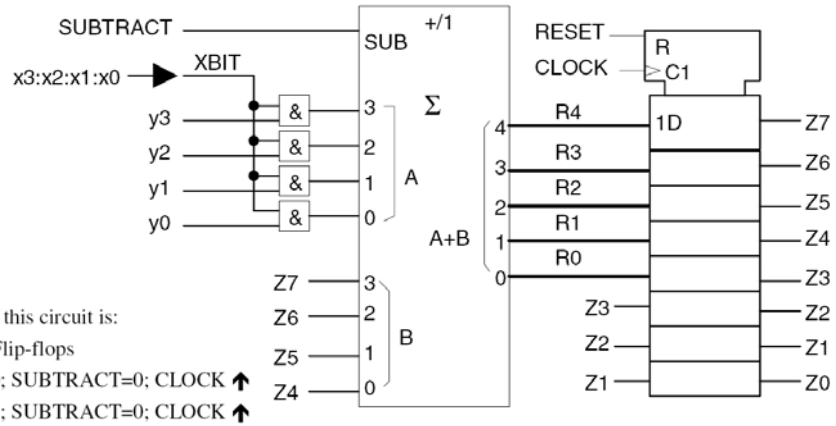
The operation of this circuit is:

- Reset the Flip-flops
- XBIT = x0; CLOCK ↑
- XBIT = x1; CLOCK ↑
- XBIT = x2; CLOCK ↑
- XBIT = x3; CLOCK ↑
- Answer: Z7:0 = Xu * Yu

2's complement signed multiplication



4x4 sequential signed multiplier circuit



The operation of this circuit is:

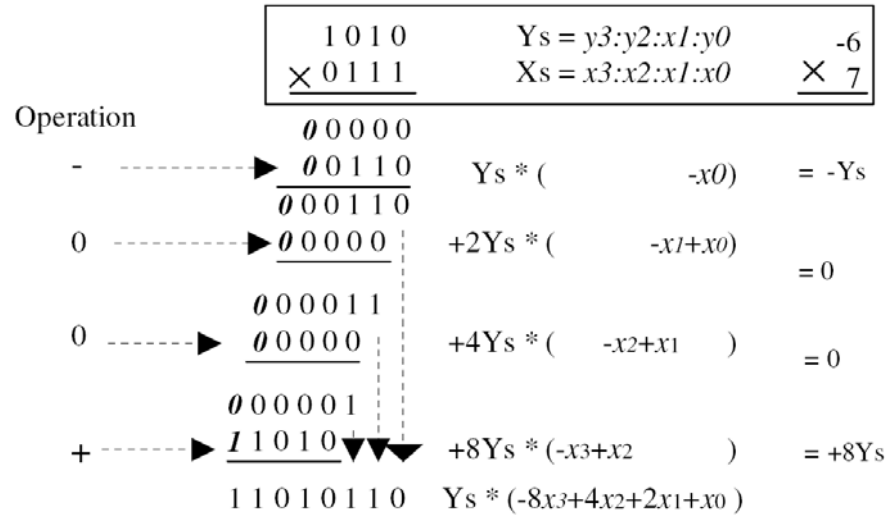
- Reset the Flip-flops
- XBIT = x0; SUBTRACT=0; CLOCK ↑
- XBIT = x1; SUBTRACT=0; CLOCK ↑
- XBIT = x2; SUBTRACT=0; CLOCK ↑
- XBIT = x3; SUBTRACT=1; CLOCK ↑
- Answer: Z7:0 = Xs * Ys

Recoded Multiplier – Booth Algorithm (1)

Instead of treating the MSB differently from all other bits, it is possible to rearrange the binary bits and code them differently. ***Booth Algorithm*** is one of many algorithms that group together a number of bits in the multiplier and perform a ***recoding*** of the binary bits before the actual addition/subtraction. The table above shows how the Booth algorithm work:

- At each stage, we add $Ys * 2^i * (-x_i + x_{i-1})$
- We assume that $x_{-1} = 0$

Recoded Multiplier – Booth Algorithm (1)



Proof of Booth Algorithm

$$\sum_{i=0}^{N-1} 2^i(-x_i + x_{i-1})$$

Booth Algorithm does this

$$= -\sum_{i=0}^{N-1} 2^i x_i + \sum_{i=0}^{N-1} 2^i x_{i-1}$$

$$= -\sum_{i=0}^{N-1} 2^i x_i + \sum_{i=-1}^{N-2} 2^{i+1} x_i$$

$$= -2^{N-1}x_{N-1} - \sum_{i=0}^{N-2} 2^i x_i + \sum_{i=0}^{N-2} 2^{i+1} x_i + 2^0 x_{-1}$$

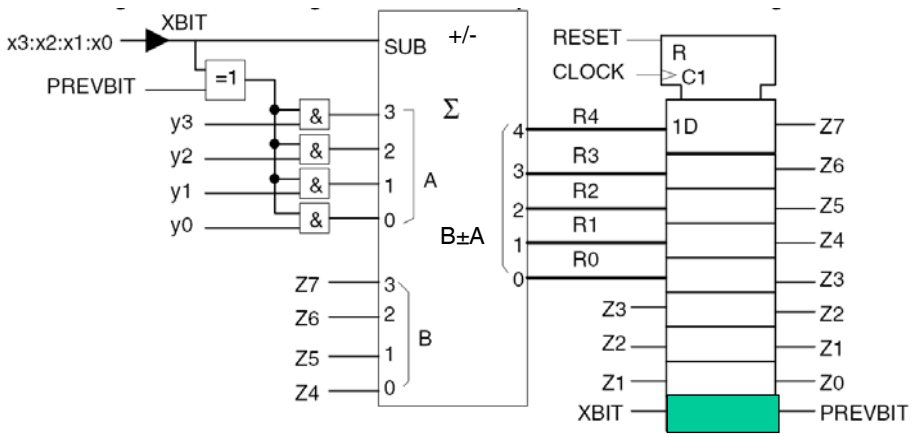
$$= -2^{N-1}x_{N-1} + \sum_{i=0}^{N-2} (-2^i + 2^{i+1}) x_i$$

$$= -2^{N-1}x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i$$

2's complement rep of x

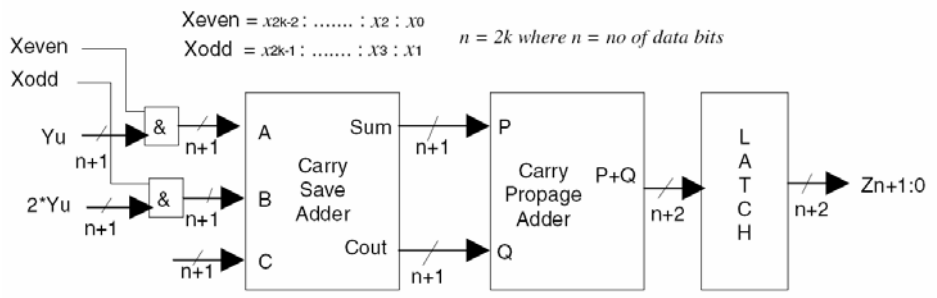
x_i	x_{i-1}	$(-x_i+x_{i-1})$	Comments
0	0	0	Do nothing
0	1	+1	Add Ys
1	0	-1	Subtract Ys
1	1	0	Do nothing

Sequential Booth Multiplier



Multi-bit sequential multiplier

All the circuits considered so far handle only one bit multiplication at each clock cycle. There are no reasons why we could not deal with two (or more) bits at a time.



Modified Booth Algorithm (2 bits at a time)

Booth Algorithm: bit i $2^i(-x_i + x_{i-1})$

bit i+1 $2^{i+1}(-x_{i+1} + x_i)$

Modified Booth: bit i & i+1 $2^i(-x_i + x_{i-1}) + 2^{i+1}(-x_{i+1} + x_i)$

$= 2^i(-2x_{i+1} + x_i + x_{i-1})$

x_{i+1}	x_i	x_{i-1}	$-2x_{i+1}+x_i+x_{i-1}$	Comments
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Modified Booth Recoding (2 bits at a time)

XBIT1	XBIT0	PREVBIT	Mult. factor
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

We can derive the following equations from the above table

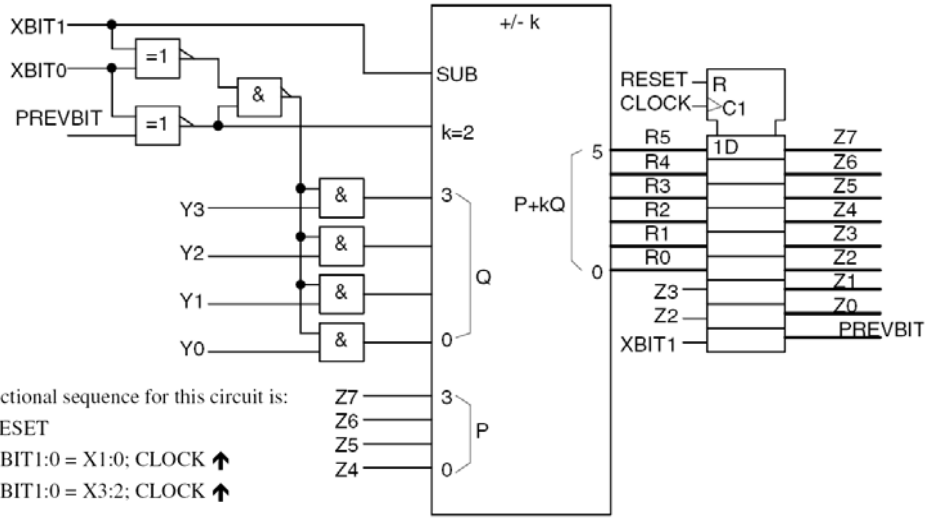
$SUB = XBIT1$

$DOUBLE = \overline{XBIT0 \oplus PREVBIT}$

$ZERO = (XBIT1 = XBIT0 = PREVBIT)$

$= \overline{XBIT1 \oplus XBIT0} \bullet DOUBLE$

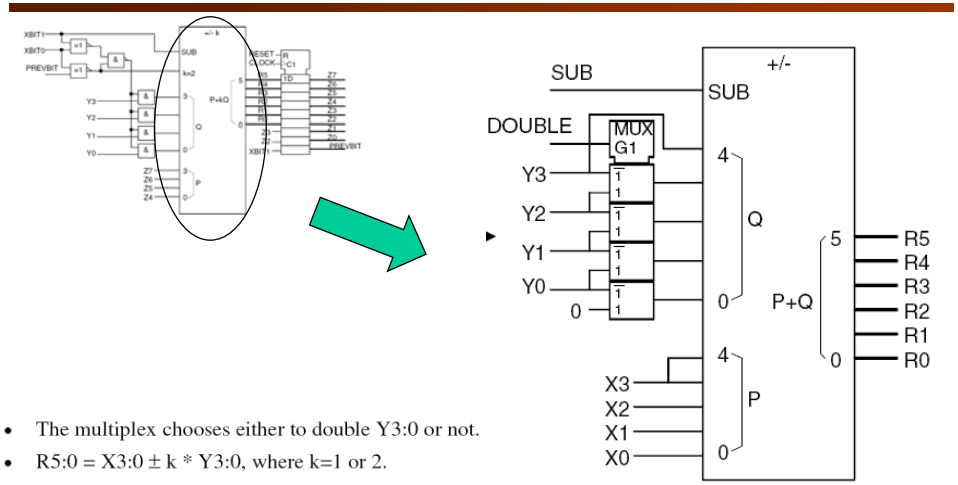
Modified Booth Multiplier Circuit



The functional sequence for this circuit is:

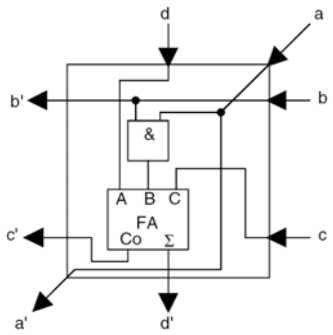
- RESET
- XBIT1:0 = X1:0; CLOCK ↑
- XBIT1:0 = X3:2; CLOCK ↑

Modified Booth Multiplier Circuit



- The multiplex chooses either to double Y3:0 or not.
- R5:0 = X3:0 ± k * Y3:0, where k=1 or 2.
- We need 6 bit adder output to accommodate the answer without overflow.
- X3:0 is sign-extended to make it 5 bits, the same as kY.

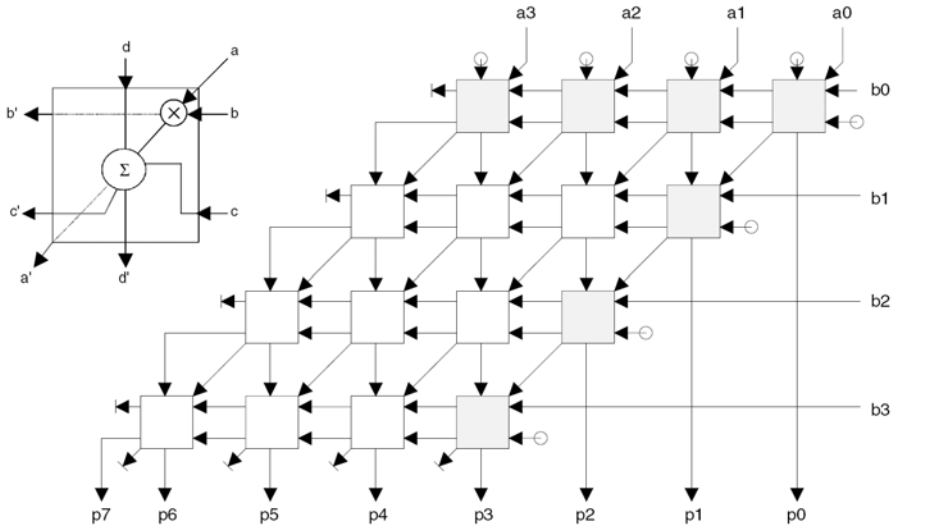
Array Multiplier



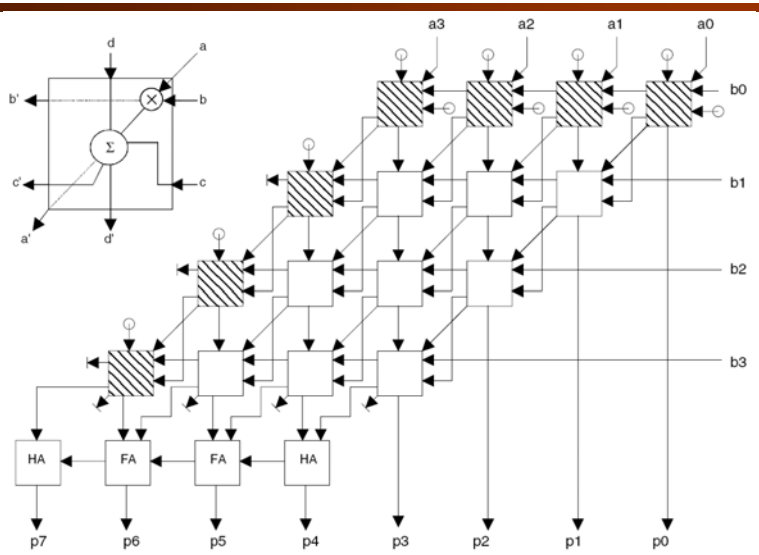
$$a' = a$$
$$b' = b$$
$$c' = a \cdot b \cdot d + a \cdot b \cdot c + c \cdot d$$
$$d' = (a \cdot b \oplus d) \oplus c$$

	a_3	a_2	a_1	a_0	= A
X	b_3	b_2	b_1	b_0	= B
	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_3b_2	a_2b_2	a_1b_2	a_0b_2	
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3	
p_7	p_6	p_5	p_4	p_3	p_2 p_1 p_0 P = A x B

Array Multiplier – obvious, but slow version

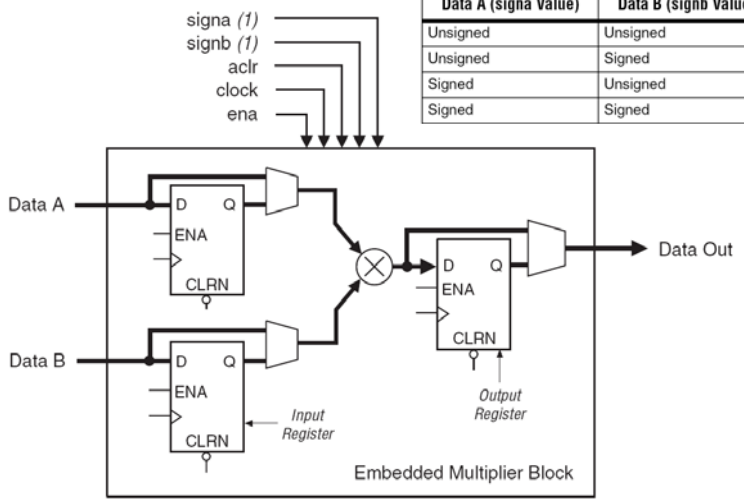


Array Multiplier – using carry-save adders



Embedded Multipliers in Altera Cyclone II (1)

Data A (signa Value)	Data B (signb Value)	Result
Unsigned	Unsigned	Unsigned
Unsigned	Signed	Signed
Signed	Unsigned	Signed
Signed	Signed	Signed



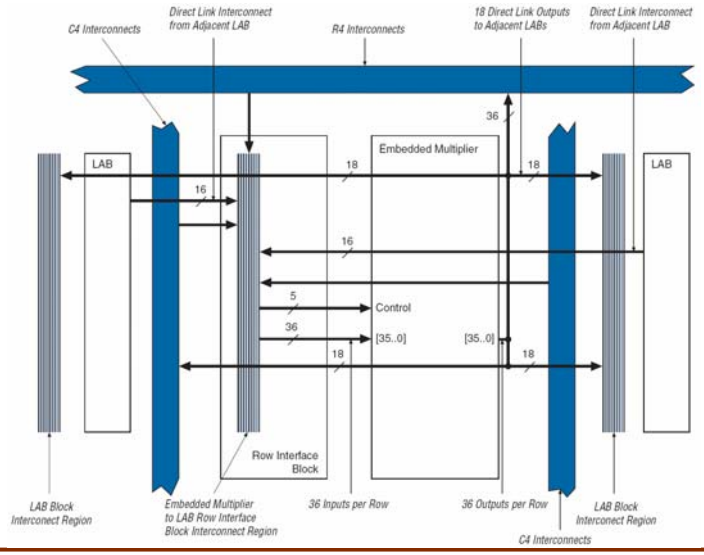
Source: ALTERA

Embedded Multipliers in Altera Cyclone II (2)

Multiplier Mode	Description
18-bit Multiplier	An embedded multiplier can be configured to support a single 18 × 18 multiplier for operand widths up to 18 bits. All 18-bit multiplier inputs and results can be registered independently. The multiplier operands can accept signed integers, unsigned integers, or a combination of both.
9-bit Multiplier	An embedded multiplier can be configured to support two 9 × 9 independent multipliers for operand widths up to 9-bits. Both 9-bit multiplier inputs and results can be registered independently. The multiplier operands can accept signed integers, unsigned integers or a combination of both. There is only one `signa` signal to control the sign representation of both data A inputs and one `signb` signal to control the sign representation of both data B inputs of the 9-bit multipliers within the same dedicated multiplier.

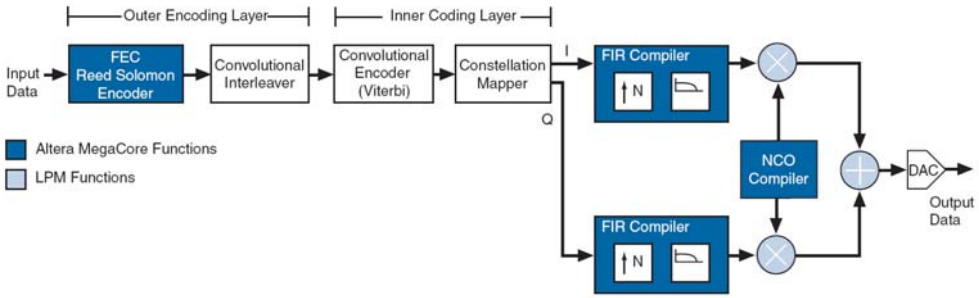
Source: ALTERA

Embedded Multipliers in Altera Cyclone II (3)



Source: ALTERA

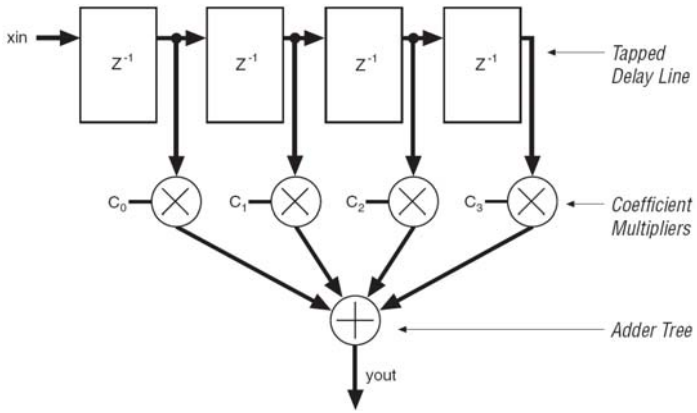
Application of Multipliers: Typical DSP System



- Altera and Xilinx provide FIR filter compiler support.
- These examples are taken from Altera's "FIR Compiler User's Guide".
- MegaCore functions pre-designed core (large modules).
- LPM Functions are parameterised building blocks (e.g. adder, multiplier)

Source: ALTERA

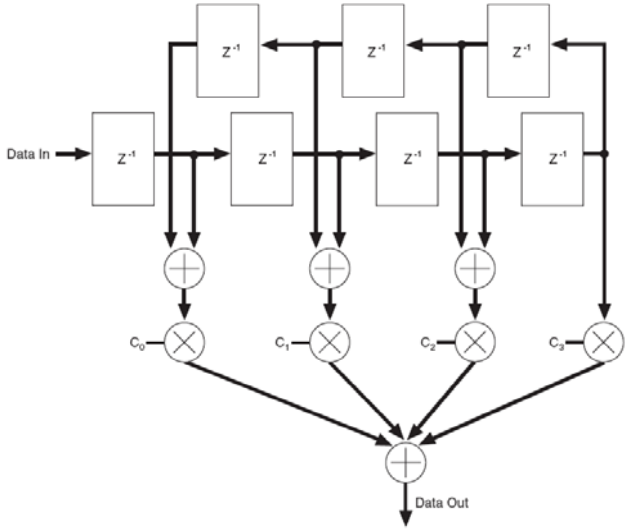
Basic FIR Filter



- Altera and Xilinx provide FIR filter compiler support.
- These examples are taken from Altera's "FIR Compiler User's Guide".

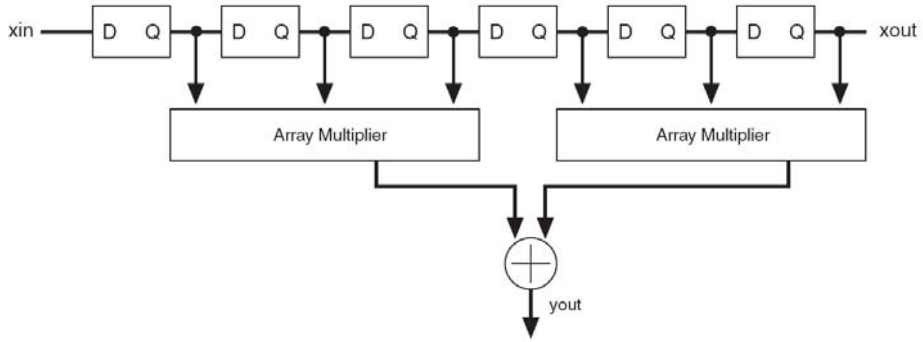
Source: ALTERA

Exploiting Symmetric Coefficients (7-tap)



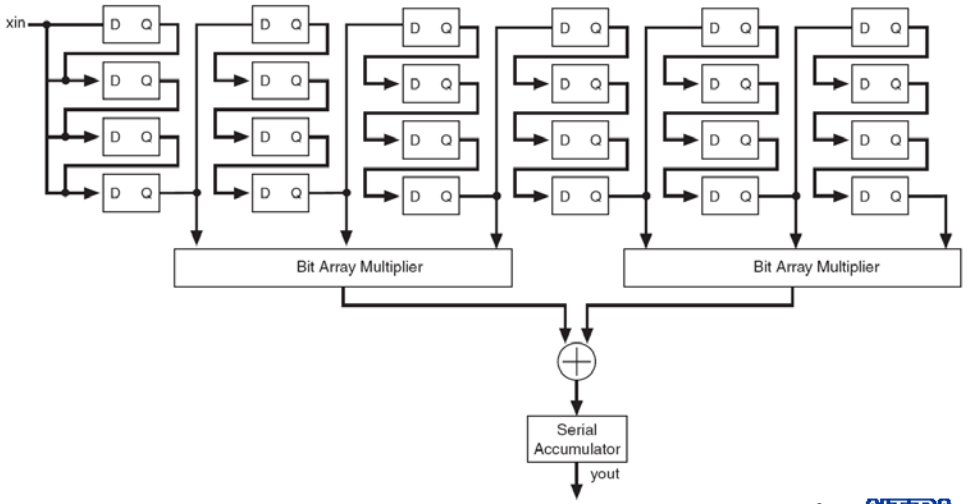
Source: ALTERA

Parallel Implementation of FIR Filter



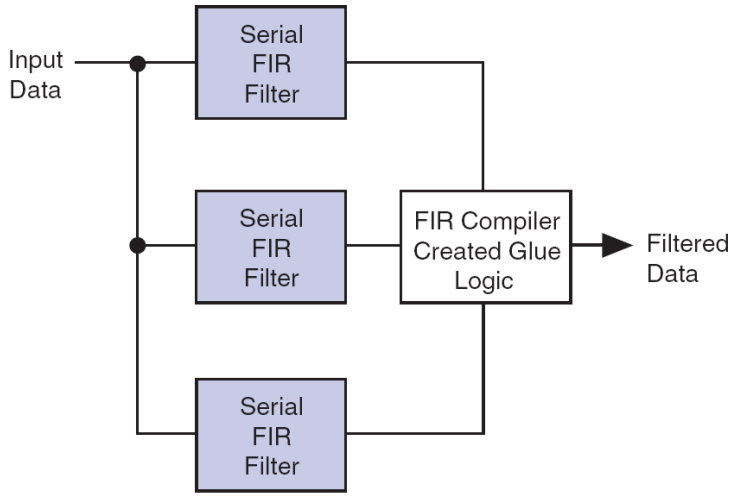
Source: ALTERA

Serial Implementation of FIR Filter



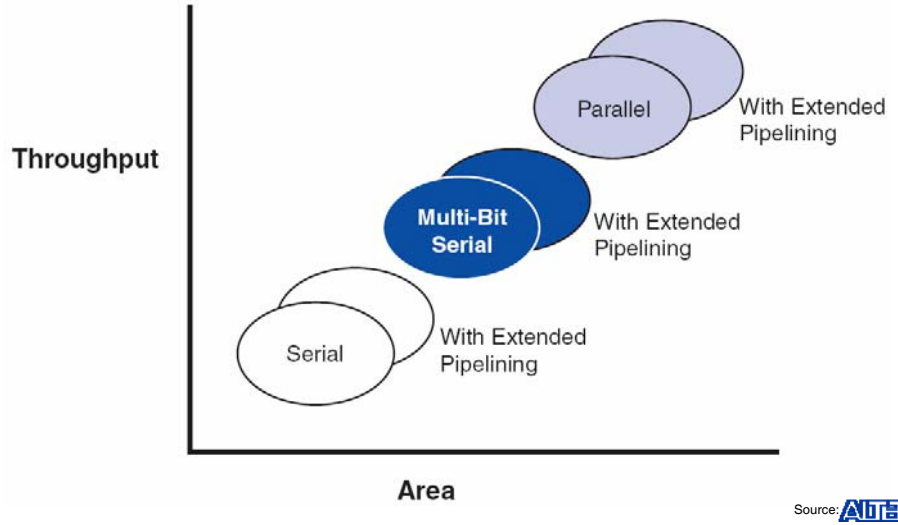
Source: ALTERA

Multibit Serial Implementation of FIR Filter



Source: ALTERA

FIR Filter Compiler Design Space



Source: ALTERA

Floating-Point Numbers

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$	Low precision and/or range
Rational	$\pm p/q$	Difficult arithmetic
Floating-point	$\pm s \times b^e$	Most common scheme
Logarithmic	$\pm \log_b x$	Limiting case of floating-point

Fixed-point numbers

$x = (0000\ 0000\ .\ 0000\ 1001)_{two}$ Small number
$y = (1001\ 0000\ .\ 0000\ 0000)_{two}$ Large number

Floating-point numbers

$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

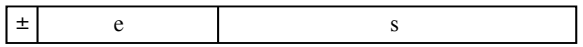
Note that a floating-point number comes with two signs:

Number sign, usually represented by a separate bit
Exponent sign, usually embedded in the biased exponent

Source: Parhami

Floating-Point Number Format and Distribution

Typical floating-point number format.



Sign: 0 : +, 1 : –

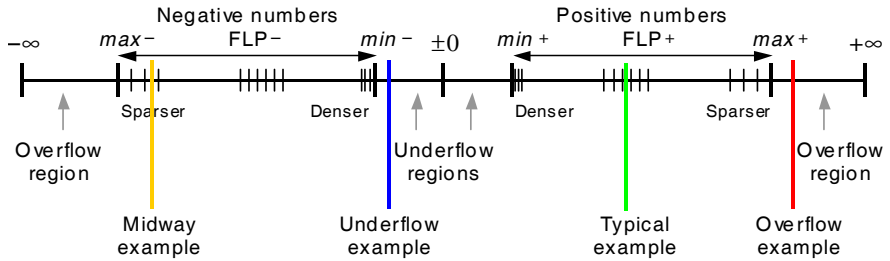
Exponent: Signed integer, often represented as unsigned value by adding a bias

Range with h bits: [–bias, 2^h–1–bias]

Significand: Represented as a fixed-point number

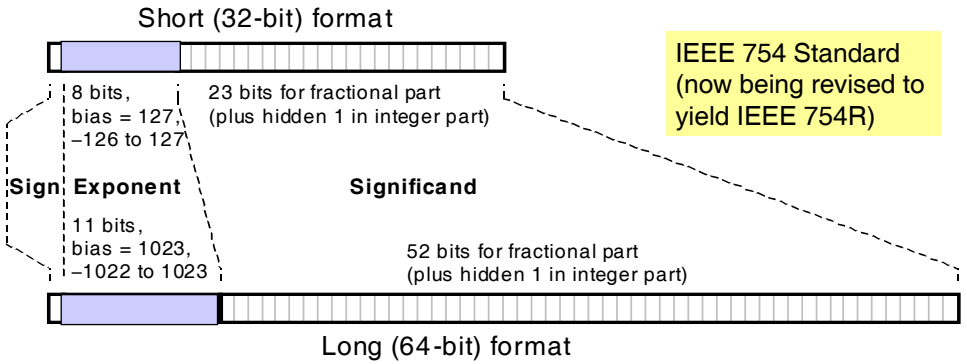
Usually normalized by shifting, so that the MSB becomes nonzero. In radix 2, the fixed leading 1 can be removed to save one bit; this bit is known as "hidden 1".

Subranges and special values in floating-point number representations.



Source: Parhami

The ANSI/IEEE Floating-Point Representation



IEEE 754 Standard (now being revised to yield IEEE 754R)

Source: Parhami

Overview of IEEE 754 Standard Formats

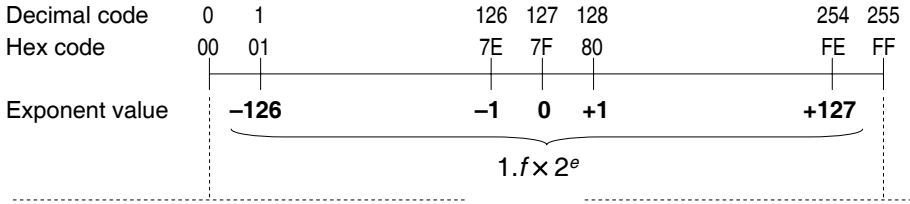
Some features of the ANSI/IEEE standard floating-point number representation formats.

Feature	Single/Short	Double/Long
Word width (bits)	32	64
Significand bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (±0)	$e + bias = 0$, $f = 0$	$e + bias = 0$, $f = 0$
Denormal	$e + bias = 0$, $f \neq 0$	$e + bias = 0$, $f \neq 0$
	represents $\pm 0.f \times 2^{-126}$	represents $\pm 0.f \times 2^{-1022}$
Infinity (±∞)	$e + bias = 255$, $f = 0$	$e + bias = 2047$, $f = 0$
Not-a-number (NaN)	$e + bias = 255$, $f \neq 0$	$e + bias = 2047$, $f \neq 0$
Ordinary number	$e + bias \in [1, 254]$	$e + bias \in [1, 2046]$
	$e \in [-126, 127]$	$e \in [-1022, 1023]$
	represents $1.f \times 2^e$	represents $1.f \times 2^e$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Source: Parhami

Exponent Encoding

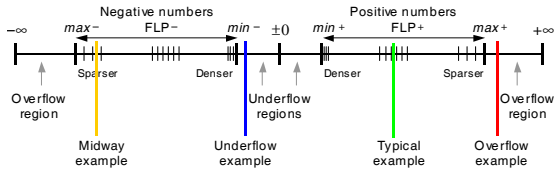
Exponent encoding in 8 bits for the single/short (32-bit) ANSI/IEEE format



$f = 0$: Representation of ±0
$f \neq 0$: Representation of denormals, $0.f \times 2^{-126}$

$f = 0$: Representation of ±∞
$f \neq 0$: Representation of NaNs

Exponent encoding in 11 bits for the double/long (64-bit) format is similar



Floating-Point Adders/Subtractors

Assume $e1 \geq e2$; *alignment shift* (*preshift*) is needed if $e1 > e2$

$$(\pm s1 \times b^{e1}) + (\pm s2 \times b^{e2}) = (\pm s1 \times b^{e1}) + (\pm s2 / b^{e1-e2}) \times b^{e1}$$
$$= (\pm s1 \pm s2 / b^{e1-e2}) \times b^{e1} = \pm s \times b^{e}$$

Example:

Numbers to be added:

$x = 2^5 \times 1.00101101$
$y = 2^1 \times 1.11101101$ ← Operand with smaller exponent to be preshifted

Operands after alignment shift:

$x = 2^5 \times 1.00101101$
$y = 2^5 \times 0.000111101101$

Result of addition:

$s = 2^5 \times 1.010010111101$ ← Extra bits to be rounded off
$s = 2^5 \times 1.01001100$ ← Rounded sum

Like signs:
Possible 1-position normalizing right shift

Different signs:
Possible left shift by many positions

Overflow/underflow during addition or normalization

Source: Parhami

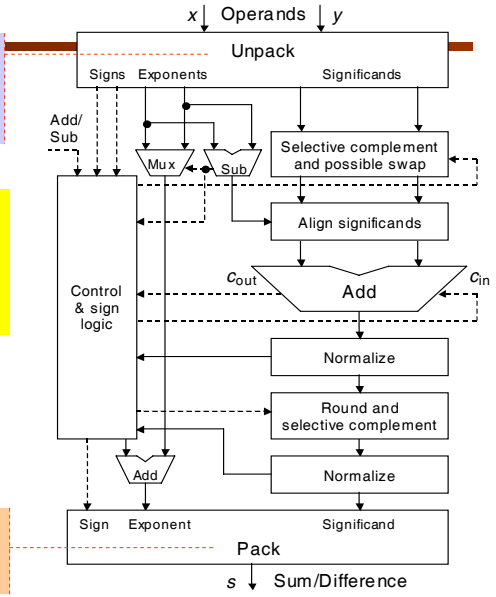
FP Adder/Sub

Isolate the sign, exponent, significand
Reinstate the hidden 1
Convert operands to internal format
Identify special operands, exceptions

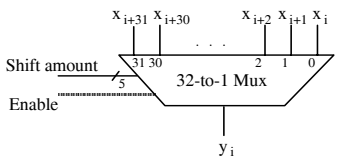
Other key parts of the adder:
- Significand aligner (preshifter)
- Result normalizer (postshifter), including leading 0s detector/predictor
- Rounding unit
- Sign logic

Converting internal to external representation, if required, must be done at the rounding stage

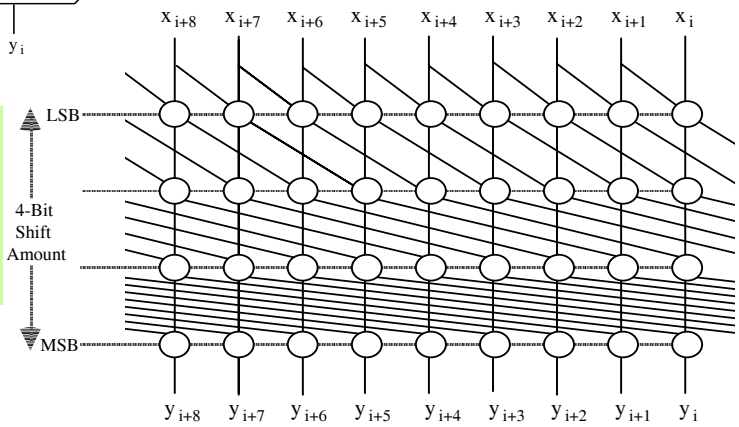
Combine sign, exponent, significand
Hide (remove) the leading 1
Identify special outcomes, exceptions



re- and Postshifting



One bit-slice of a single-stage pre-shifter.



Four-stage combinational shifter for preshifting an operand by 0 to 15 bits.

Source: Parhami

Leading Zeros/Ones Detection or Prediction

Leading zeros prediction, with adder inputs $(0x_0.x_{-1}x_{-2}\ldots)_{2\text{'s-compl}}$ and $(0y_0.y_{-1}y_{-2}\ldots)_{2\text{'s-compl}}$

Ways in which leading 0s/1s are generated:

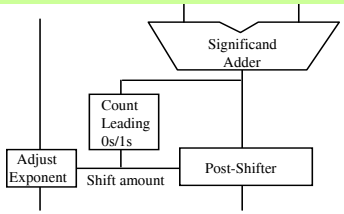
p p . . . p p g a a . . . a a g . . .
p p . . . p p g a a . . . a a p . . .
p p . . . p p a g g . . . g g a . . .
p p . . . p p a g g . . . g g p . . .

Prediction might be done in two stages:

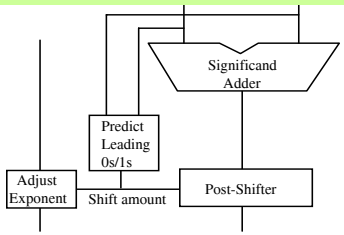
- Coarse estimate, used for coarse shift
- Fine tuning of estimate, used for fine shift

In this way, prediction can be partially overlapped with shifting

Leading zeros/ones counting



Leading zeros/ones prediction.



Source: Parhami

Floating-Point Multipliers

$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$

$s1 \times s2 \in [1, 4)$: may need postshifting

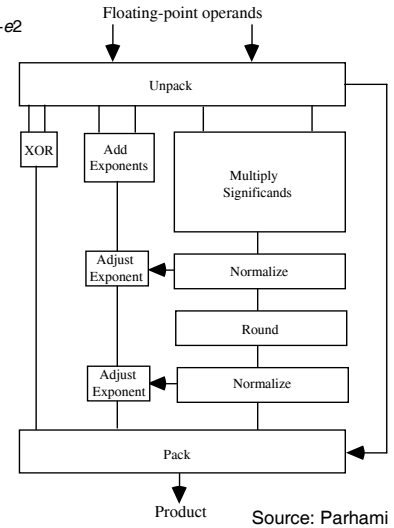
Overflow or underflow can occur during multiplication or normalization

Speed considerations

Many multipliers produce the lower half of the product (rounding info) early

Need for normalizing right-shift is known at or near the end

Hence, rounding can be integrated in the generation of the upper half, by producing two versions of these bits



Source: Parhami

Further references for Floating Point on FPGAs

- **An analysis of the double-precision floating-point FFT on FPGAs**
  Hemmert, K.S.; Underwood, K.D.; 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 18-20 April 2005 Page(s):171 - 180
- **Architectural Modifications to Improve Floating-Point Unit Efficiency in FPGAs**
  Beauchamp, M.J.; Hauck, S.; Underwood, K.D.; Hemmert, K.S.; International Conference on Field Programmable Logic and Applications, 28-30 Aug. 2006 Page(s):1 - 6
- **Double precision floating-point arithmetic on FPGAs**
  Paschalakis, S.; Lee, P.; IEEE International Conference on Field-Programmable Technology (FPT), 15-17 Dec. 2003 Page(s):352 - 358