HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

Digitale Synthese

Bachelor in de Industriële Wetenschappen: elektronica-ICT

SP E

EmSD
Embedded System Design

ir. J. Meel
feb 2010

HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT SINT-KATELIJNE-WAVER

Digitale Elektronica

Combinatorial Functions

EmSD Embedded System Design

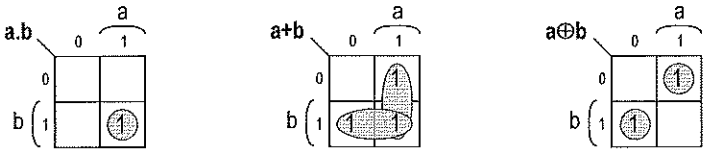
ir. J. Meel
feb 2009

1. Basic Logic Functions

truth table

b	a	AND	OR	EXOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
		all '1'	not all '0'	different (odd #'1')

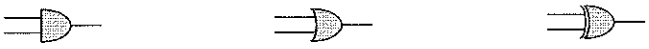
Karnaugh map



Som Of Products

a . b | **a + b** | **a⊕b = $\bar{a}$.b + a.$\bar{b}$**

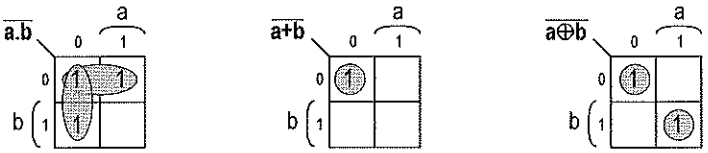
schematic



truth table

b	a	NAND	NOR	EXNOR
0	0	1	1	1
0	1	1	0	0
1	0	1	0	0
1	1	0	0	1
		not all '1'	all '0'	equal (even #'0')

Karnaugh map

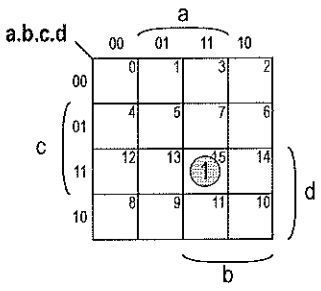


Som Of Products

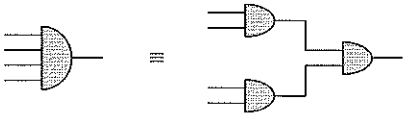
$\overline{a} + \overline{b}$ $\quad \overline{a} \cdot \overline{b}$ $\quad \overline{a \oplus b} = \overline{a} \oplus b = \overline{a}.\overline{b} + a.b$

schematic

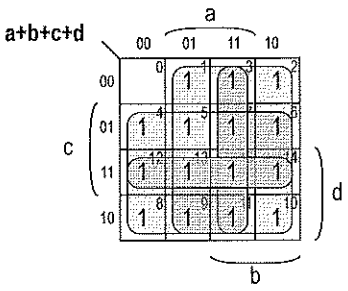
1.1 Multiple Inputs



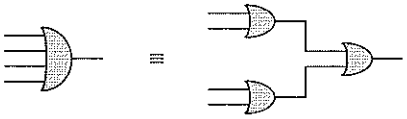
a.b.c.d = (a.b) . (c.d)



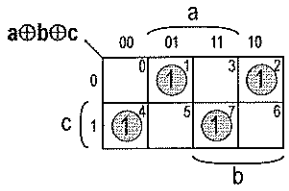
multilevel logic



a+b+c+d = (a+b) + (c+d)

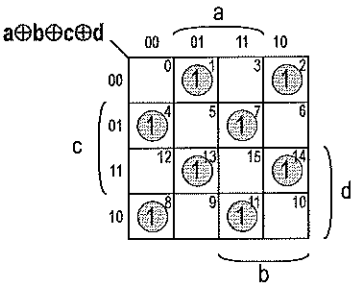


multilevel logic

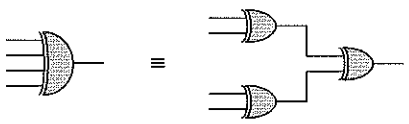


$a \oplus b \oplus c = a \oplus b.\bar{c} + \overline{a \oplus b}.c = \bar{a}.b.\bar{c} + a.\bar{b}.\bar{c} + \bar{a}.\bar{b}.c + a.b.c$

odd # '1'

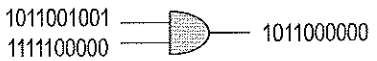


$a \oplus b \oplus c \oplus d = (a \oplus b) \oplus (c \oplus d)$

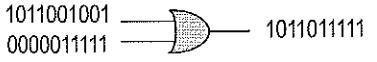


multilevel logic

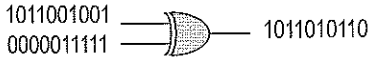
1.2 Gating (Masking) Functions



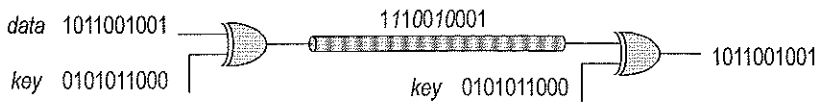
'1' = (passing data)
'0' = dominant (blocking/masking data)



'0' = (passing data)
'1' = dominant (blocking/masking data)



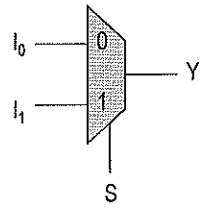
'0' = (passing data)
'1' = (inverting data)



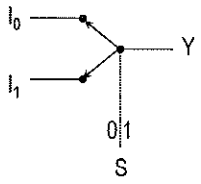
2. Multiplexer/Demultiplexer

2.1 Multiplexer

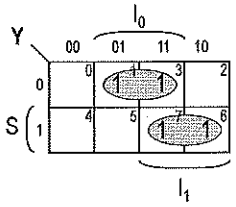
block diagram symbol



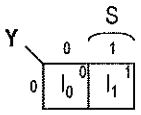
mechanical switch equivalent



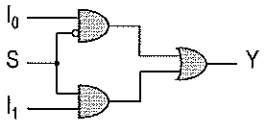
truth table (K-map)



Entered Variable K-map



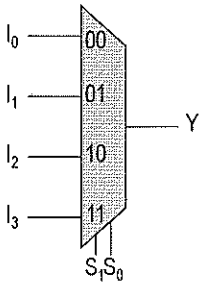
SOP synthesis



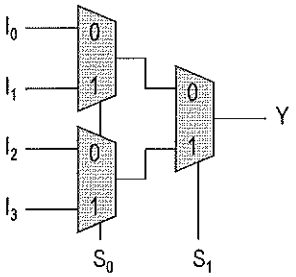
SOP equation

$$Y = I_0.\overline{S} + I_1.S$$

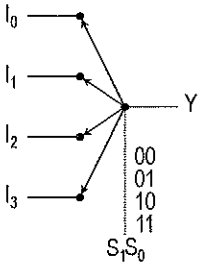
block diagram symbol



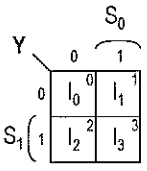
multilevel logic



mechanical switch equivalent



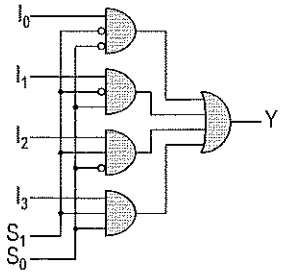
Entered Variable K-map



SOP equation

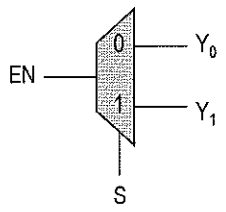
$$Y = I_0.\overline{S}_1\overline{S}_0 + I_1.\overline{S}_1S_0 + I_2.S_1\overline{S}_0 + I_3.S_1S_0$$

SOP synthesis

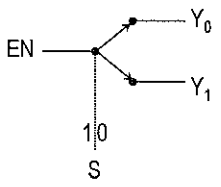


2.2 Demultiplexer/Decoder

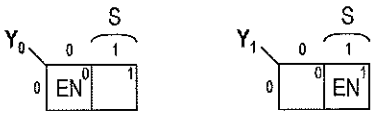
block diagram symbol



mechanical switch equivalent



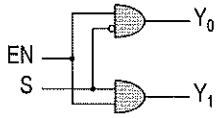
Entered Variable K-maps



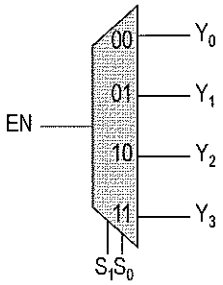
$Y_0 = \bar{S}.EN$ $Y_1 = S.EN$

SOP equation

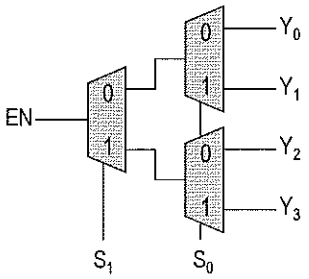
SOP synthesis



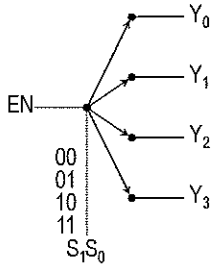
block diagram symbol



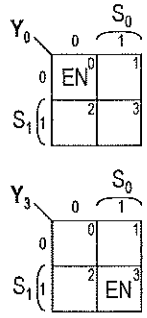
multilevel logic



mechanical switch equivalent



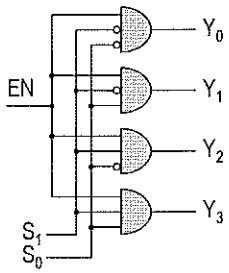
Entered Variable K-map



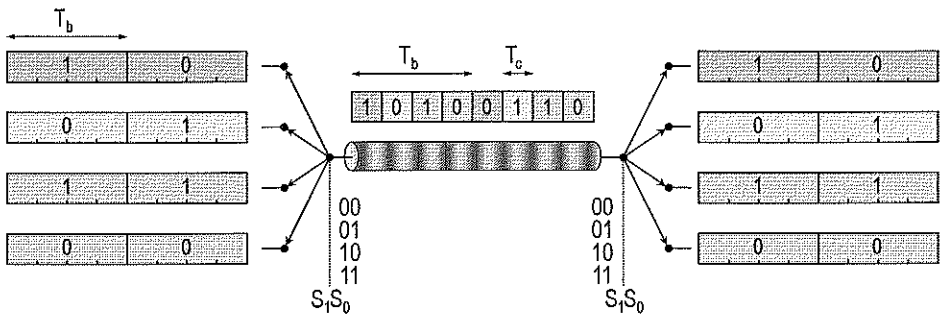
truth table - SOP equation

EN	S_1	S_0	Y_3	Y_2	Y_1	Y_0	
0	x	x	0	0	0	0	null state
1	0	0	0	0	0	1	$Y_0 = \bar{S}_1.\bar{S}_0.EN$
1	0	1	0	0	1	0	$Y_1 = \bar{S}_1.S_0.EN$
1	1	0	0	1	0	0	$Y_2 = S_1.\bar{S}_0.EN$
1	1	1	1	0	0	0	$Y_3 = S_1.S_0.EN$

SOP synthesis



2.3 Time Division Multiplexing (TDM)



$T_b = n.T_c$

bitrate: $R_c = n.R_b$

wires: $n \rightarrow 1$

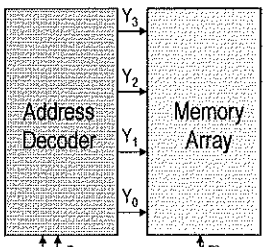
2.4 Address Decoder

EN = 1

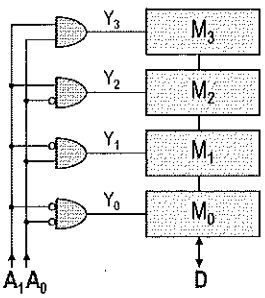
$S_1 = A_1$

$S_0 = A_0$

select 1 out of 2^n

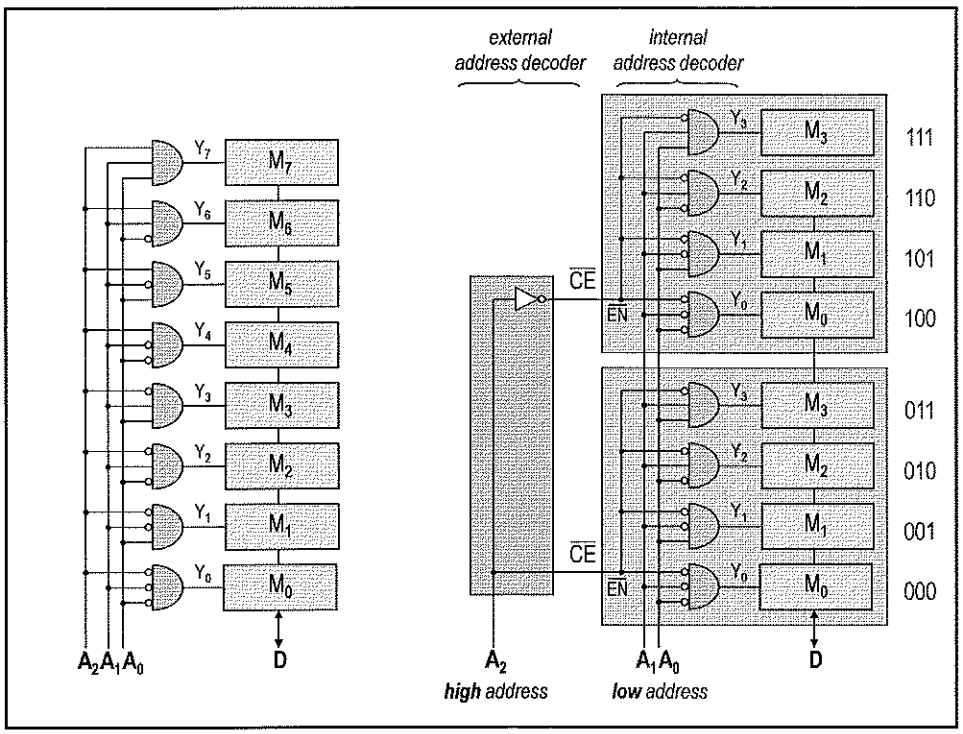


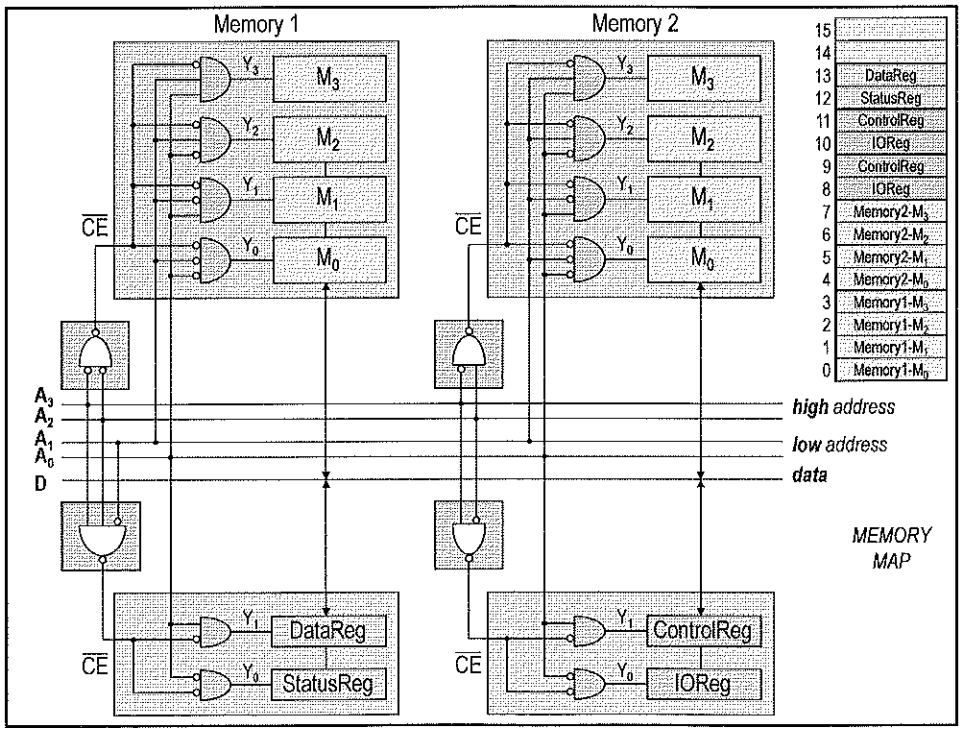
n-bit address = location A_1A_0 D *data = content*



Y_i = address line

M_i = memory cell

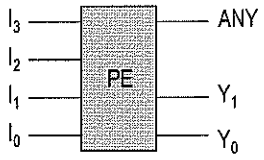

external address decoder
internal address decoder
Y_7 M_7
Y_6 M_6
Y_5 M_5
Y_4 M_4
Y_3 M_3
Y_2 M_2
Y_1 M_1
Y_0 M_0
$A_2A_1A_0$
D
$\overline{CE}$
$\overline{EN}$
111
110
101
100
011
010
001
000
A_2
high address
A_1A_0
low address


Memory 1
Memory 2
$\overline{CE}$
M_3 M_2 M_1 M_0
DataReg
StatusReg
ControlReg
IOReg
A_3 A_2 A_1 A_0 D
high address
low address
data
MEMORY MAP
15
14
13 DataReg
12 StatusReg
11 ControlReg
10 IOReg
9 ControlReg
8 IOReg
7 Memory2-M_3
6 Memory2-M_2
5 Memory2-M_1
4 Memory2-M_0
3 Memory1-M_3
2 Memory1-M_2
1 Memory1-M_1
0 Memory1-M_0

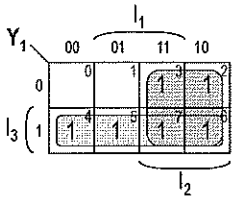
3. Encoder

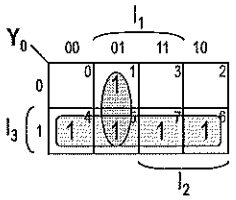
Priority Encoder

block diagram symbol



truth table (K-map)



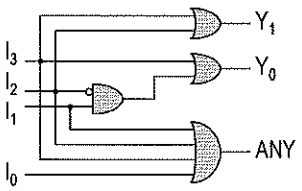


$Y_1 = I_3 + I_2$

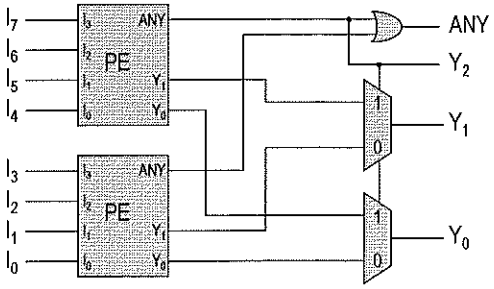
$Y_0 = I_3 + \bar{I}_2 . I_1$

I_3	I_2	I_1	I_0	Y_1	Y_0	ANY	
1	x	x	x	1	1	1	I_3 = highest priority (code 11)
0	1	x	x	1	0	1	I_2 = (code 10)
0	0	1	x	0	1	1	I_1 = (code 01)
0	0	0	1	0	0	1	I_0 = lowest priority (code 00)
0	0	0	0	0	0	0	inactive state

SOP synthesis

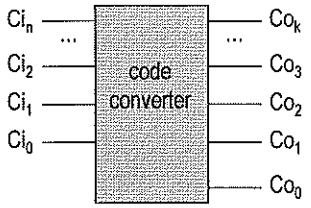


Combining Priority Encoders



I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0	ANY	
1	x	x	x	x	x	x	x	1	1	1	1	I_7 = highest priority (code 111)
0	1	x	x	x	x	x	x	1	1	0	1	I_6 = (code 110)
0	0	1	x	x	x	x	x	1	0	1	1	I_5 = (code 101)
0	0	0	1	x	x	x	x	1	0	0	1	I_4 = (code 100)
0	0	0	0	1	x	x	x	0	1	1	1	I_3 = (code 011)
0	0	0	0	0	1	x	x	0	1	0	1	I_2 = (code 010)
0	0	0	0	0	0	1	x	0	0	1	1	I_1 = (code 001)
0	0	0	0	0	0	0	1	0	0	0	1	I_0 = lowest priority (code 000)
0	0	0	0	0	0	0	0	0	0	0	0	inactive state

4. Code Converter



binary ↔ Gray

binary ↔ 2'C (two's complement)

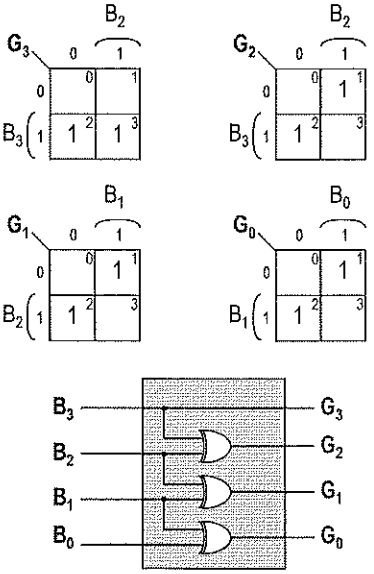
binary ↔ BCD (Binary Coded Decimal)

binary ↔ 7-segment

4.1 Binary-Gray code converter

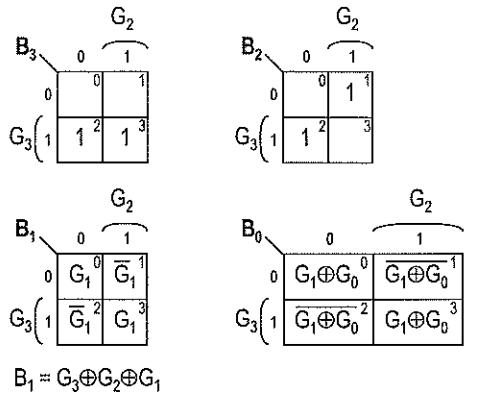
Binary-to-Gray code converter

B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

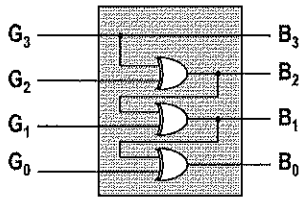


Gray-to-Binary code converter

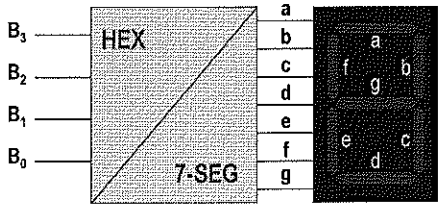
G_3	G_2	G_1	G_0	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0



$B_1 = G_3 \oplus G_2 \oplus G_1$



4.2 Binary-to 7-Segment code converter

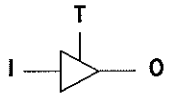


hex		a	b	c	d	e	f	g
0000		1	1	1	1	1	1	0
0001		0	1	1	0	0	0	0
0010		1	1	0	1	1	0	1
0011		1	1	1	1	0	0	1
0100		0	1	1	0	0	1	1
0101		1	0	1	1	0	1	1
0110		1	0	1	1	1	1	1
0111		1	1	1	0	0	0	0
1000		1	1	1	1	1	1	1
1001		1	1	1	1	0	1	1

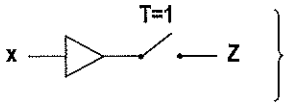
5. Buses

5.1 Bus Components

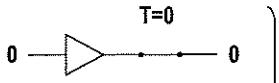
Tri-State buffer

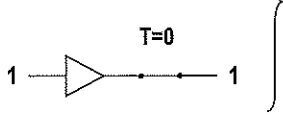


T	I	O	
1	x	Z	high impedance
0	0	0	buffer
	1	1	



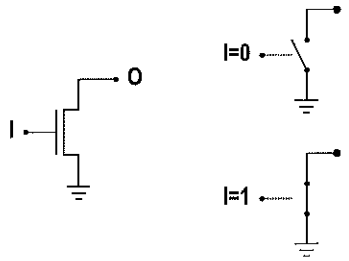
high impedance



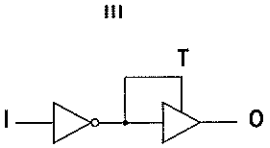


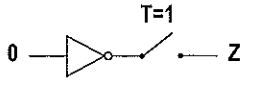
buffer

Open Drain / Open Collector

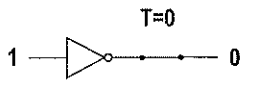


I	O	
0	Z	high impedance
1	0	inverter





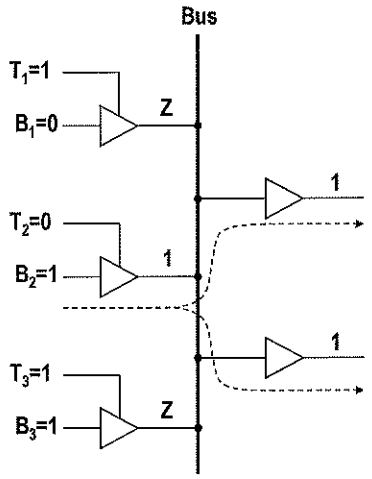
high impedance



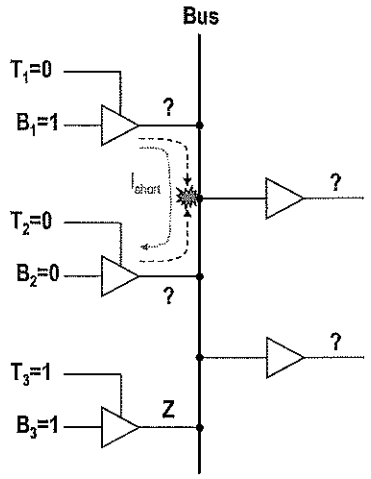
inverter

6.2 Bus Architecture

Multiplexed Bus

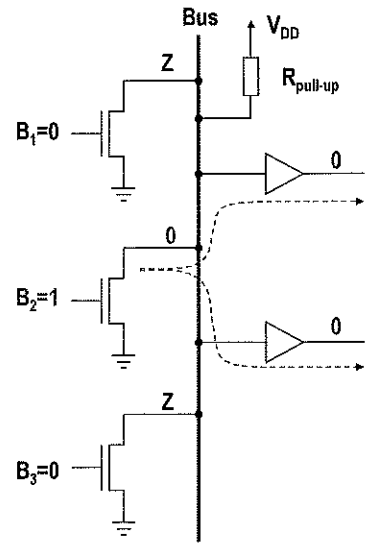


- *only 1 output active (T_i=0)*
- *several inputs can be active*

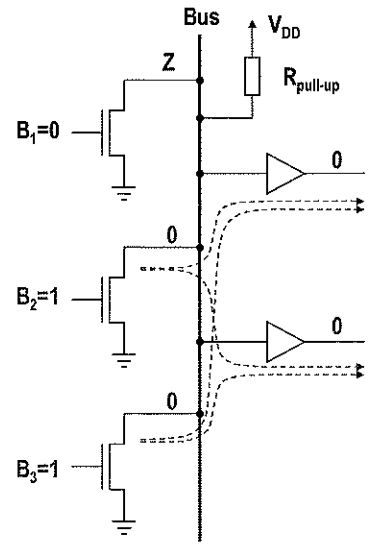


BUSCONFLICT
when several outputs
are active simultaneously

Wired-OR Bus

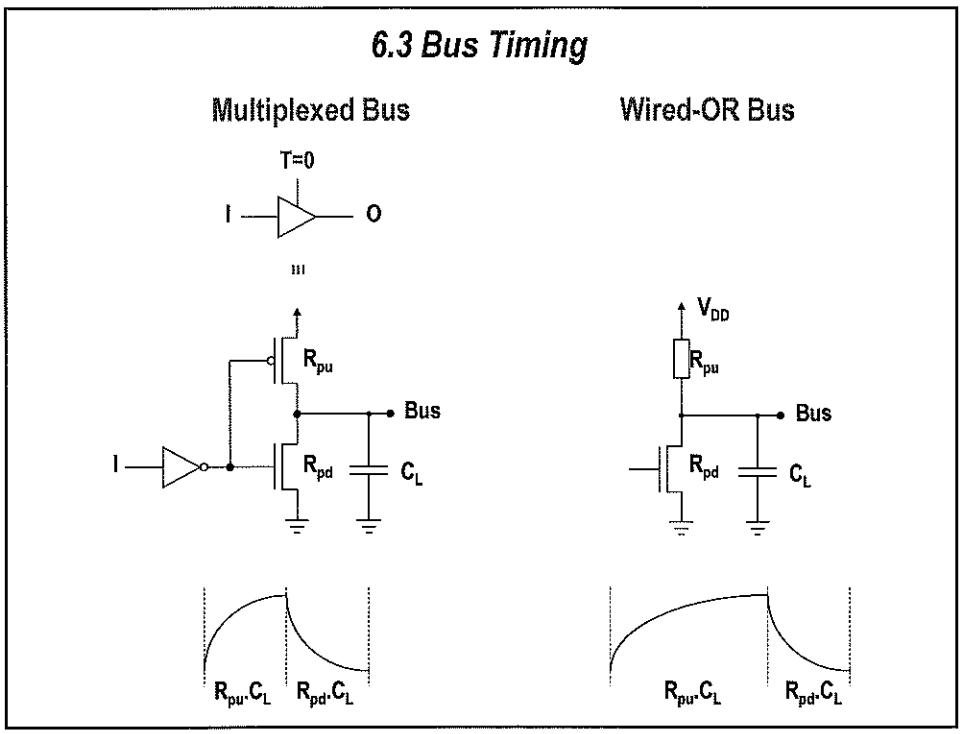


- *only 1 output active (B_i=0)*
- *several inputs can be active*

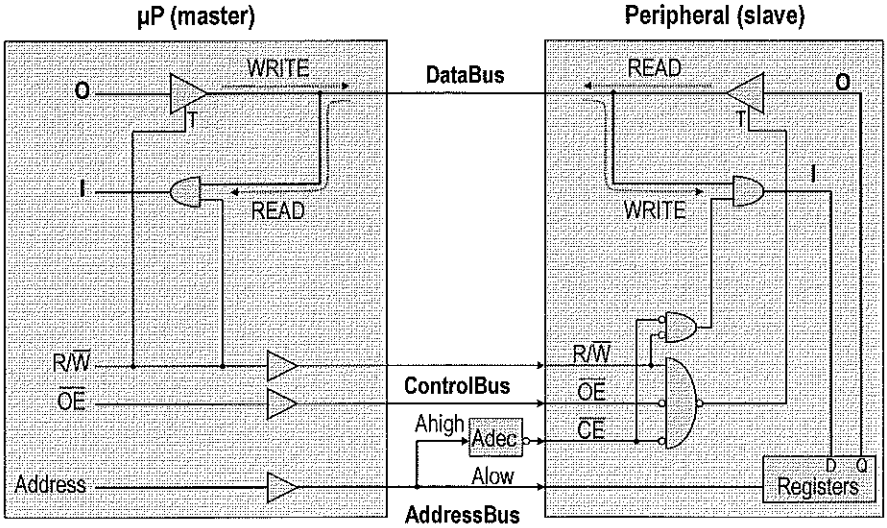


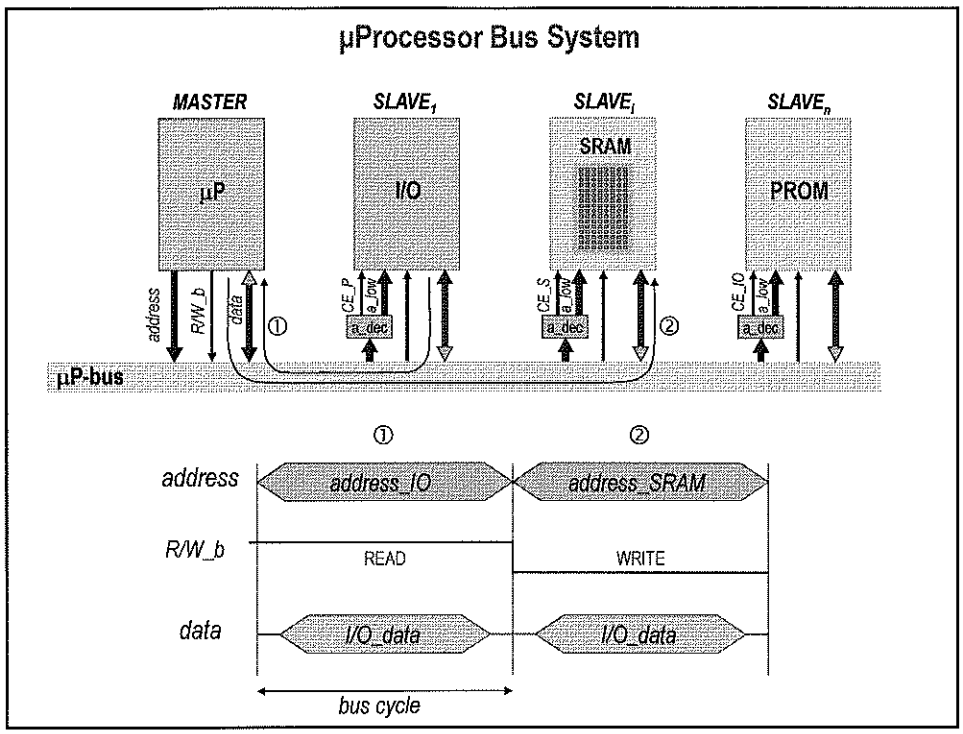
several outputs
can be active simultaneously

6.3 Bus Timing



6.4 Bidirectional Bus




µProcessor Bus System
MASTER
SLAVE1
SLAVEi
SLAVEn
µP
I/O
SRAM
PROM
address
R/W_b
data
CE_P
CE_S
CE_IO
a_low
a_dec
µP-bus
①
②
address
address_IO
address_SRAM
R/W_b
READ
WRITE
data
I/O_data
I/O_data
bus cycle

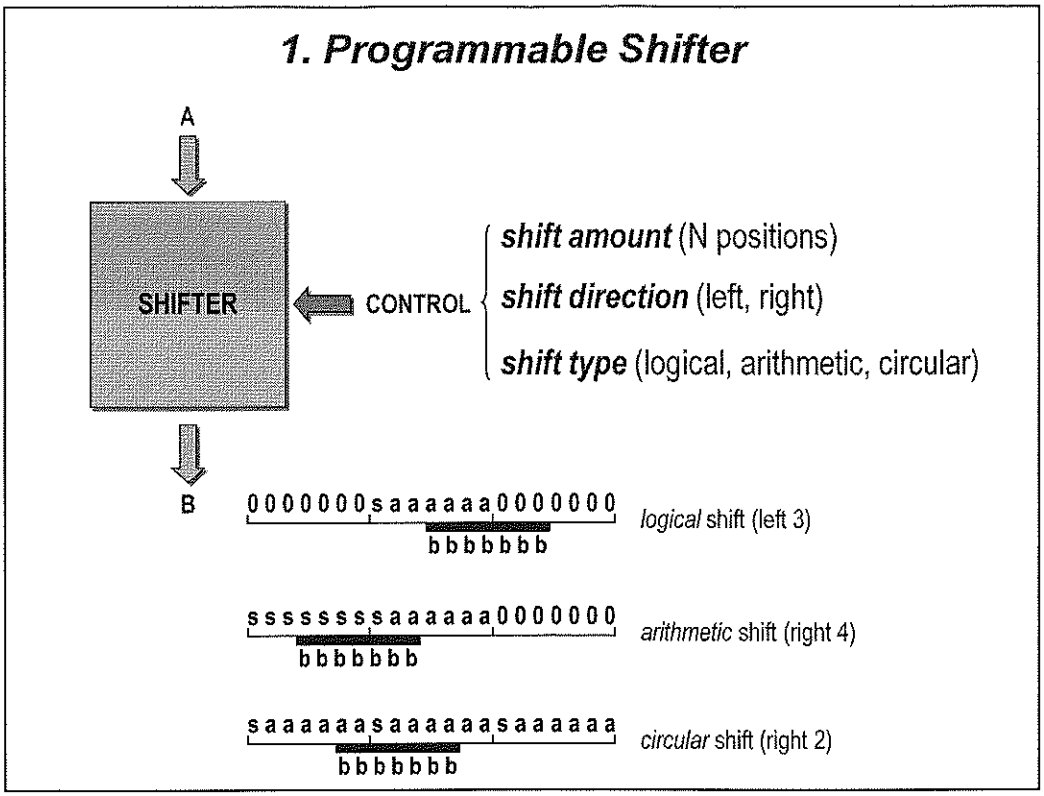
HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT SINT-KATELIJNE-WAVER

Digitale Synthese
Arithmetic Functions
SHIFT COMPARE

EmSD Embedded System Design

ir. J. Meel

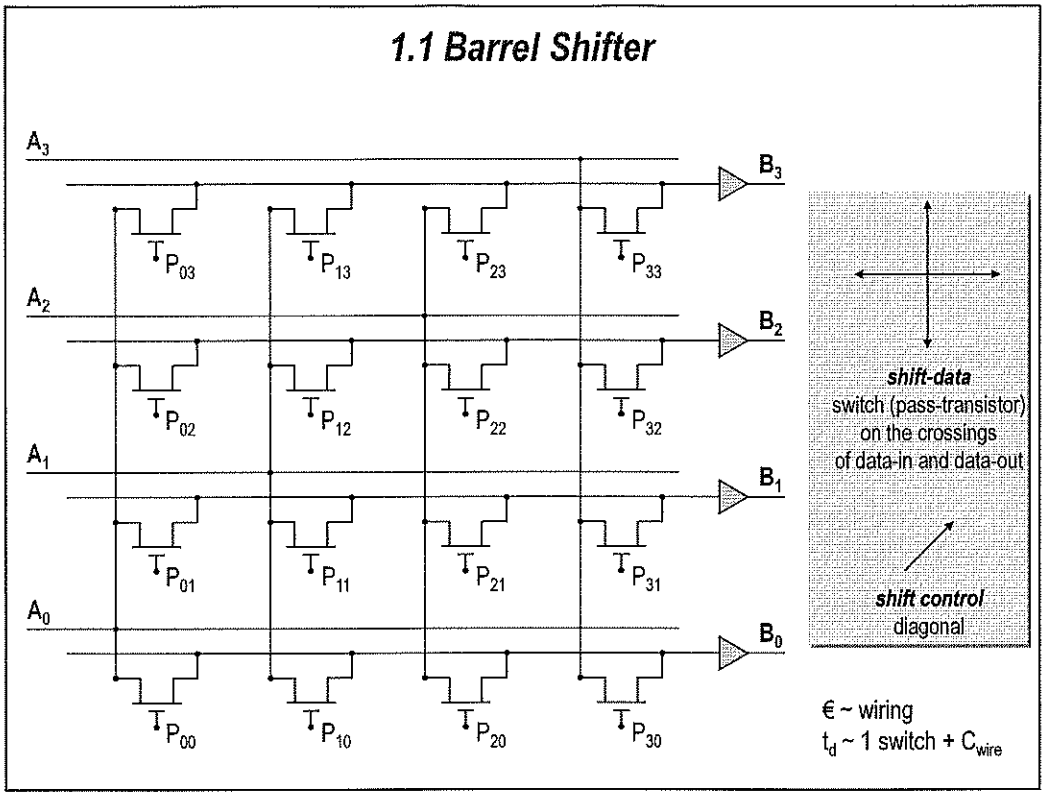
april 2007



Shifting and rotating data is required in several applications including arithmetic operations (floating-point units, scalers, and multiplications by constant numbers which can be implemented as a combination of add and shift operations), variable-length coding, and bit-indexing. Consequently, shifters, which are capable of shifting or rotating data in a single cycle, are commonly found in both digital signal processors and general-purpose processors.
The shift operation is an essential arithmetic operation that requires adequate hardware support. Shifting a data word left or right over a *constant* amount is a trivial hardware operation and is implemented by the appropriate signal wiring. A *programmable* shifter, on the other hand, is more complex and requires active circuitry. In essence, such a shifter is nothing less than an intricate multiplexer circuit.
Consider a simple one-bit left-right shifter. Depending on the control signals, the input word is either shifted left or right, or else it remains unchanged. Multi-bit shifters can be built by cascading a number of these units. This approach rapidly becomes complex, unwieldy, and ultimately too slow for larger shift values. Therefore, a more structured approach is advisable. Two commonly used shift structures, the barrel shifter and the logarithmic shifter, will be discussed.

A is the input operand denoted as saaaaaa (s=sign bit), B the shifted/rotated result and N the shift/rotate amount.
The following six operations are performed:

- *logic right shift*:n -bit right shift of A, the upper N bits of the result B are set to zeros.
- *logic left shift*:n -bit left shift of A, the lower N bits of the result B are set to zeros.
- *arithmetic right shift*:n -bit right shift of A, the upper N bits of the result B are set to s (=sign-bit of A).
- *arithmetic left shift*: n-bit left shift of A, the lower N bits of the result B are set to zeros.
- *circular right shift (rotate):* n-bit right shift of A, upper N bits of the result B are set to lower N bits of A.
- *circular left shift (rotate):* n-bit left shift of A, lower N bits of the result B are set to upper N bits of A.



Barrel Shifter

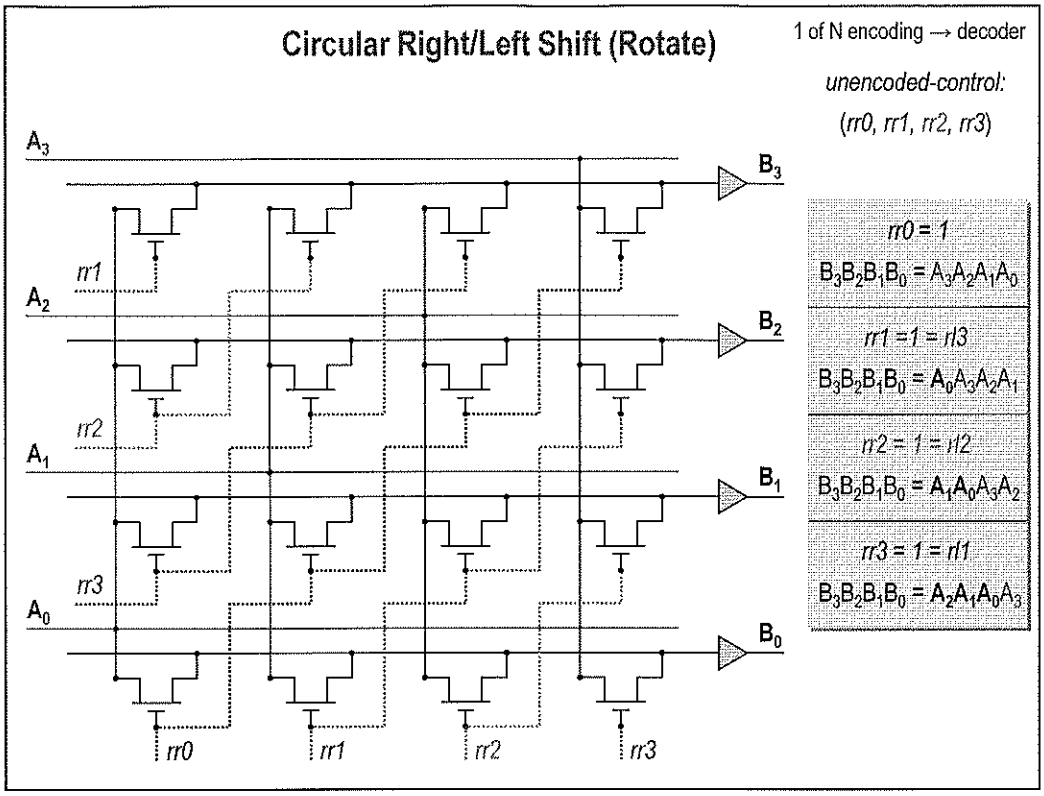
The structure of a barrel shifter is shown. It consists of an array of transistors, in which the number of rows equals the word length of the data, and the number of columns equals the maximum shift width N. In this case, both are set equal to N=4.

The lines of the input data A have a matrix structure: N horizontal lines, N vertical lines. Each horizontal line A_i is connected to a corresponding vertical line A_i. The horizontal lines of the output data B are crossing the vertical lines with the input data A. At each crossing a pass transistor (switch) is placed. By activating the pass transistor P_{ij} the input bit A_i is connected with the output data bit B_j.

The control wires are routed diagonally through the array when a shift operation is performed.

A major advantage of this shifter is that the signal has to pass through at most one transmission gate. In other words, the propagation delay is theoretically constant and independent of the shift value or shifter size. This is not true in reality, however, because the capacitance at the input of the buffers rises linearly with the maximum shift width.

An important property of this circuit is that the layout size is not dominated by the active transistors as in the case of all other arithmetic circuits, but by the number of wires running through the cell. More specifically, the size of the cell is bounded by the pitch of the metal wires. So, the cost (area) of a barrel shifter is dominated by wiring.



unencoded control

Another important consideration when selecting a shifter is the format in which the shift value N must be presented.F rom the schematic diagram shown, it is clear that the barrel shifter needs a control wire for every shift value. For example, a four-bit shifter needs four control signals. To shift/rotate over N=3 bits, the signals rr3:rr0 take on the value 1000. Only one of the signals is high. This is called 1 of N encoding. There are N control signals needed to implement N different shift amounts.

In a processor, the required shift/rotate amount N normally comes in an encoded binary format, which is substantially more compact. For instance, the encoded control word needs only two control signals and is represented as 11 for a shift over N=3 bits. To translate the latter representation into the former (with only one bit high), an extra decoder is required.

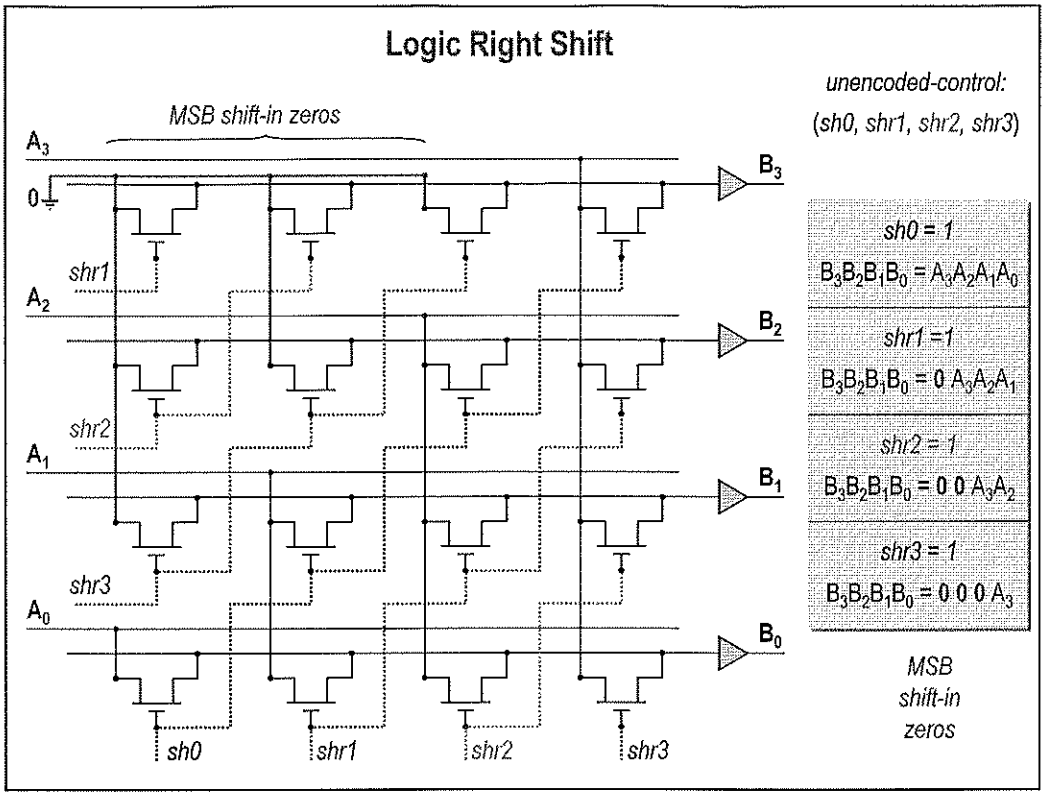
• *circular right shift (rotate):* n-bit right shift of A, upper N bits of the result B are set to lower N bits of A.

• *circular left shift (rotate):* n-bit left shift of A, lower N bits of the result B are set to upper N bits of A.

The control wires rr0, rr1, rr2, rr3 are routed diagonally through the array when a shift operation is performed.

For an N-bit left shift, the output data bit B_j is connected to the less significant input bit $A_{(i-N) \text{modulo} N}$.

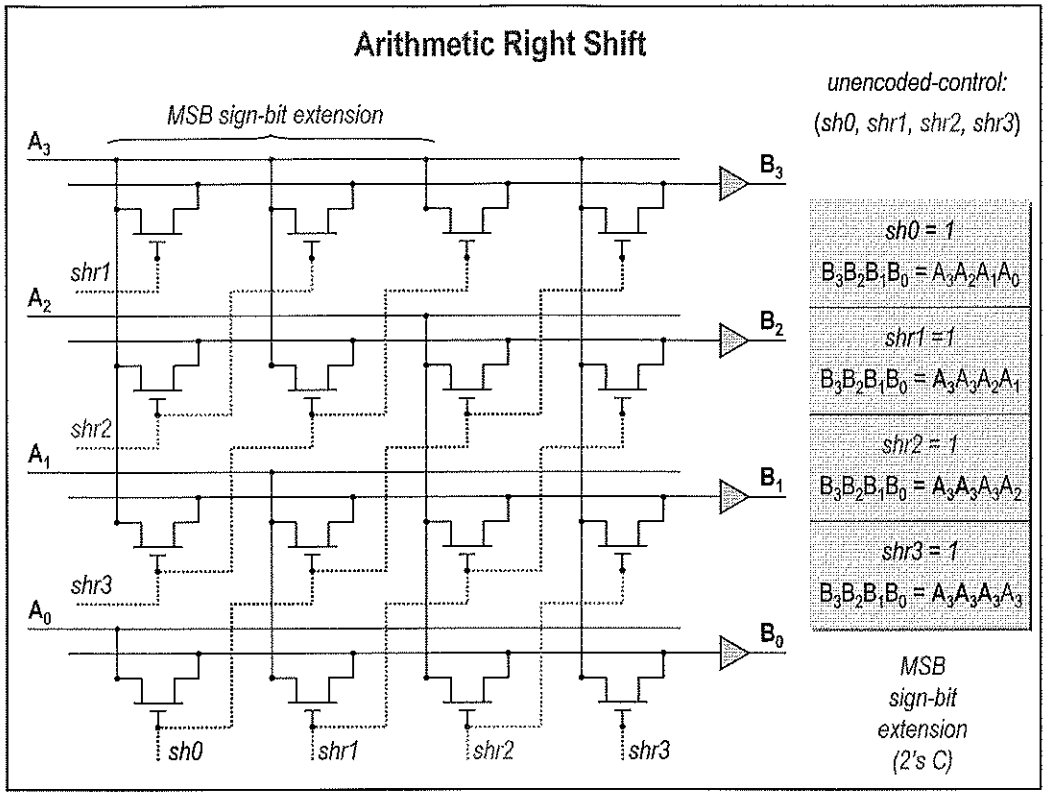
For an N-bit right shift, the output data bit B_j is connected to the more significant input bit $A_{(i+N) \text{modulo} N}$.



• *logic right shift*:n -bit right shift of A, the upper N bits of the result B are set to zeros.

The control wires sh0, shr1, shr2, shr3 are routed diagonally through the array when a shift operation is performed.

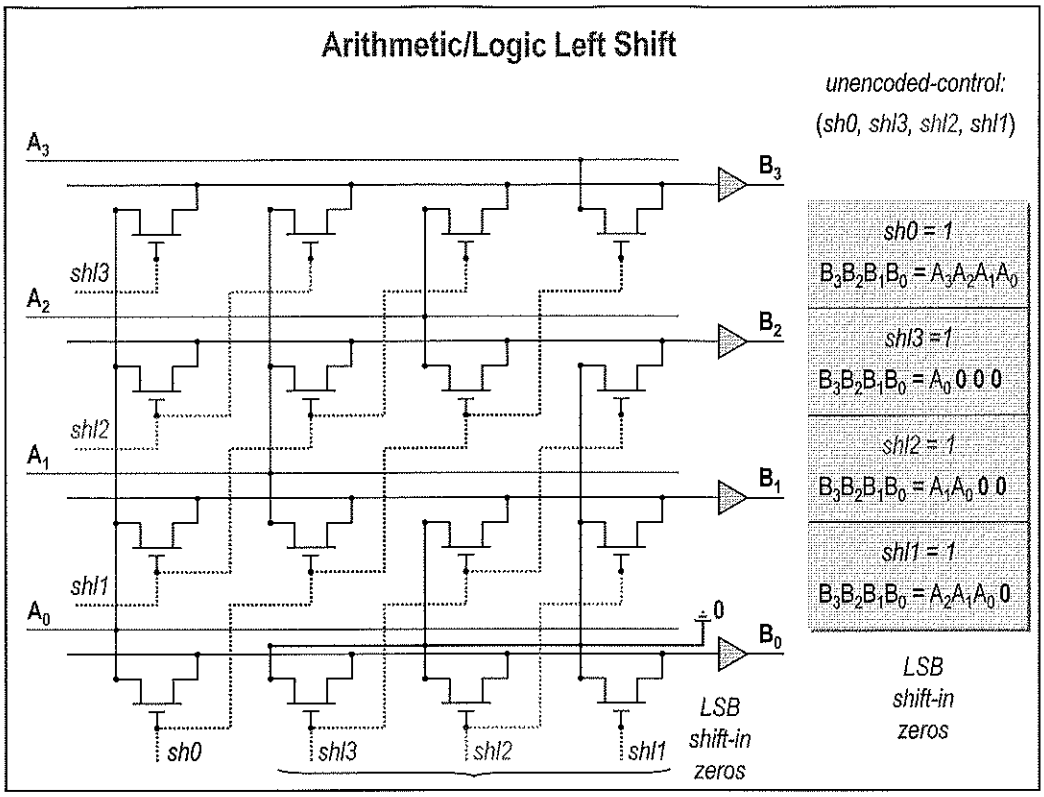
For an N-bit right shift, the output data bit B_j is connected to the more significant input bit A_{i+N}. The upper N bits of the result B are connected to the zero signal.



• *arithmetic right shift*:n -bit right shift of A, the upper N bits of the result B are set to s (=sign-bit of A).

The control wires sh0, shr1, shr2, shr3 are routed diagonally through the array when a shift operation is performed.

For an N-bit right shift, the output data bit B_j is connected to the more significant input bit A_{j+N}. The upper N bits of the result B are connected to the sign-bit of A. This result in a sign-extension for twos complement numbers.

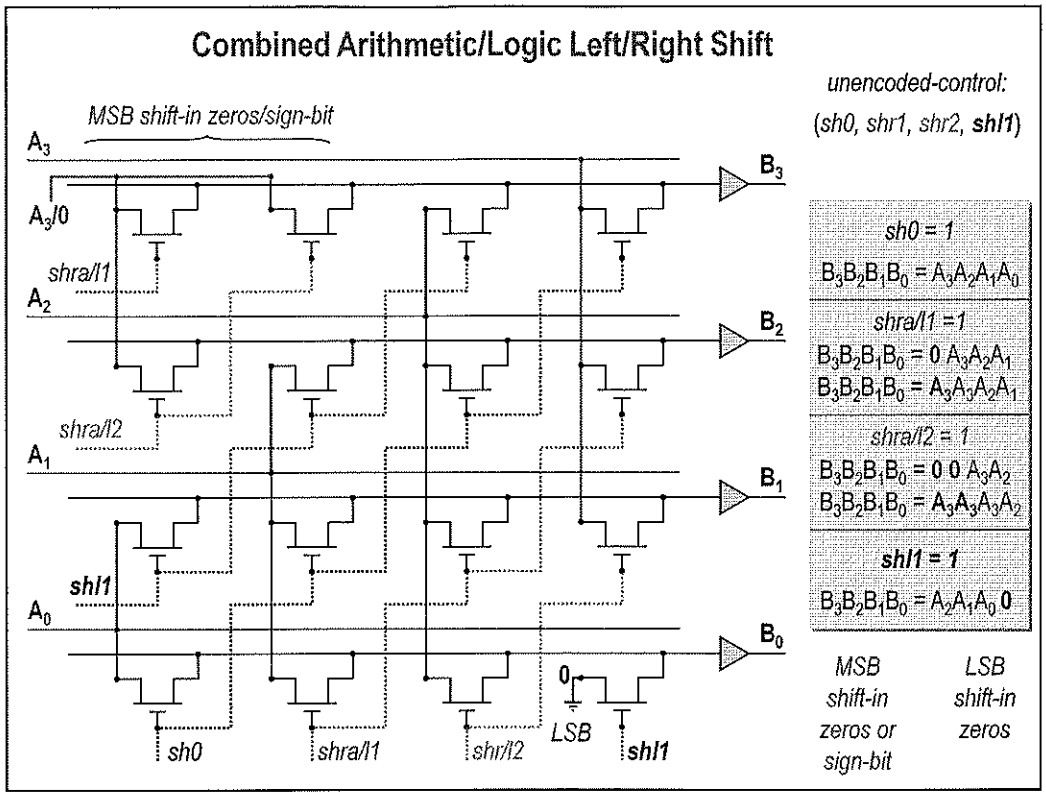


• *logic left shift*:n -bit left shift of A, the lower N bits of the result B are set to zeros.

• *arithmetic left shift*: n-bit left shift of A, the lower N bits of the result B are set to zeros.

The control wires sh0, shl1, shl2,shl 3 are routed diagonally through the array when a shift operation is performed.

For an N-bit left shift, the output data bit B_j is connected to the less significant input bit A_{i-N}. The lower N bits of the result B are connected to the zero signal.

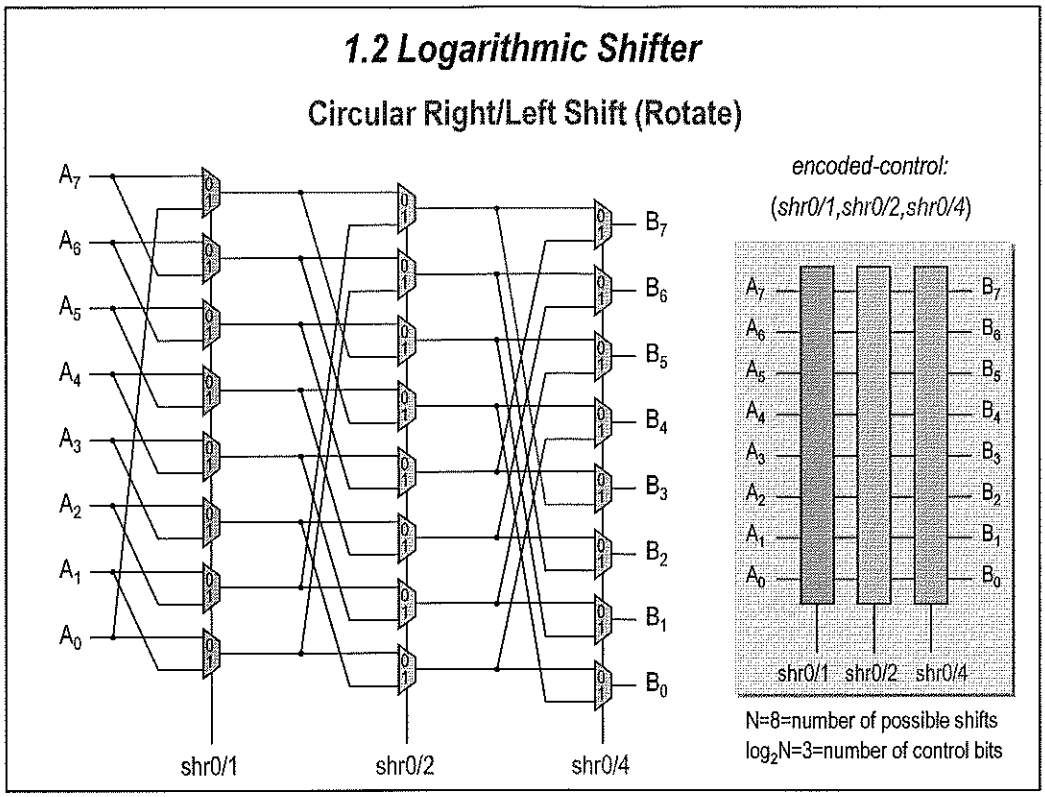


- *logic right shift*:n -bit right shift of A, the upper N bits of the result B are set to zeros.
- *logic left shift*:n -bit left shift of A, the lower N bits of the result B are set to zeros.
- *arithmetic right shift*:n -bit right shift of A, the upper N bits of the result B are set to s (=sign-bit of A).
- *arithmetic left shift*: n-bit left shift of A, the lower N bits of the result B are set to zeros.

The control wires are routed diagonally through the array when a shift operation is performed.

The number of columns limits the number of possible shifts. In the presented case: 2 right shifts and 1 left shift. To implement all possible left and right shifts (each N−1), a total of 2N−1 columns are needed.

The combination of arithmetic and logic shift only requires a multiplexing between A_{MSB} and 0 for the MSB bit that is shifted-in from the left (right shift).



Logarithmic Shifter

While the barrel shifter implements the whole shifter as a single array of pass transistors, the logarithmic shifter uses a *staged approach*. The total shift value N is decomposed into *shifts over powers of two*. A shifter with a maximum shift width of N consists of a $\log_2 N$ stages, where the i^{th} stage either shifts over i or passes the data unchanged. An example of a shifter with a maximum shift value of N−1=7 bits is shown.F or instance, to shift over 5 = $(101)_2$ bits, the first stage is set to shift mode (1), the second to pass mode (0), and the last stage again to shift (1). Notice that the control word for this shifter is already encoded, and no separate decoder is required.

The speed of the logarithmic shifter depends on the shift width N in a logarithmic way, since an N-bit shifter requires $\log_2 N$ stages. Furthermore, the series connection of pass transistors slows the shifter down for larger shift values. A careful introduction of intermediate buffers (in the multiplexers) is therefore necessary.

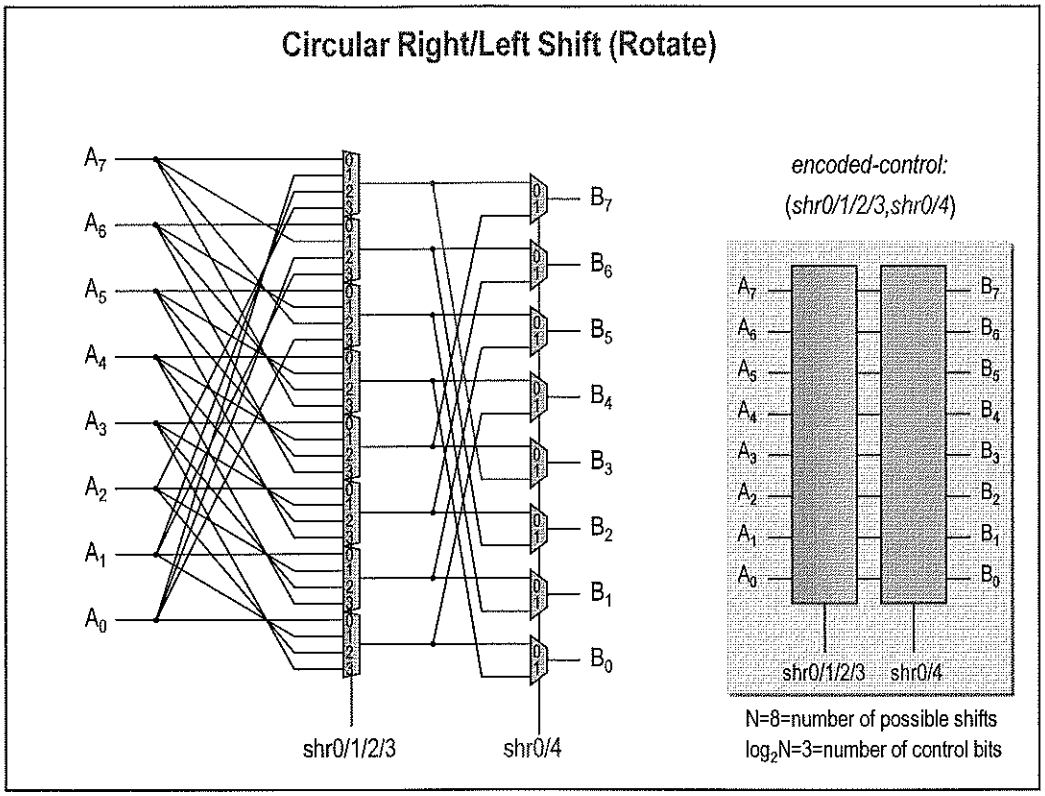
In general, a barrel shifter is appropriate for smaller shifters. For larger shift values, the logarithmic shifter becomes more effective, in terms of both area and speed. Furthermore, the logarithmic shifter is easily parameterized, allowing for automatic generation. The exploitation of regularity in an arithmetic operator can lead to dense and high-speed circuit implementations.

Logarithmic Shifter with Binary Stages

A logarithmic shifter (fully combinatorial), decomposes the shift operation into stages (3 stages in the figure shown). The first stage rotates the data by 0 or 1 positions, the second rotates the result by 0 or 2 positions.th e third rotates the result by 0 or 4 positions Together, these three shift-stages provide the desired rotations of 0:7 positions.

This binary decomposition scheme can be used for any number of bits. The number of levels required for an N-bit shifter is $\log_2 N$, rounded to the next higher number if N is not a power of two.T he first level rotates 0 or 1 positions, and subsequent levels each rotate by twice as many positions as the preceding level.

The select bits to each level form a binary-encoded shift control.



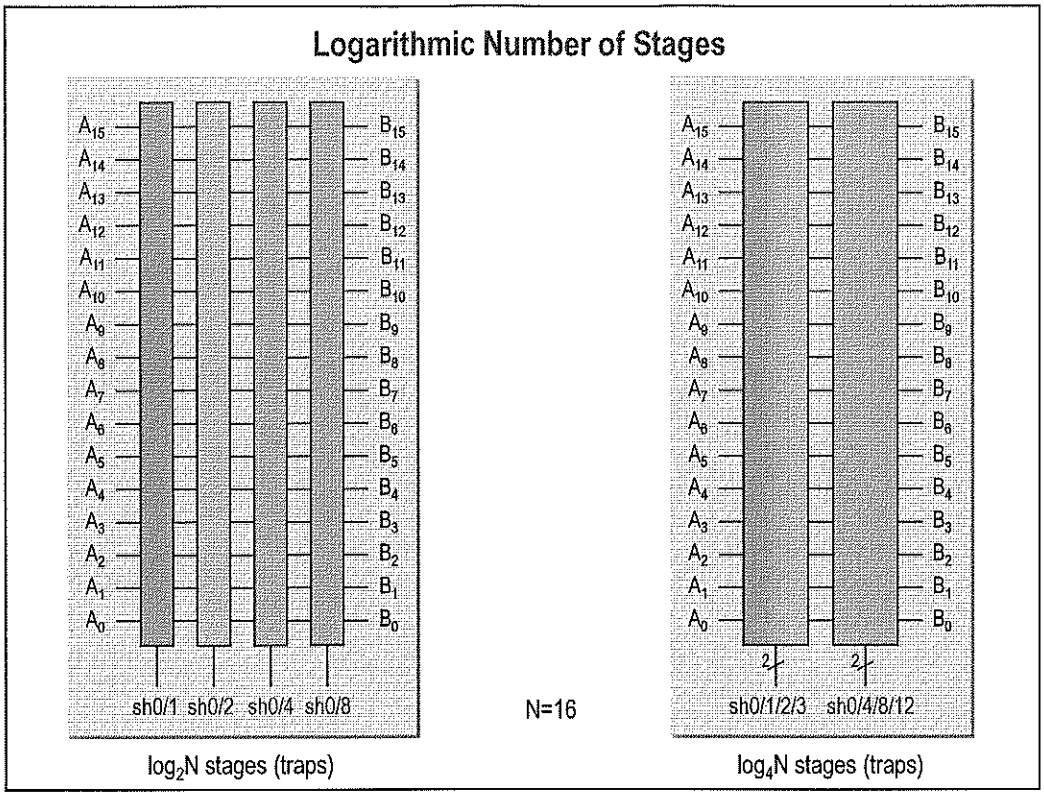
Logarithmic Shifter with Binary and Quaternary Stages

The 8-bit logarithmic shifter (fully combinatorial) shown, decomposes the shift operation into only 2 stages. The first stage rotates the data by 0, 1, 2 or 3 positions (4-input multiplexers), and the second rotates the result by 0 or 4 positions (2-input) multiplexers.T ogether, these two shift-stages provide the desired rotations of 0:7 positions.

The select bits to each level form a binary-encoded shift control.

A 16-bit logarithmic shifter can be implemented with 2 stages with 4-input multiplexers. The first level of multiplexers rotates by 0, 1, 2 or 3 positions, and the second by 0, 4, 8 or 12 positions. Together, these two shift-stages provide the desired rotations of 0:15 positions. The shift control remains binary.

This scheme can be expanded to any number of bits using $\log_4 N$ rotators that successively rotate by four times as many bit positions. For sizes that are odd powers of two, the final level should consist of less costly 2-input multiplexers.



A logarithmic shifter implemented with 2-input multiplexer stages requires $\log_2 N$ stages.

A logarithmic shifter implemented with 4-input multiplexer stages requires $\log_4 N$ stages.

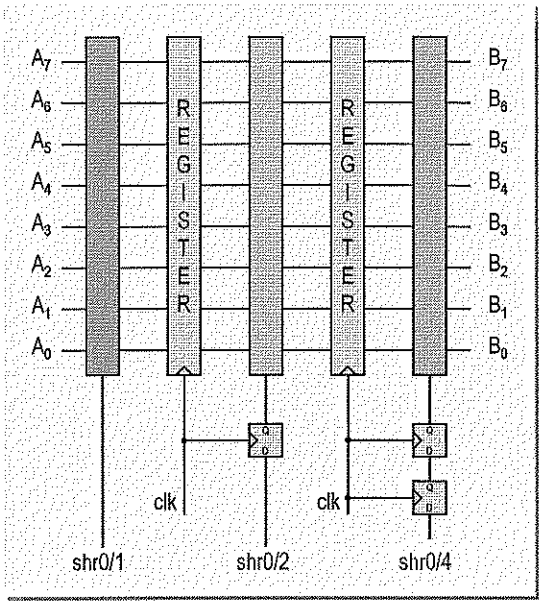
Minor changes (small amount of logic) will convert a logarithmic shifter from a rotator to a logic or arithmetic shifter. to a logical shift, or a can replace the hard wiring to allow several kinds of shifts.

Multiplexers can be implemented with transmission gates. A circuit like this with many transmission gates in series would benefit from having buffers inserted along the path.

Logarithmic shifters have intrinsic decode, as the shift bits are used directly to control the multiplexer stages.

The barrel shifter is better for small shifters (faster, not much bigger) and the logarithmic shifter is preferred for larger shifters both due to size and delay. Log shifters are always smaller.

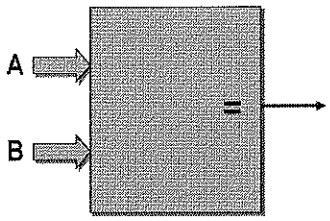
Pipelining



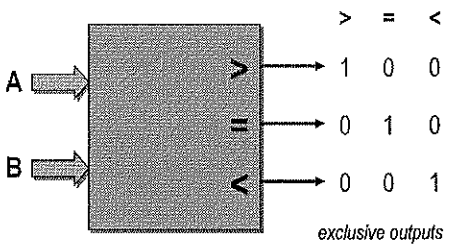
By inserting pipeline registers between the multiplexer stages, the speed of the logarithmic shifter can be increased. To synchronise (align) the control signals (shr) with the data, compensating registers must be introduced.

2. Comparator

Identity Comparator



Magnitude Comparator



>	=	<
1	0	0
0	1	0
0	0	1

exclusive outputs
(2 independent outputs)

exclusive outputs

$(A>B) = (\overline{A=B}).(\overline{A<B})$

$(A=B) = (\overline{A>B}).(\overline{A<B})$

$(A<B) = (\overline{A=B}).(\overline{A>B})$

derived outputs

$(A>=B) = (A>B) + (A=B)$

$(A<=B) = (A<B) + (A=B)$

$(A\neq B) = (A<B) + (A>B)$

The comparison of the magnitude of two numbers A and B is a standard operation in a microprocessor and is therefore included in the CPU as a hardware component. The result of the comparison is often used as a condition to make decisions.

A typical design application is the end detection of a modulo-N counter. When the counter reaches the value N−1,th e binary counter is given a reset to restart the binary sequence until N−1. When this must be done in a programmable way (the value of N can be selected), an identity comparator can be used. The actual value of the binary counter is compared with the value N−1 stored in a register (that can be programmed). The identity comparator detects when both values are equal. The resulting binary signal can be used to reset the binary counter.

The identity comparator only detects if the two numbers A and be are equal. It has one output: A=B.

A magnitude comparator also signals if the number A is greater than or less than B. It has three exclusive outputs:A>B, A=B, A<B.Only one of these outputs can be active.With the knowledge of two outputs the third one can be determined:

- $A>B = (\overline{A=B}).(\overline{A<B}) = \overline{(A=B)+(A<B)}$
- $A<B = (\overline{A=B}).(\overline{A>B}) = \overline{(A=B)+(A>B)}$
- $A=B = (\overline{A>B}).(\overline{A<B}) = \overline{(A>B)+(A<B)}$

One of the three outputs is redundant. It is sufficient to calculate two of the three outputs to save hardware.

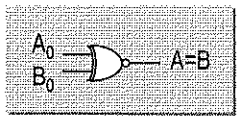
These outputs can be combined to generate other comparisons:

- (A>=B) = (A>B) + (A=B)
- (A<=B) = (A<B) + (A=B)
- (A≠B) = (A<B) + (A>B)

2.1 Identity Comparator

1-bit Identity Comparator

B_0 \ A_0	0	1
0	1	0
1	0	1

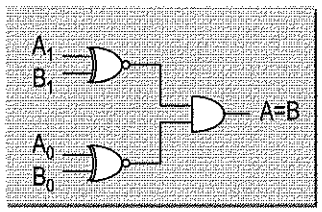


$$(A=B)=\overline{A_0}.\overline{B_0}+A_0.B_0=\overline{A_0 \oplus B_0}$$

2^1 PT of 2 inputs

2-bit Identity Comparator

B_1B_0 \ A_1A_0	00	01	10	11
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1



$$(A=B)=\overline{A_1}.\overline{A_0}.\overline{B_1}.\overline{B_0}+\overline{A_1}.A_0.\overline{B_1}.B_0+A_1.\overline{A_0}.B_1.\overline{B_0}+A_1.A_0.B_1.B_0=(\overline{A_1 \oplus B_1}).(\overline{A_0 \oplus B_0})$$

2^2 PT of 4 inputs

Identity Comparator

- ***1-bit identity comparator***

This comparator detects if two bits are equal. There are two Product Terms with each two inputs. There is no mineralization possible in the K-map.The resulting function is an XNOR.

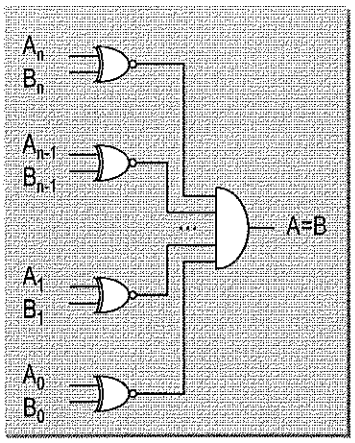
- ***2-bit identity comparator***

This comparator detects if two 2-bit numbers are equal. Also in this case there is no minimization possible in the K-map. The resulting function contains four Product Terms with each four inputs. The function can be rewritten as an AND of two XNORS. When all the corresponding bits are equal, the numbers are also equal.

n-bit Identity Comparator

2^n PT

each PT has 2.n inputs



n-input AND: multi-level implementation (tree structure)

• ***n-bit identity comparator***

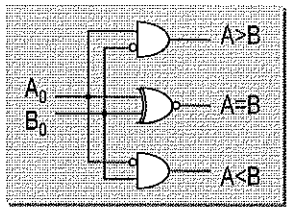
The previous result can be generalized for an n-bit identity comparator. When all the corresponding bits are equal, the numbers are also equal.T his results in an AND of n XNOR functions. The n-bit AND can be realised as an AND-tree (multiple k-input AND gates in a tree arrangement, with k<n).

For an and-or-invert implementation, 2^n Product Terms are needed, each with 2.n inputs.

2.2 Magnitude Comparator

1-bit Magnitude Comparator

B_0 \ A_0	0	1
0	=	>
1	<	=



>	=	<
1	0	0
0	1	0
0	0	1

exclusive outputs

2-bit Magnitude Comparator

B_1B_0 \ A_1A_0	00	01	10	11
00	=	>	>	>
01	<	=	>	>
10	<	<	=	>
11	<	<	<	=

B_1B_0 \ A_1A_0	00	01	10	11
00	$A_1 = B_1$		$A_1 > B_1$	
01				
10	$A_1 < B_1$		$A_1 = B_1$	
11				

A_1,B_1	A_0,B_0	A>B	A=B	A<B
$A_1>B_1$	x	1	0	0
$A_1<B_1$	x	0	0	1
$A_1=B_1$	$A_0>B_0$	1	0	0
$A_1=B_1$	$A_0<B_0$	0	0	1
$A_1=B_1$	$A_0=B_0$	0	1	0

Magnitude Comparator

- ***1-bit magnitude comparator***

This comparator makes a comparison of two bits and gives three outputs: greater than, equal, less than. The A>B and A<B outputs each have only one entry in the K-map and can be implemented with one Product Term of two inputs.

The three outputs are exclusive: one and only one of the outputs can be active at any time. This means that the result contains redundancy. When two outputs are known, the third one can be determined.

- ***2-bit magnitude comparator***

This comparator makes a comparison of two 2-bit numbers.

When the two Most Significant Bits are different ($A_1 \neq B_1$), the A>B and A<B results are only determined by the comparison of these MSBits.

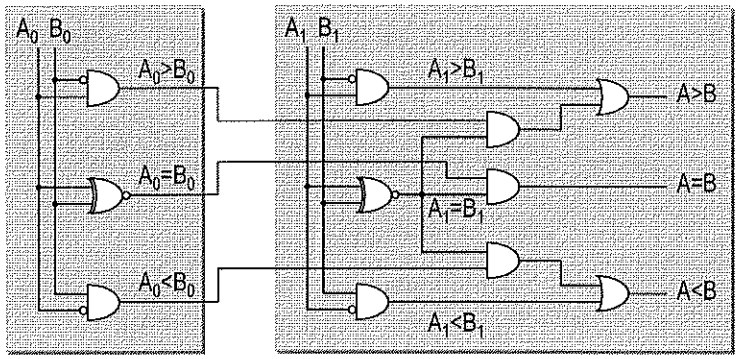
When the two Most Significant Bits are equal ($A_1 = B_1$), the output is determined by the comparison of the LSBits.

2-bit Magnitude Comparator - Serial

B_1B_0 \ A_1A_0	00	01	10	11
00	=	>	>	>
01	<	=	>	>
10	<	<	=	>
11	<	<	<	=

B_1B_0 \ A_1A_0	00	01	10	11
00	$A_1 = B_1$	$A_1 > B_1$		
01				
10	$A_1 < B_1$	$A_1 = B_1$		
11				

A_1,B_1	A_0,B_0	A>B	A=B	A<B
$A_1>B_1$	x	1	0	0
$A_1<B_1$	x	0	0	1
$A_1=B_1$	$A_0>B_0$	1	0	0
$A_1=B_1$	$A_0<B_0$	0	0	1
$A_1=B_1$	$A_0=B_0$	0	1	0



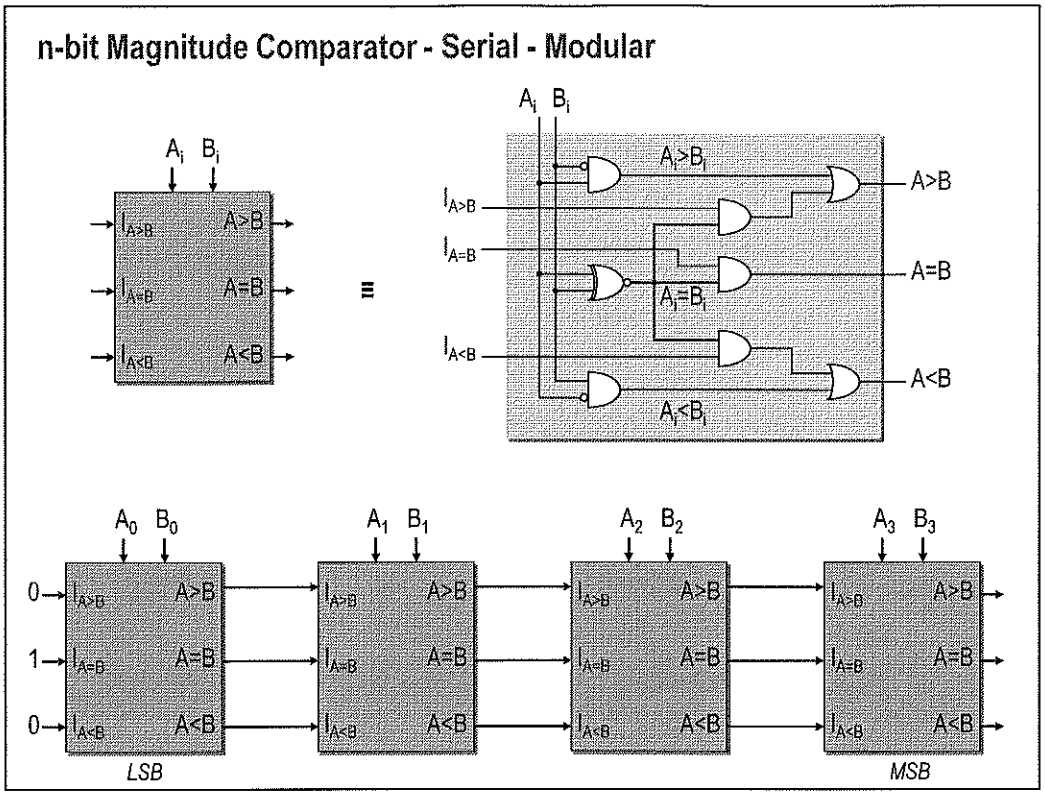
• ***2-bit magnitude comparator***

This comparator makes a comparison of two 2-bit numbers.

When the two Most Significant Bits are different ($A_1 \neq B_1$), the A>B and A<B results are only determined by the comparison of these MSBits.

When the two Most Significant Bits are equal ($A_1 = B_1$), the output is determined by the comparison of the LSBits.

These observations result in a serial (multi-level) implementation of the 2-bit magnitude comparator. Two separate 1-bit magnitude comparators can be seen. The MSB-comparator uses the result of the previous stage when the MSBits are equal.



• ***n-bit magnitude comparator - serial (cascade)***

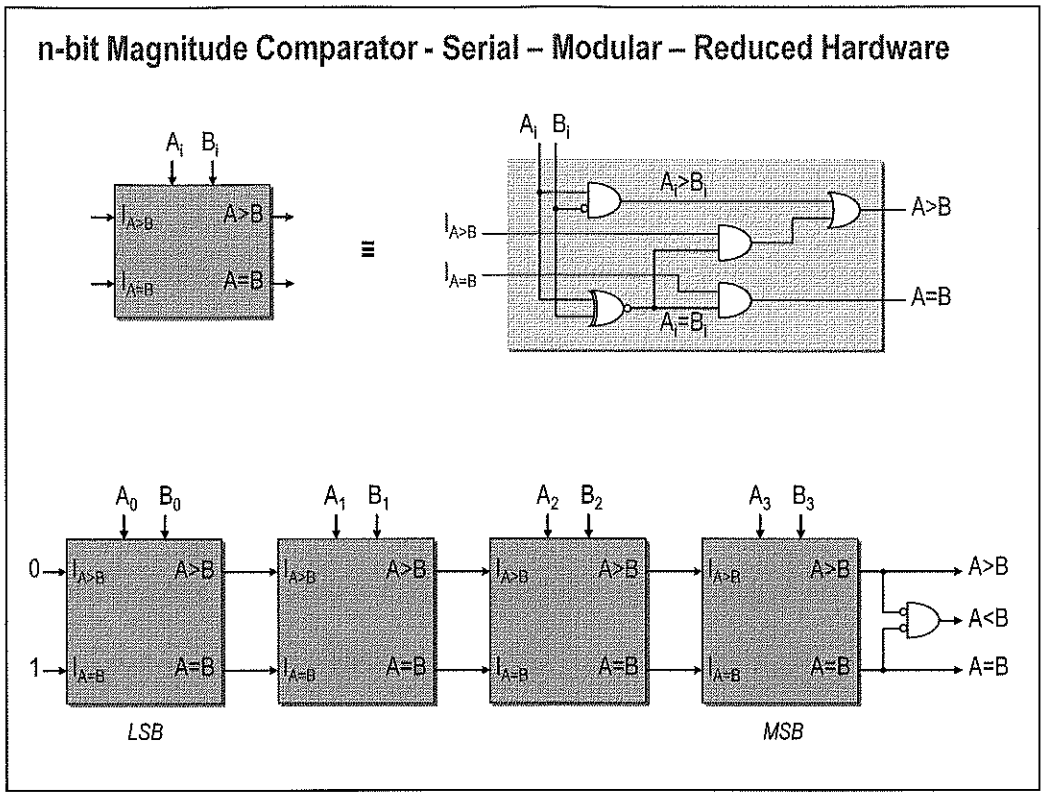
The previous result can be generalized for an n-bit magnitude comparator.

The n-bit comparator can be realised as a cascade of 1-bit comparator modules. Each module contains a 1-bit magnitude comparator. When the current bits are equal, the output is determined by the outputs of the previous module (I-inputs).

The largest critical path of the comparator occurs when all but the LSBits are equal. In that case the result of the LSBits determine the comparator result:

$$t_{critical\ path} = t_{XNOR} + n.t_{AND}$$

The throughput of the comparator can be increased with the use of pipelining. By inserting pipeline registers between the 1-bit comparator modules, the clock speed of the magnitude comparator can be increased. To synchronise (align) the data bits (A_i,B_i) with the module outputs, compensating registers must be introduced.

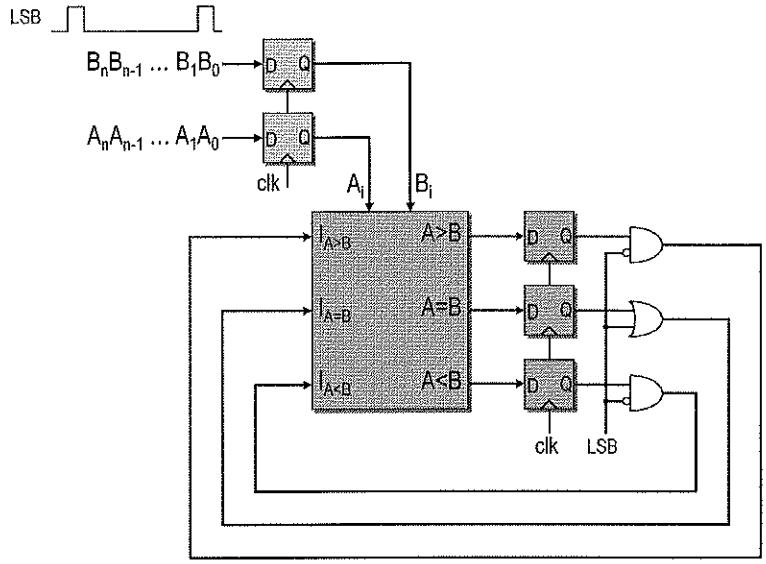


• ***n-bit magnitude comparator - serial (cascade) - reduced hardware***

The comparator hardware can be reduced, based on the observation that one of the three basic outputs (>, =, <) is redundant. Based on only two outputs, all possible inequalities can be calculated.

Each module now contains 4 inputs and two outputs. In an FPGA this can be realised with two LUTs (=Look Up Table) of 4 inputs.

n-bit Magnitude Comparator - Serial - Bit-Serial (Hardware Re-use)



• n-bit magnitude comparator - bit-serial – hardware re-use

An n-bit magnitude comparator can also be implemented with a single 1-bit comparator module. The data bits of the numbers A and B are presented to the module in a serial way, starting with the LSBits. The intermediate outputs of the module are stored in a register and used in the next clock cycle as an input for the module (I-inputs) . After the MSBits are compared by the module, the final outputs are stored in the same register.

When a new number-set is presented to the comparator module, the result of the comparison of the previous numbers may not be used as input for the module (I-inputs). Therefore, at the clock cycle where the LSBits are presented to the 1-bit comparator module (indicated by the active level of the LSB signal), the I-inputs of the module are reset to zero.

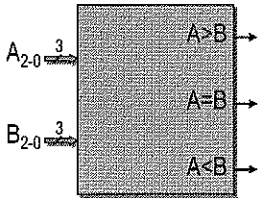
The principle of re-using a 1-bit module for the processing of an n-bit number is called hardware re-use.

3-bit Magnitude Comparator - Parallel

$A_2A_1A_0$

$B_2B_1B_0$	000	001	010	011	100	101	110	111
000	=	>	>	>	>	>	>	>
001	<	=	>	>	>	>	>	>
010	<	<	=	>	>	>	>	>
011	<	<	<	=	>	>	>	>
100	<	<	<	<	=	>	>	>
101	<	<	<	<	<	=	>	>
110	<	<	<	<	<	<	=	>
111	<	<	<	<	<	<	<	=

A_2,B_2	A_1,B_1	A_0,B_0	A>B	A=B	A<B
$A_2>B_2$	x	x	1	0	0
$A_2<B_2$	x	x	0	0	1
$A_2=B_2$	$A_1>B_1$	x	1	0	0
	$A_1<B_1$	x	0	0	1
	$A_1=B_1$	$A_0>B_0$	1	0	0
		$A_0<B_0$	0	0	1
		$A_0=B_0$	0	1	0



In a CPLD, combinatorial functions with a lot of inputs can be realised in an and-or-invert architecture. In this case a parallel realisation of a comparator is more efficient.

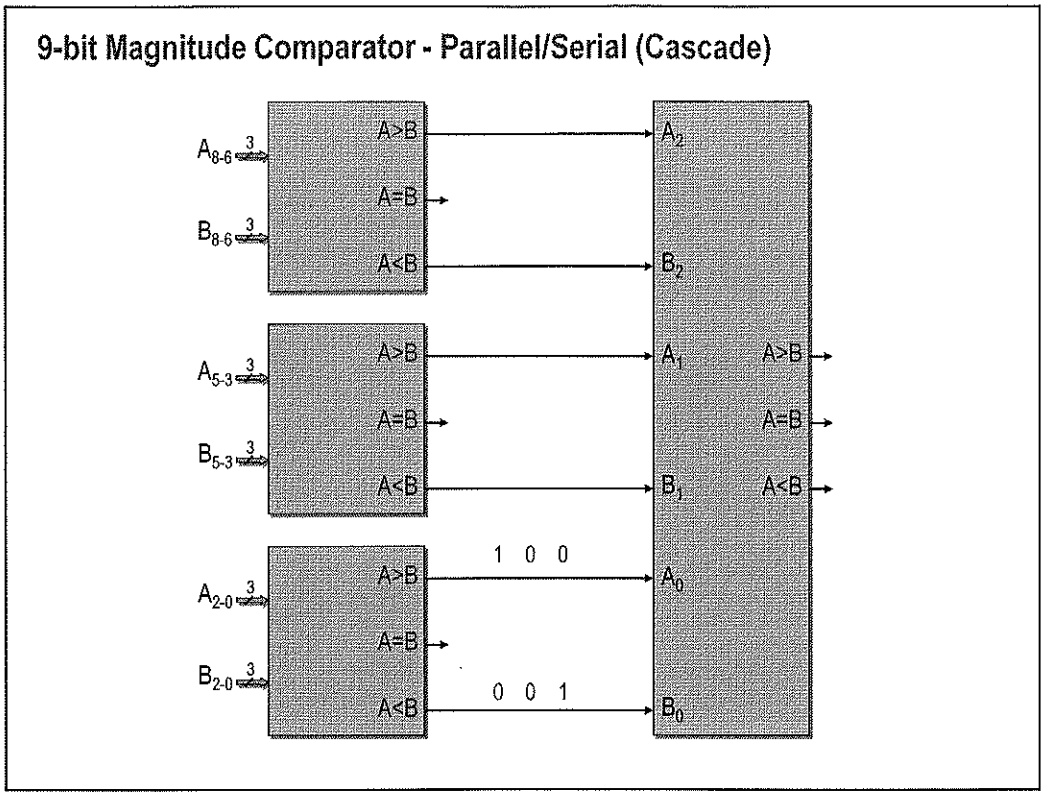
• ***3-bit magnitude comparator - parallel***

When the two Most Significant Bits are different ($A_2 \neq B_2$), the A>B and A<B results are only determined by the comparison of these MSBits.

When the two Most Significant Bits are equal ($A_2=B_2$), the output is determined by the comparison of the bits A_1 and B_1.

When the bits A_1 and B_1 are equal ($A_1=B_1$), the comparison the LSBits A_0 and B_0 is needed to determine the result.

The comparator consists of two (three) outputs with each 6 inputs.



• ***9-bit magnitude comparator – parallel/serial***

A 9-bit comparator can be build as a combination of for 3-bit modules.

Three independent 3-bit modules are each used to compare a group of 3 successive bits. The results of these modules (A>B and A<B) can directly be used as inputs for the forth 3-bit module.

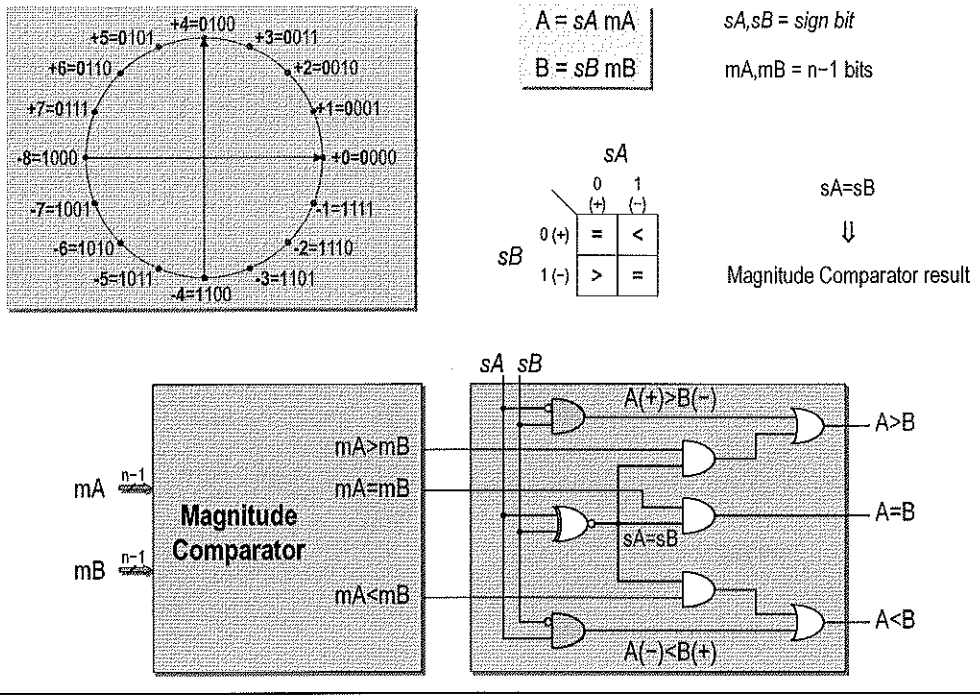
A > B:1 0 0

A < B:0 0 1

↓ ↓ ↓

> = <

2.3 Twos Complement Comparator



Twos Complement Comparator

In twos complement notation the MSBits represent the sign bit (sA, sB):

- sA = 0 = positive number
- sA = 1 = negative number

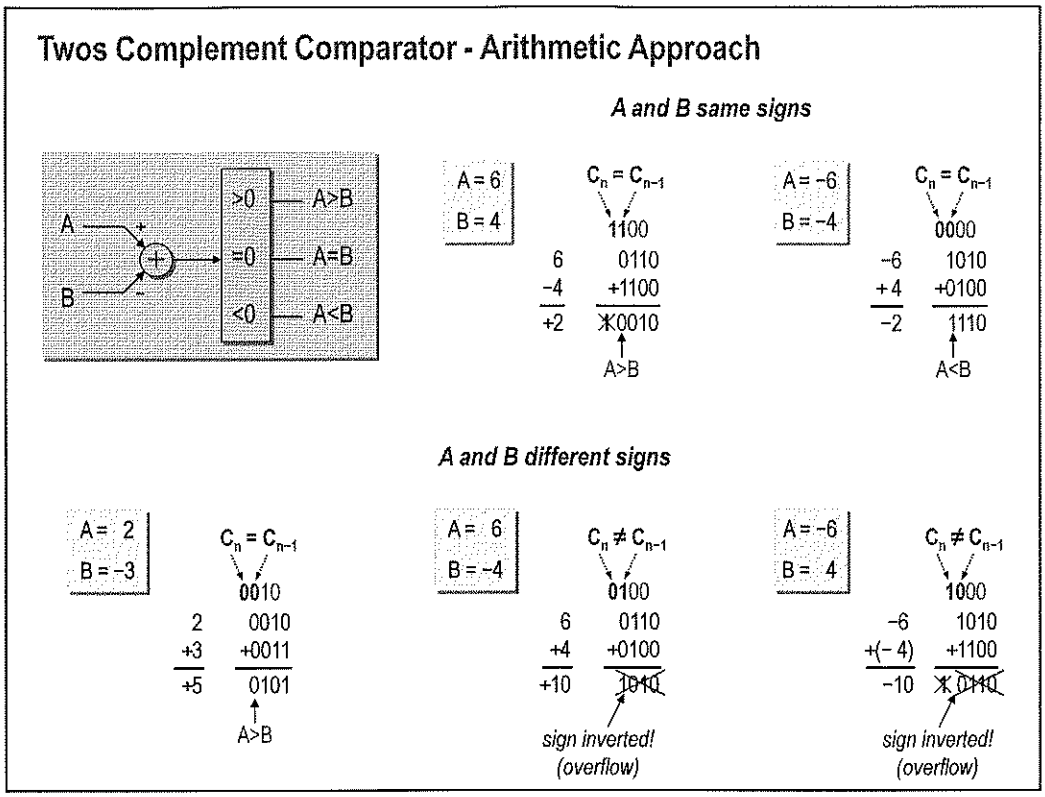
Due to this convention, a special handling of the sign bits is needed:

positive number (sign = 0) > negative number (sign = 1)

This convention is opposite to the convention of the magnitude comparator:

magnitude of 0 < magnitude of 1

The rest of the bits (mA, mB) can be handled by a conventional magnitude comparator.



• ***twos complement comparator – arithmetic approach***

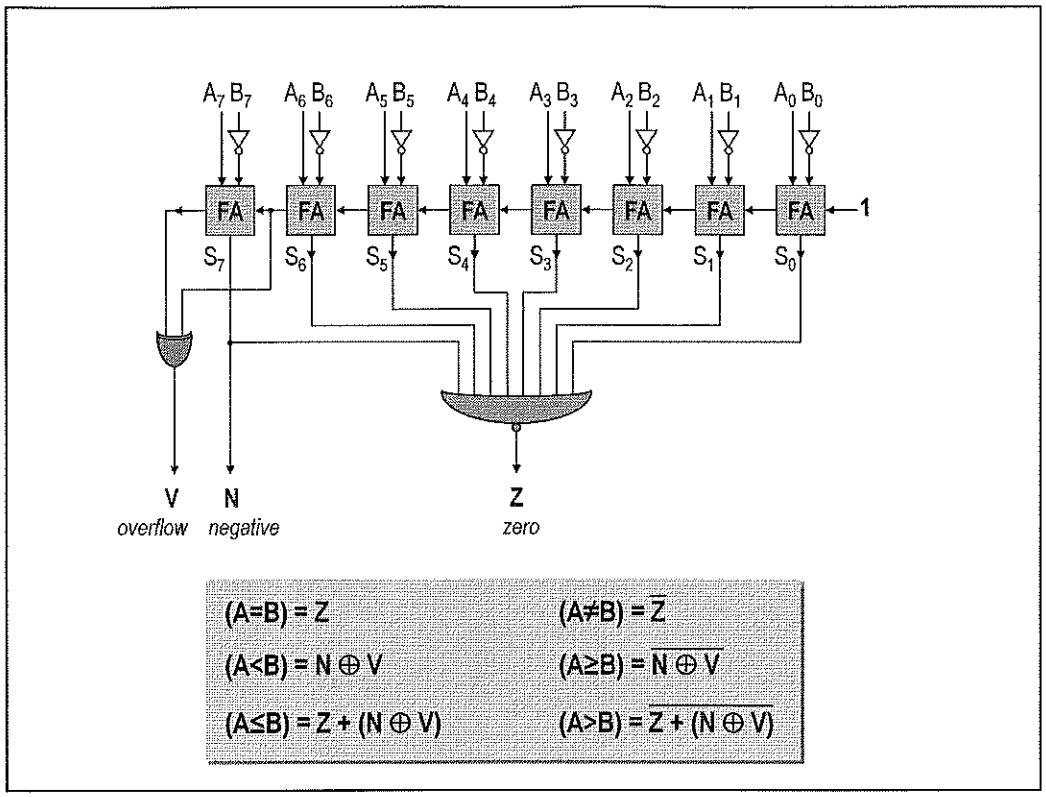
In an ALU, an adder/subtractor is already available. A subtraction can be used to make a comparison of 2 twos complement numbers A and B:

$$A - B > 0 \leftrightarrow A > B$$

$$A - B = 0 \leftrightarrow A = B$$

$$A - B < 0 \leftrightarrow A < B$$

When A and B have different signs, the subtraction A − B results in a number larger in magnitude than A or B with the same sign as A. When overflow occurs, the sign of the result is inverted and wrong conclusions can be made. Therefore, the sign of the result must be inverted when overflow occurs.



Three status flags of the result of the subtraction A−B are needed to determine the comparison result:

N = true when negative (A<B, when no overflow)

V = true when overflow (inversion of the sign bit is needed to compensate the overflow effect)

Z = true when zero (A=B)

A=B, the two numbers are equal, when the result of the subtraction A−B is zero:

$$(A=B) = Z$$

A<B, A is less than B, when the result of the subtraction A−B is negative and no overflow occurs, or when the result of the subtraction A−B is positive and an overflow occurs:

$$(A<B) = N \oplus V$$

The other comparator results can be determined as a combination of these two results.

The critical path of the comparator is determined by the carry-propagation in the binary adder:

$$t_{critical\ path} = n.t_{carry} + t_{NOR}$$

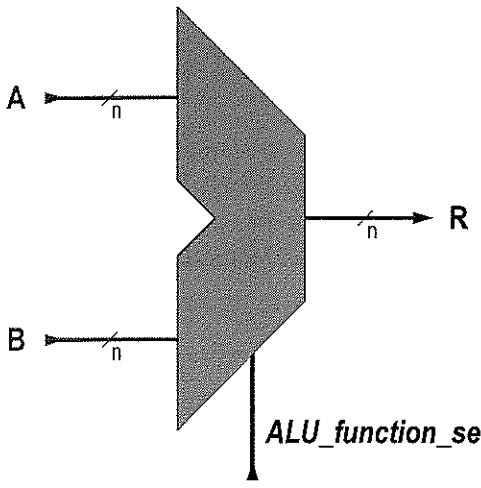
HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT SINT-KATELIJNE-WAVER

Digitale Synthese
Arithmetic Functions
ALU

EmSD
Embedded System Design

ir. J. Meel
march 2006

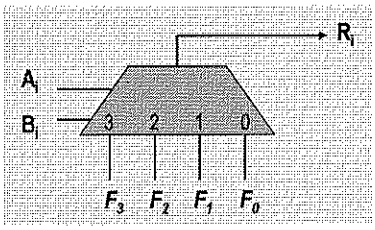
1. ALU = Arithmetic Logic Unit



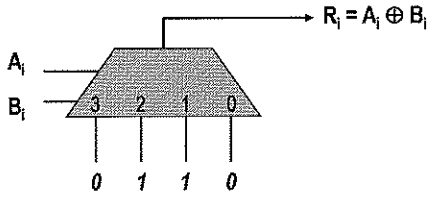
µcode { **ARITHMETIC** (complement, increment, decrement, add, subtract, ...)
LOGIC (inv, and, nand, or, nor, xor, xnor, ...)

2. Logic Function Unit (Multiplexer)

2.1 Bit-Slice



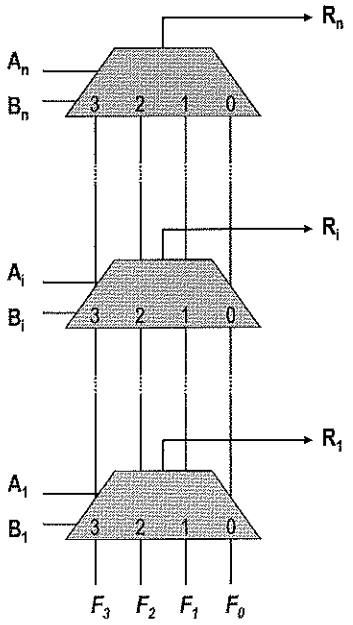
(truth table values)



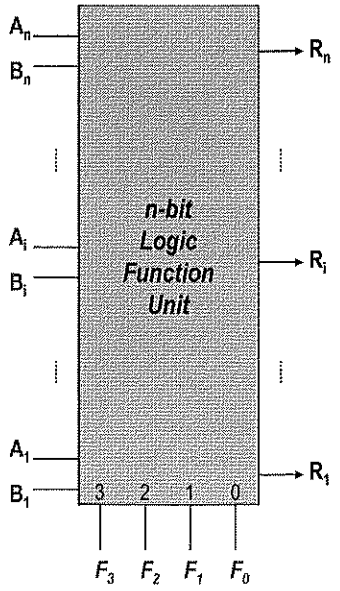
$\mu code = 6$

A_i	1	1	0	0	
B_i	1	0	1	0	
µcode	F_3	F_2	F_1	F_0	function
0	0	0	0	0	zero
1	0	0	0	1	$\overline{A_i + B_i}$ = nor
2	0	0	1	0	$\overline{A_i}B_i$
3	0	0	1	1	$\overline{A_i}$ = inv(A_i)
4	0	1	0	0	$A_i\overline{B_i}$
5	0	1	0	1	$\overline{B_i}$ = inv(B_i)
6	0	1	1	0	$A_i \oplus B_i$ = xor
7	0	1	1	1	$\overline{A_i B_i}$ = nand
8	1	0	0	0	$A_i B_i$ = and
9	1	0	0	1	$\overline{A_i \oplus B_i}$ = xnor
10	1	0	1	0	B_i
11	1	0	1	1	$\overline{A_i} + B_i$
12	1	1	0	0	A_i
13	1	1	0	1	$A_i + \overline{B_i}$
14	1	1	1	0	$A_i + B_i$ = or
15	1	1	1	1	one

2.2 n-bit Logic Function Unit

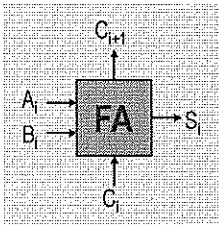


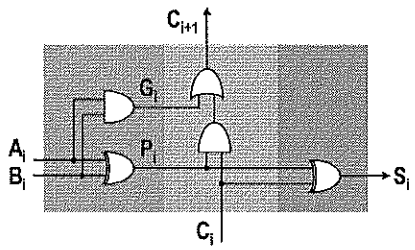
bit-wise logic



3. Arithmetic Function Unit (Adder)

3.1 Full Adder (Bit-Slice)

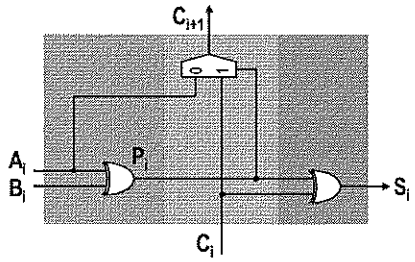




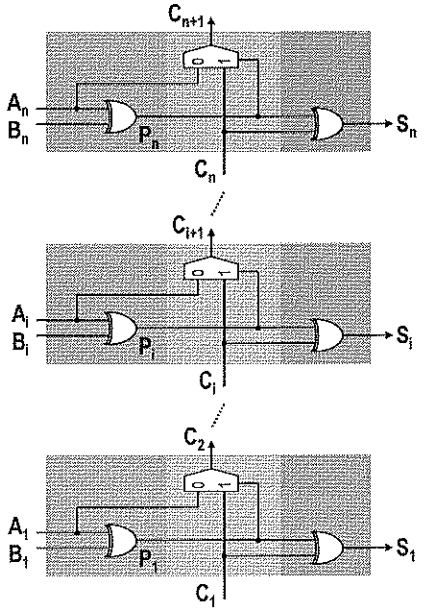
$$C_{i+1} = A_i.B_i + (A_i \oplus B_i).C_i \Rightarrow C_{i+1} = G_i + P_i.C_i$$

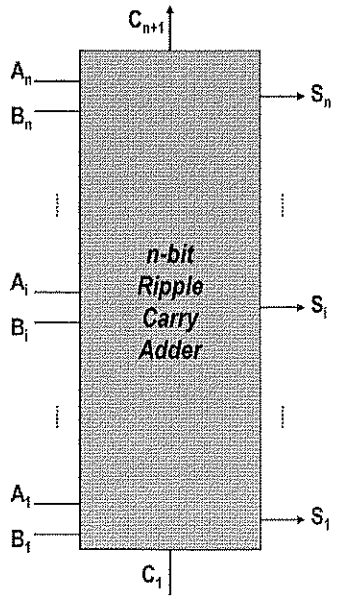
$$S_{i+1} = A_i \oplus B_i \oplus C_i \Rightarrow S_{i+1} = P_i \oplus C_i$$

S_{i+1}	A_i	B_i	C_{i+1}	
C_i	0	0	0	$K_i = \overline{A_i}.\overline{B_i}$ = ***kill***
$\overline{C_i}$	0	1	C_i	$P_i = A_i \oplus B_i$ = ***propagate***
$\overline{C_i}$	1	0	C_i	
C_i	1	1	1	$G_i = A_i.B_i$ = ***generate***

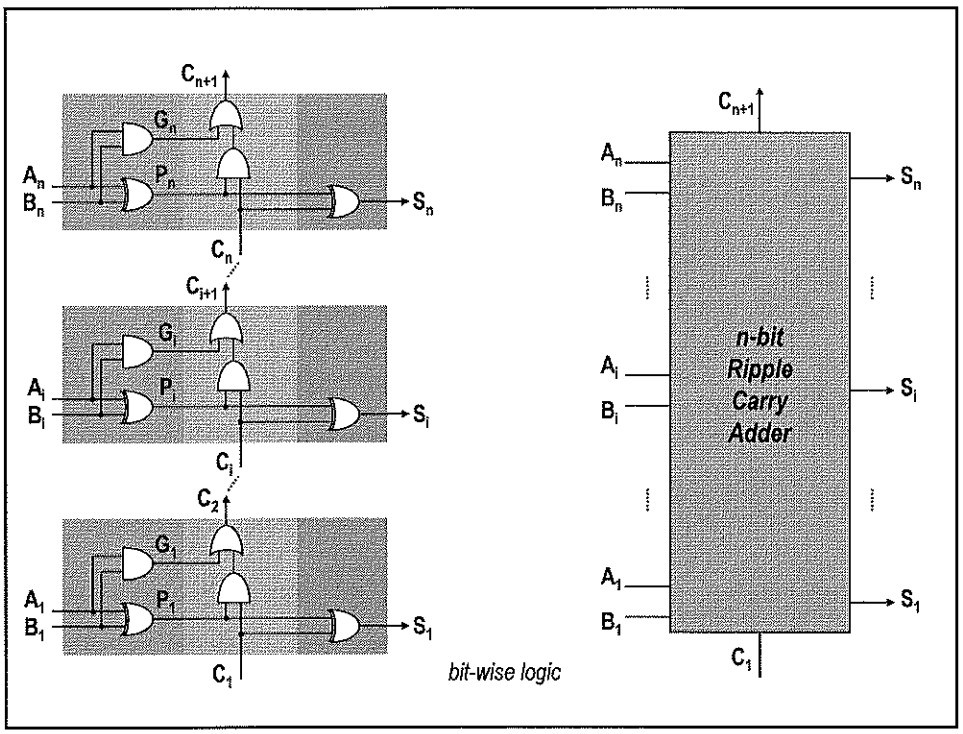


3.2 n-bit Ripple Carry Adder



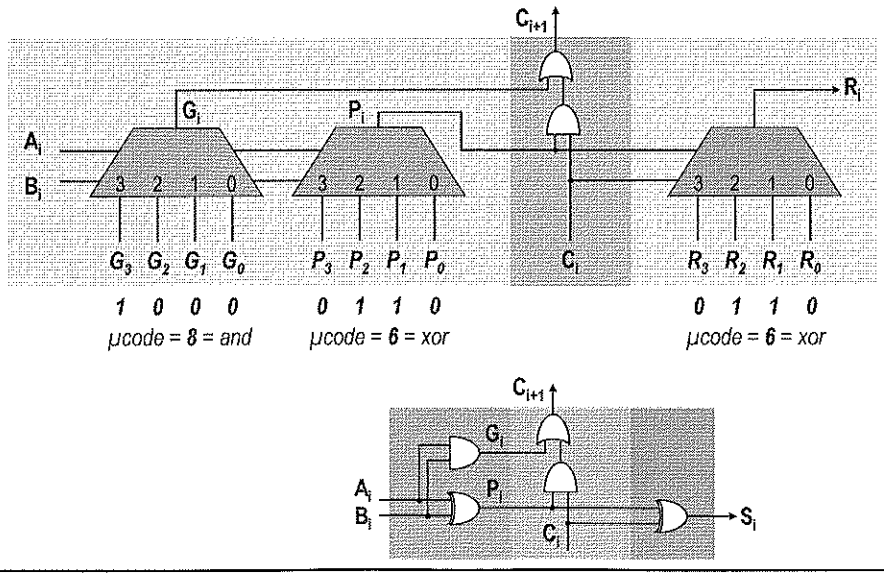


bit-wise logic

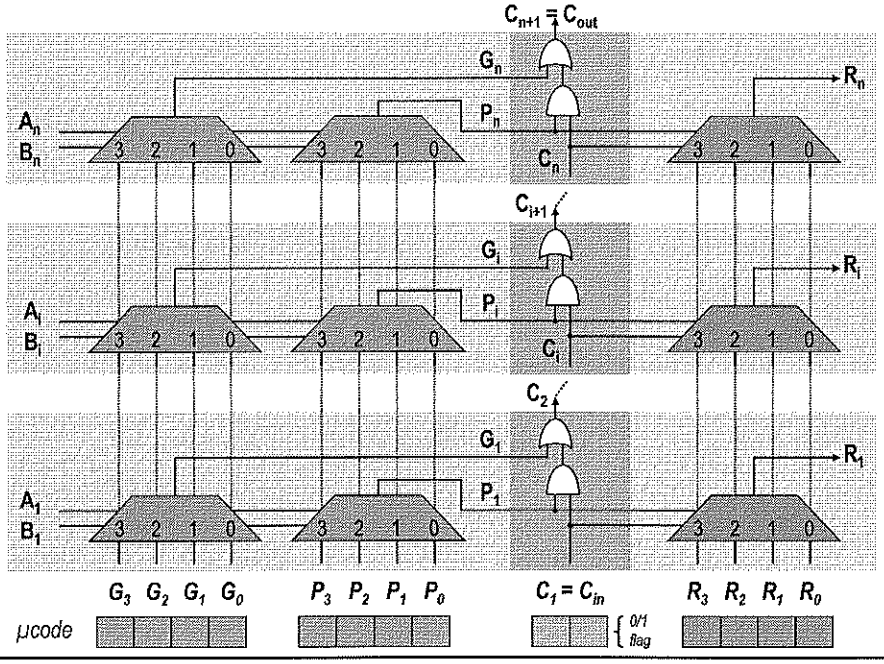


4. Arithmetic Logic Unit (Generalized Adder)

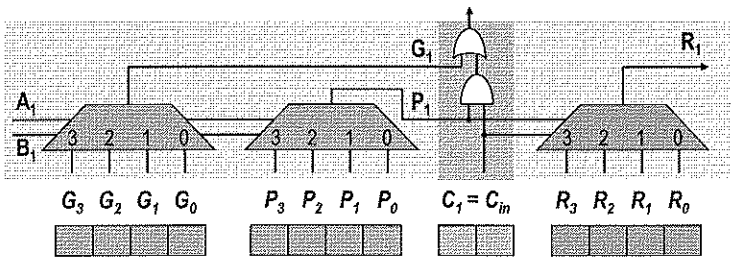
4.1 Generalized Full Adder (Bit-Slice)



4.2 n-bit Arithmetic Logic Unit



4.3 n-bit Arithmetic Logic Unit - Operations



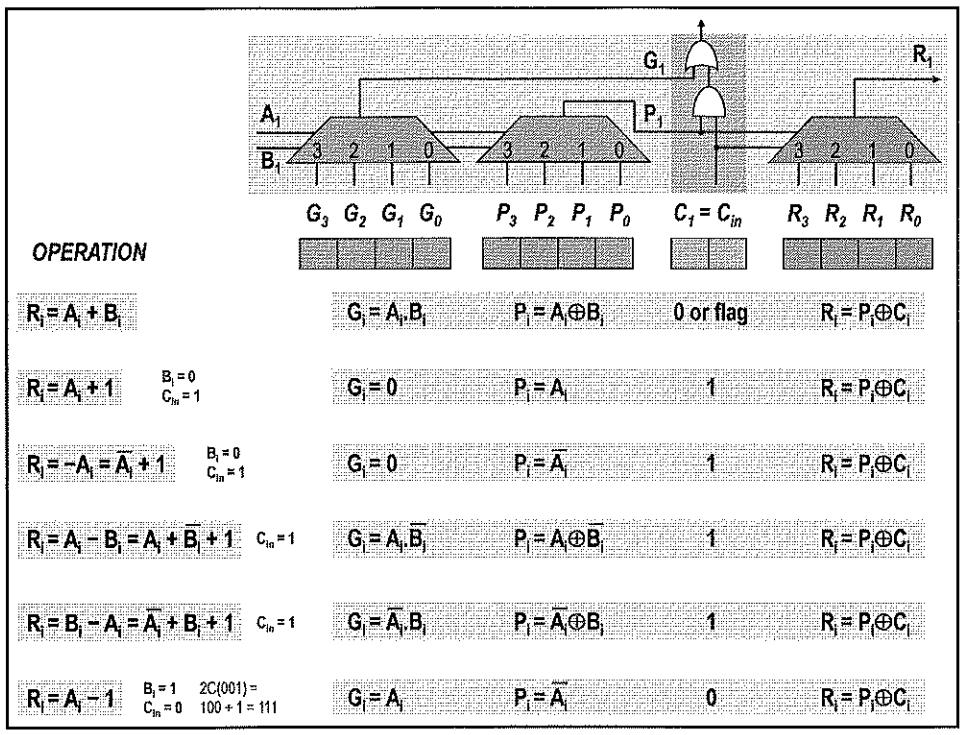
OPERATION

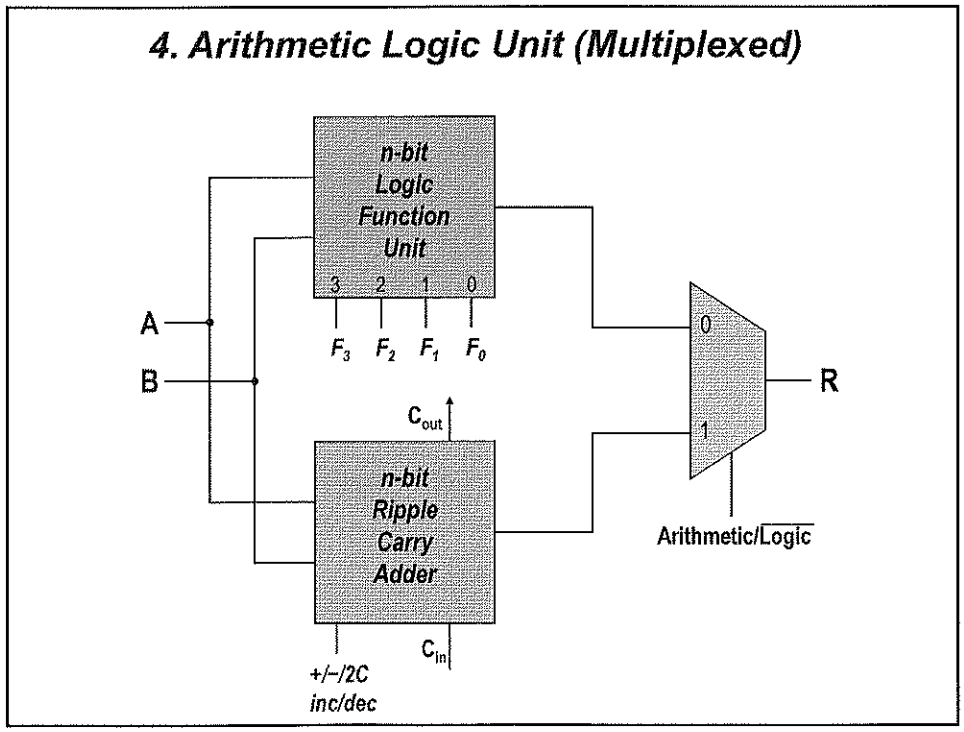
LOGIC

$R_i = f(A_i,B_i)$	$G_i = x$	$P_i = f(A_i,B_i)$	$R_i = P_i$

ARITHMETIC

$R_i = A_i + B_i$	$G_i = A_i.B_i$	$P_i = A_i \oplus B_i$	0 or flag	$R_i = P_i \oplus C_i$


G1
R1
A1
B1
P1
3 2 1 0
3 2 1 0
3 2 1 0
G3 G2 G1 G0
P3 P2 P1 P0
C1 = Cin
R3 R2 R1 R0
OPERATION
Ri = Ai + Bi
Gi = Ai.Bi
Pi = Ai⊕Bi
0 or flag
Ri = Pi⊕Ci
Ri = Ai + 1
Bi = 0
Cin = 1
Gi = 0
Pi = Ai
1
Ri = Pi⊕Ci
Ri = −Ai = Āi + 1
Bi = 0
Cin = 1
Gi = 0
Pi = Āi
1
Ri = Pi⊕Ci
Ri = Ai − Bi = Ai + B̄i + 1
Cin = 1
Gi = Ai.B̄i
Pi = Ai⊕B̄i
1
Ri = Pi⊕Ci
Ri = Bi − Ai = Āi + Bi + 1
Cin = 1
Gi = Āi.Bi
Pi = Āi⊕Bi
1
Ri = Pi⊕Ci
Ri = Ai − 1
Bi = 1
Cin = 0
2C(001) =
100 + 1 = 111
Gi = Ai
Pi = Āi
0
Ri = Pi⊕Ci


4. Arithmetic Logic Unit (Multiplexed)
n-bit Logic Function Unit
3 2 1 0
F3 F2 F1 F0
A
B
Cout
n-bit Ripple Carry Adder
Cin
+/−/2C inc/dec
0
1
R
Arithmetic/Logic

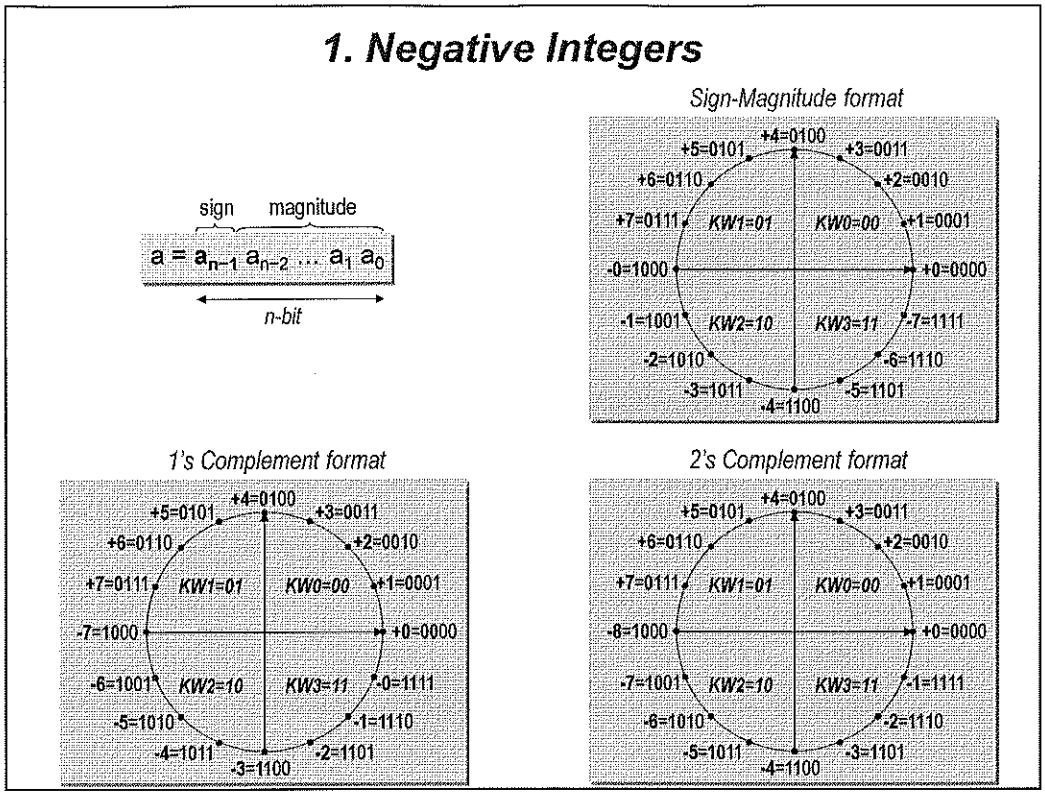
HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT SINT-KATELIJNE-WAVER

Digitale Synthese
Arithmetic Functions
MAC

EmSD Embedded System Design

ir. J. Meel

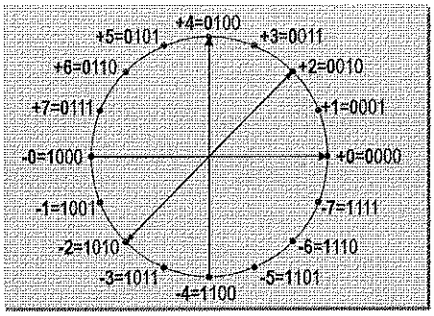
oct 2009



In mathematics, there are infinitely many positive and negative integers. However, in a practical hardware system only a fixed number of integers can be represented, based on the number of bits allocated to the representation.

As shown, 4-bit binary quantities can represent 16 unique binary numbers. An *overflow* occurs when an arithmetic operation results in a number outside the range of those that can be represented. Roughly half of these will represent positive numbers and zero, while the remainder will be negative numbers. Each of the three representation schemes (sign-magnitude, ones complement, twos complement) handles negative numbers slightly differently.

1.1 Sign-Magnitude



sign magnitude

$a = a_{n-1}\ a_{n-2} \ldots a_1\ a_0$ (number)

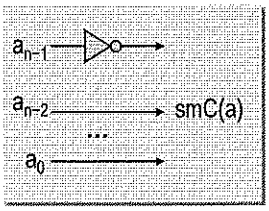
$[a] = \text{sign}(a) \cdot \sum_{i=0}^{n-2} a_i 2^i$ (value)

$a_{n-1} = 0$ $[a] = +0 \ldots 2^{n-1}-1$

$a_{n-1} = 1$ $[a] = -0 \ldots -2^{n-1}+1$ (range)

a	[a]
00000	+ 0
01001	+ 9
01111	+ 15
10000	- 0
11001	- 9
11111	- 15

calculation of S-M complement



$a = +6 = 0110_2$

$\bar{a} = -6 = 1110_2$

- negative numbers have a 1 in the MSB (sign bit)
- two representations for 0
- symmetric: # positive numbers = # negative numbers
- subtraction: comparator and subtractor

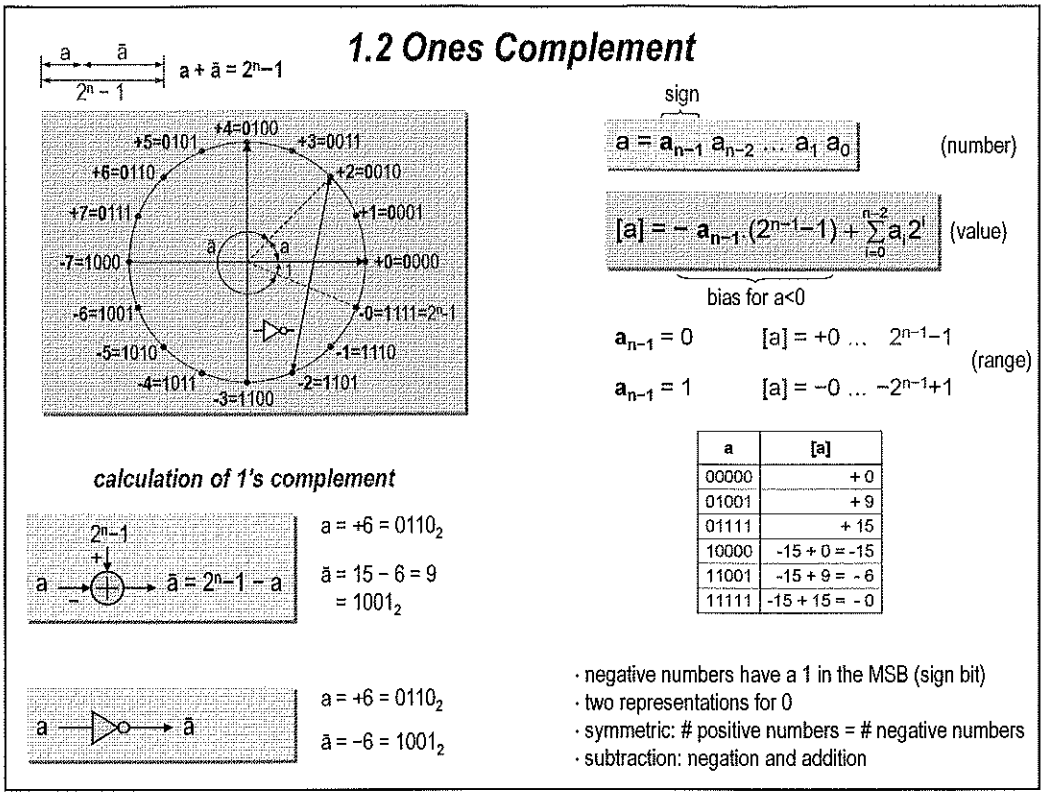
In *sign-magnitude* systems, the most significant bit (a_{n-1}) represents the number's sign, while the remaining bits represent its absolute value as an unsigned binary magnitude. If the sign bit a_{n-1} is a 0, the number is positive.If the sign bit a_{n-1} is a 1, the number is negative.

Number wheel

A 4-bit sign-magnitude number system is represented on a number wheel. The figure shows the binary numbers and their decimal integer equivalents, assuming that the numbers are interpreted as sign and magnitude. The largest positive number that can be represent in three data bits is $+7 = 2^3 - 1$. By a similar calculation, the smallest negative number is -7. There is an equal number of positive and negative numbers (symmetric).
Zero has two different representations, even though +0 and -0 don't make much sense mathematically.

Calculation of the sign-magnitude complement

Negation of a number is done simply by replacing the sign bit with its complement. The magnitude value stays unchanged.



Aones complement approach represents the positive numbers just as in the sign-magnitude representation. The only difference is in how it represents negative numbers.

Number wheel

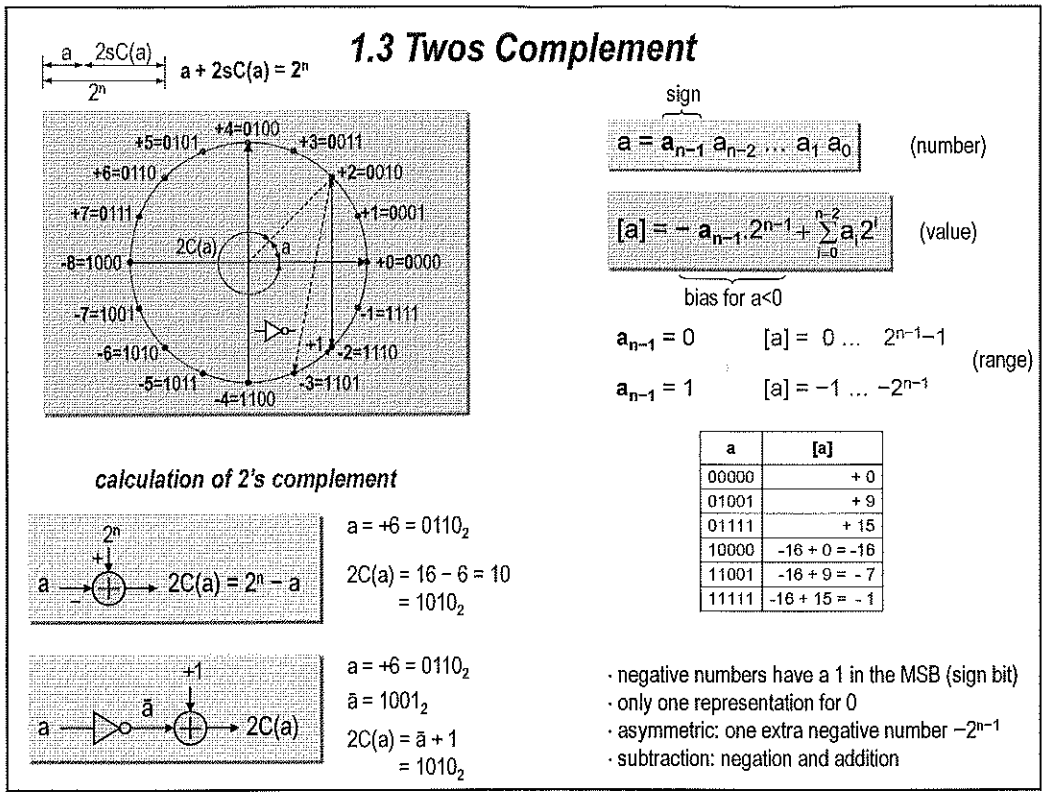
The number wheel representation of the 4-bit ones complement number system is shown. All negative numbers have a 1 in their sign bit, making it easy to distinguish between positive and negative numbers.
There are still two different representations of zero.

Calculation of the ones complement

The first procedure to derive a negative ones complement integer, denoted ā, from a positive integer, denoted a. If the word length is n bits (n = 4 in the case shown), then ā = (2^n - 1) - a.For example, in a 4-bit system, a=+7 is represented as 0111. The ones complement ā=−7 can be computed as:

$2^n=2^4$	10000
subtract 1	- 0001
	1111
subtract 7	- 0111
	1000 representation of −7

This rather complicated method is just one way to compute the negative of a ones complement number. A simpler method forms the ones complement by taking the number's bitwise complement (bitwise inversion). Thus, +7 = 0111 and −7 = 1000, +4 = 0100 and −4 = 1011, and so on.



a	[a]
00000	+ 0
01001	+ 9
01111	+ 15
10000	-16 + 0 = -16
11001	-16 + 9 = - 7
11111	-16 + 15 = - 1

A twos complement approach represents the positive numbers just as in the sign-magnitude representation. The only difference is in how it represents negative numbers.

Number wheel

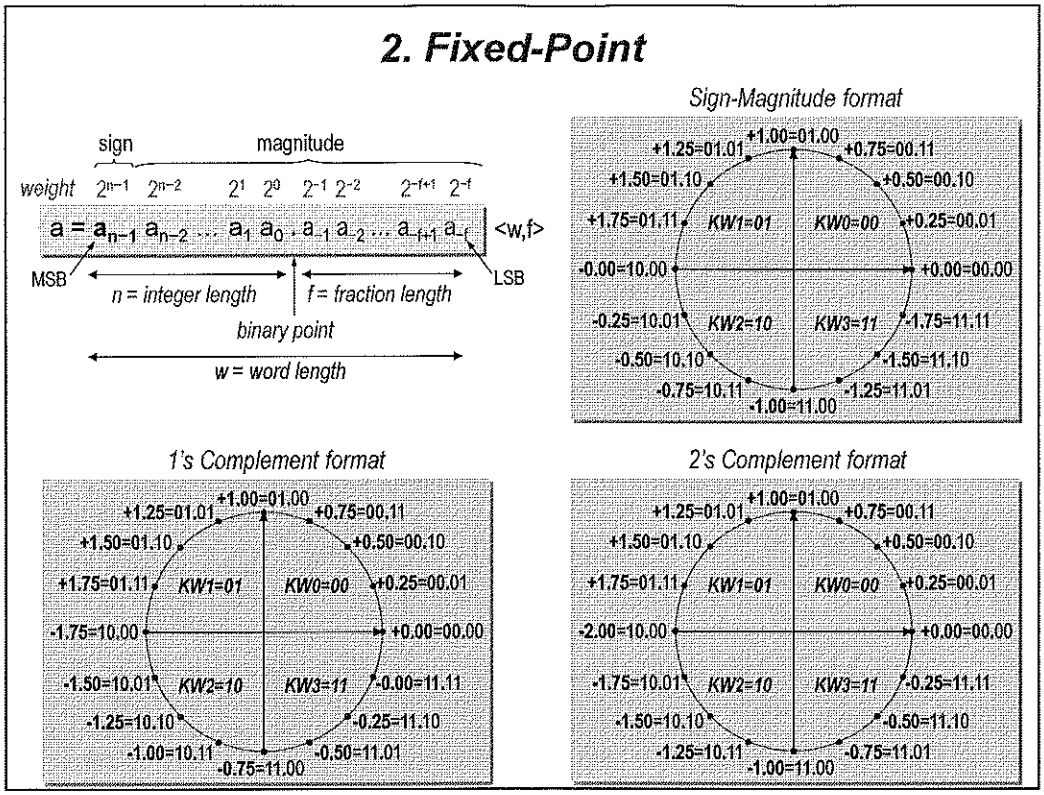
The twos complement scheme is similar to ones complement. The twos complement numbers can be derived from the ones complement representation by shifting the negative numbers one position in the clockwise direction. By this operation there is only one representation for zero.
This operation also results in one extra negative number that can now be represented: -2^{n-1} (−8 in the 4-bit case). This makes the twos complement representation asymmetric.
The negative numbers still have a 1 in their highest-order bit, the sign bit.

Calculation of the twos complement

A twos complement negative number, denoted 2C(a) is derived from its positive number a, by the equation $2C(a) = 2^n - a$, where n is the number of bits in the representation. This equation omits the ones complement step that subtracts 1 from 2^n.
For example the twos complement of +7, can be calculated as:

$2^n=2^4$	10000
subtract 7	- 0111
	1001 representation of −7

The same shortcut used to find ones complement numbers also applies for the twos complement system, but with a twist.T he number wheel suggests the scheme. Simply form the bitwise complement of the number and then add one to form its twos complement.
For example, +7 = 0111_2, its bitwise complement is 1000_2. plus 1 is 1001_2. This is the same twos complement representation of −7 as derived by the last calculation.
For the number −4, represented as 1100_2, its bitwise complement is 0011_2, plus 1 is 0100_2, This is exactly the twos complement representation of +4.
A third method two calculate the twos complement. Leave all least significant 0's and the first 1 unchanged, then bit-wise complement the remaining bits.



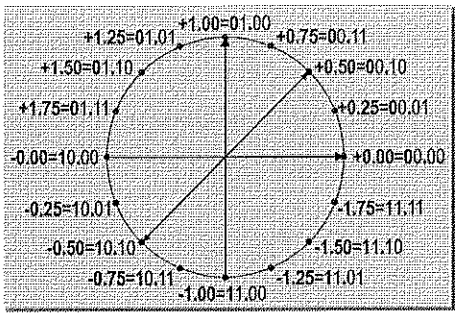
A binary number comes with a binary point. By moving the implicit "binary" point at the right of an integer number, fractions can be represented too.

The portion to the left represents an integer and the portion to the right represents fractions less than one. In fixed-point notation, the binary point is constrained to lie at a fixed position in the bit pattern, as shown.

The first bit to the left is called the sign bit. The precision of the number system is defined as the increment between two consecutive numbers and is determined by the value of the least significant bit (LSB).

Within the subclass of fixed-point representations, there are tree commonly used methods to represent binary numbers: sign-magnitude, one's complement and two's complement representation. They differ in the way they handle negative numbers.

2.1 Sign-Magnitude



sign magnitude

$a = a_{n-1}\, a_{n-2} \ldots a_0 \,.\, a_{-1} \ldots a_{-f}$ (number)

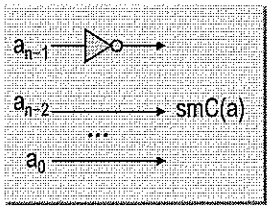
$[a] = \text{sign}(a) \cdot \sum_{i=-f}^{n-2} a_i 2^i$ (value)

$a_{n-1} = 0 \quad [a] = +0 \ldots 2^{n-1} - 2^{-f}$

$a_{n-1} = 1 \quad [a] = -0 \ldots -2^{n-1} + 2^{-f}$ (range)

a	[a]
00.000	+ 0.000
01.001	+ 1.125
01.111	+ 1.875
10.000	- 0.000
11.001	- 1.125
11.111	- 1.875

calculation of S-M complement



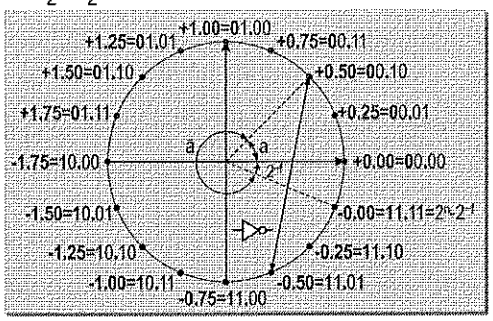
$a = +1.50 = 01.10_2$

$\bar{a} = -1.50 = 11.10_2$

- negative numbers have a 1 in the MSB (sign bit)
- two representations for 0
- symmetric: # positive numbers = # negative numbers
- subtraction: comparator and subtractor

In the *sign-magnitude* representation, the most significant bit (a_{n-1}) represents the number's sign, while the remaining bits represent its absolute value as an unsigned binary magnitude.If the sign bit a_{n-1} is a 0, the number is positive.If the sign bit a_{n-1} is a 1, the number is negative.

2.2 Ones Complement



sign

$a = a_{n-1}\, a_{n-2} \ldots a_0 . a_{-1} \ldots a_{-f}$ (number)

$[a] = -a_{n-1}(2^{n-1}-2^{-f}) + \sum_{i=-f}^{n-2} a_i 2^i$ (value)

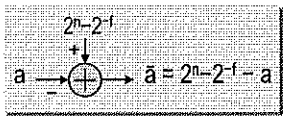
bias for a<0

$a_{n-1} = 0$ $[a] = +0 \ldots 2^{n-1}-2^{-f}$ (range)

$a_{n-1} = 1$ $[a] = -0 \ldots -2^{n-1}+2^{-f}$

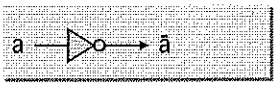
a	[a]
00.000	+ 0.000
01.001	+ 1.125
01.111	+ 1.875
10.000	-1.875 + 0 = -1.875
11.001	-1.875 + 1.125 = - 0.750
11.111	-1.875 + 1.875 = - 0.000

calculation of 1's complement



a = +1.50 = 01.10$_2$

ā = 3.75 − 1.50 = 2.25 = 10.01$_2$



a = +1.50 = 01.10$_2$

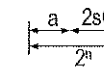
ā = −1.50 = 10.01$_2$

- negative numbers have a 1 in the MSB (sign bit)
- two representations for 0
- symmetric: # positive numbers = # negative numbers
- subtraction: negation and addition

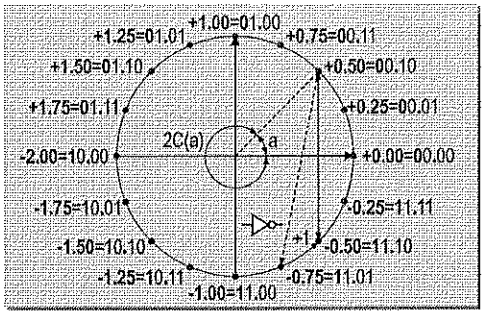
The *ones complement* representation is identical to the sign-magnitude representation for positive numbers. The only difference is in how it represents negative numbers. A negative number is formed by complementing its corresponding positive number representation.

Note that zero is now represented by 00...0.0...00 or 11...1.1...11, which is an undesired ambiguity.

2.3 Twos Complement



$a + 2sC(a) = 2^n$



sign

$a = a_{n-1}\, a_{n-2} \ldots a_0 . a_{-1} \ldots a_{-f}$ (number)

$[a] = -a_{n-1} 2^{n-1} + \sum_{i=-f}^{n-2} a_i 2^i$ (value)

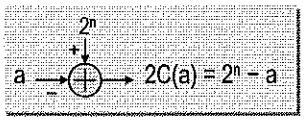
bias for a<0

$a_{n-1} = 0$ $[a] = 0 \ldots 2^{n-1} - 2^{-f}$

$a_{n-1} = 1$ $[a] = -2^{-f} \ldots -2^{n-1}$ (range)

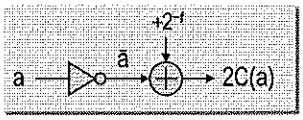
a	[a]
00.000	+ 0.000
01.001	+ 1.125
01.111	+ 1.875
10.000	-2 + 0 = -2.000
11.001	-2 + 1.125 = - 0.875
11.111	-2 + 1.875 = - 0.125

calculation of 2's complement



$a = +1.50 = 01.10_2$

$2C(a) = 4 - 1.50 = 2.50 = 10.10_2$



$a = +1.50 = 01.10_2$

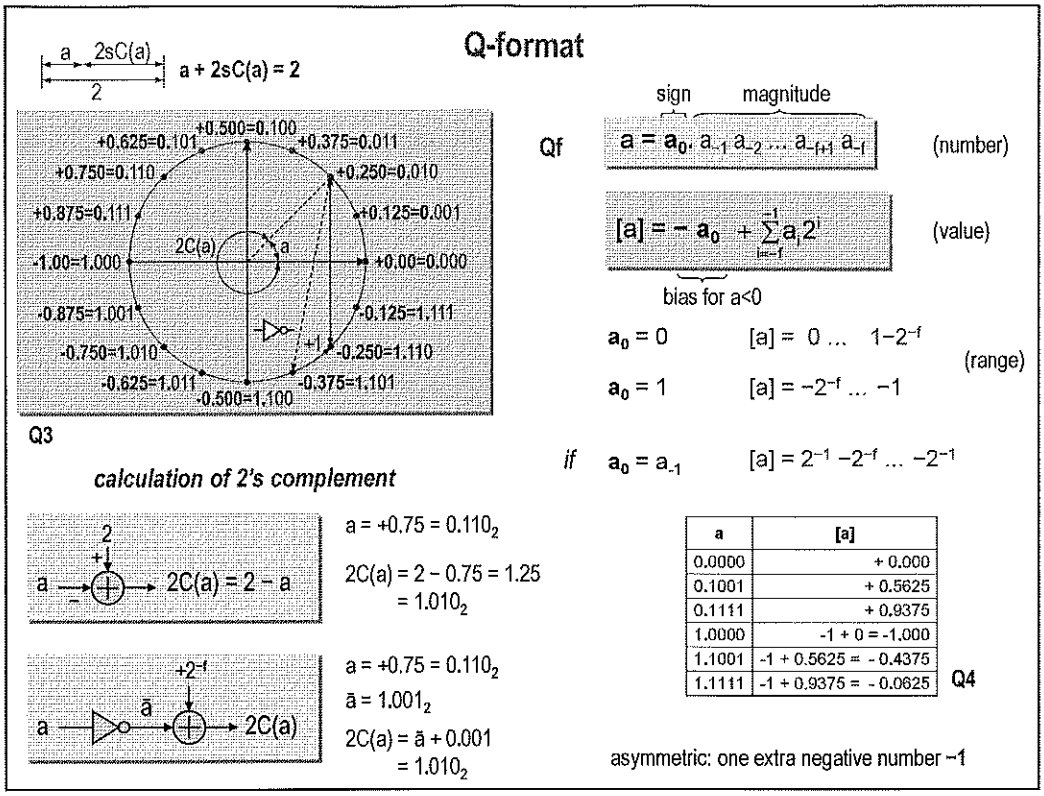
$\bar{a} = 10.01_2$

$2C(a) = \bar{a} + 0.01 = 10.10_2$

- negative numbers have a 1 in the MSB (sign bit)
- only one representation for 0
- asymmetric: one extra negative number -2^{n-1}
- subtraction: negation and addition

In the *twos complement* representation a positive numbers is identical as in the sign-magnitude representation and ones complement representation. The only difference is in how it represents negative numbers. The negative number is obtained by subtracting the corresponding positive number from 2^n.

If a two's-complement representation is used for signed fixed-point numbers, the same binary addition procedure will work for adding both signed and unsigned numbers.



The Q-format is aspecial case of the twos complement fixed-point representation. Itis apur e *fractional representation.*

Qf-format is a popular format in DSP. The most significant bit is the sign bit followed by an imaginary binary point, followed by f bits of fraction. The Qf number has a range between –1 and $1 - 2^{-f}$.

For example, the Q15 format contains 16 bit: 1 sign bit and 15 fraction bits.

$0.111111111111111 = 1 - 2^{-15} = 0.99996948242188$

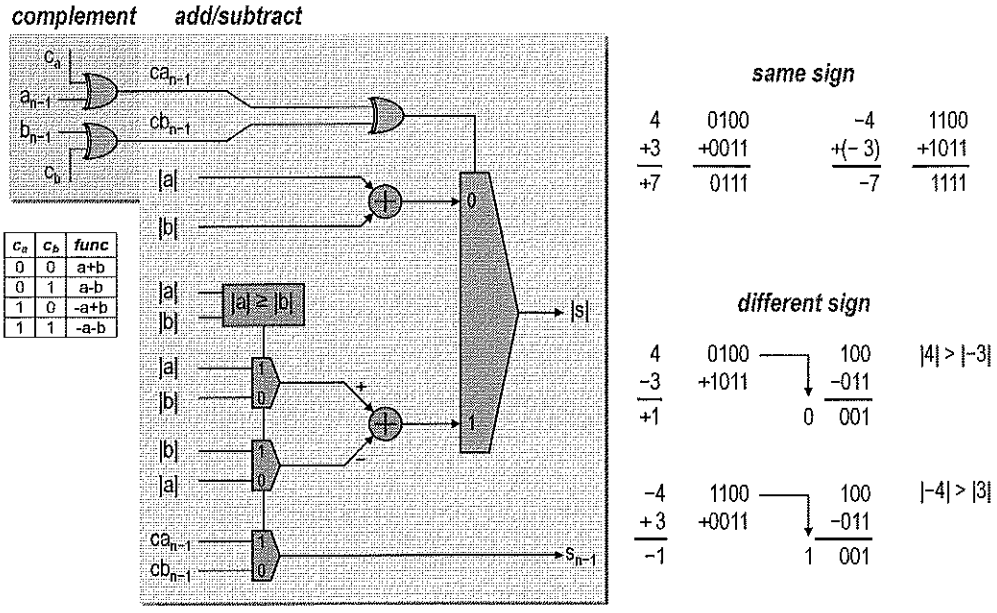
$0.100000000000000 = 0.5$

$1.000000000000000 = -1$

A negative number is obtained by subtracting its positive number from 2, hence the name 2's complement.

2. Addition and Subtraction

2.1 Sign-Magnitude



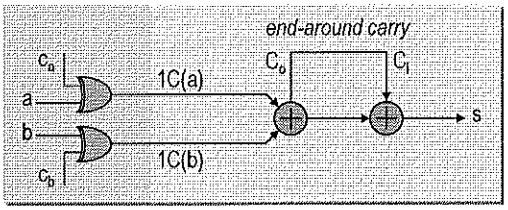
Adding two positive or two negative numbers is straightforward. Simply perform the addition and assign the result the same sign as the original operands. When the signs of the two operands are not the same, addition becomes more complex. In this case, the smaller magnitude should be subtracted from the larger. The resulting sign is the same as that of the number with the larger magnitude.

This is what makes arithmetic operations with sign-magnitude numbers so cumbersome. Any adder circuit must also include a subtractor and a comparator. The adder is used when the signs are the same, the subtractor when they differ. Subtraction is just as complicated.

Because of this burdensome complexity, hardware designers have proposed other schemes for representing negative numbers.

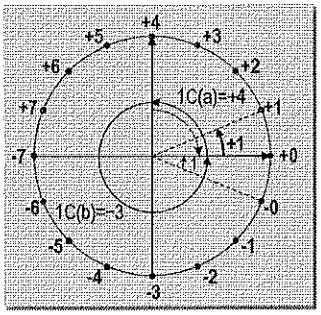
2.2 Ones Complement

complement ***add/subtract***



c_a	c_b	*func*
0	0	a+b
0	1	a-b
1	0	-a+b
1	1	-a-b

subtract = add 1's complement



same sign

4	0100	-4	1011
+3	+0011	+(-3)	+1100
+7	0111	-7	0111
			1
			1000

different sign

4	0100	-4	1011
-3	+1100	+3	+0011
+1	0000	-1	1110
	1		
	0001		

subtract 2^n add 1

end-around carry $= -2^n + 1$

$a - b \rightarrow a + \bar{b} = a + (2^n - 1 - b) = a - b + 2^n - 1$

$-a - b \rightarrow \bar{a} + \bar{b} = (2^n - 1 - a) + (2^n - 1 - b) = [2^n - 1 - (a + b)] + 2^n - 1$

carry = advance through origin **+1** = avoid counting zero twice

The advantage of ones complement numbers is the ease with which negative numbers can be computed. Subtraction is implemented by a combination of addition and negation: A − B = A + (−B). Thus, a separate subtractor circuit is not needed. However, addition is still complicated by the two zeros. This problem is solved with the twos complement representation.

End-around carry

In ones complement, −4 is represented as 1011_2 and −3 as 1100_2. When these two numbers are added, a carry-out of the high-order bit position is generated. Whenever this occurs, the carry bit must be added to the result of the sum.10 11_2 + 1100_2 yields 1 0111_2.Adding the carry-out as an LSB to the 4-bit sum, gives $0111_2 + 1 = 1000_2$. This is the representation of −7 in ones complement.

The end-around carry also happens for the sum of 4(0100_2) and −3(0011_2). This yields 1 0000_2. Adding in the carry gives 0001_2, the ones complement representation of -1.

Why does the end-around carry scheme work?

Intuitively, the carry-out of 1 means that the resulting addition advances through the origin of the number wheel. In effect, the result must be advanced by 1 to avoid counting zero twice.

More formally, the operation of the end-around carry is the equivalent of subtracting 2^n and adding 1. Consider the computation of the sum b + (−a) where b > a:

$$b - a = b + \bar{a} = b + (2^n - 1 - a) = (b - a) + 2^n - 1$$

This is exactly the situation of last example.Th e end-around carry subtracts off 2^n and adds 1, yielding the desired result of b − a.

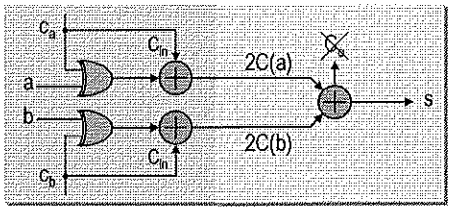
Now consider the case shown in the first example. The sum to be formed is −a + (−e), where a + e is less than $2^n - 1$. This results in the following sequence of equations:

$$-a + (-e) = \bar{a} + \bar{e} = (2^n - 1 - a) + (2^n - 1 - e) = [2^n - 1 - (a + e)] + 2^n - 1$$

After the end-around carry (subtract off 2^n and add 1), the result of the sum becomes [$2^n - 1 - (a + e)$]. This is the correct form for representing −(a + e) in ones complement form.

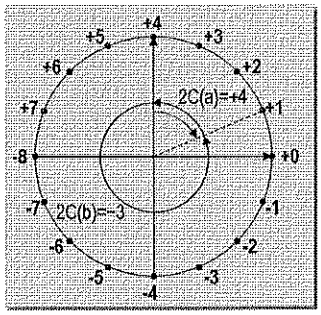
2.3 Twos Complement

complement ***add/subtract***



c_a	c_b	*func*
0	0	a+b
0	1	a-b
1	0	-a+b
1	1	-a-b

subtract
= add 2's complement



same sign

4	0100	-4	1100
+3	+0011	+(- 3)	+1101
+7	0111	-7	✗1001

different sign

4	0100	-4	1100
-3	+1101	+3	+0011
+1	✗0001	-1	1111

subtract 2^n

ignore carry = -2^n

$a - b \rightarrow a + 2C(b) = a + (2^n - b) = a - b + 2^n$

$-a - b \rightarrow 2C(a) + 2C(b) = (2^n - a) + (2^n - b) = [2^n - (a + b)] + 2^n$

carry = advance through origin **+0** (zero has only one representation)

Twos complement calculations behave very much like the ones complement method, but without the end-around carry. Subtraction is handled as before: negate the operand and perform addition. Carry-outs can still occur, but in twos complement arithmetic they are ignored.

Ignore carry-out

Summing two positive numbers, is identical to the two previous representation schemes.Summing two negative numbers is also straightforward. Simply perform binary addition, ignoring any carry-outs. Since there are no longer two representations for zero, there is no need to worry about correcting the summation. Mixed addition of positive and negative numbers is handled exactly like the other cases.

Why is it all right to ignore the carry-out?

Consider the sum b + (−a) where b > a. This can be rewritten as:

$$b + 2C(a) = b + (2^n - a) = 2^n + (b - a)$$

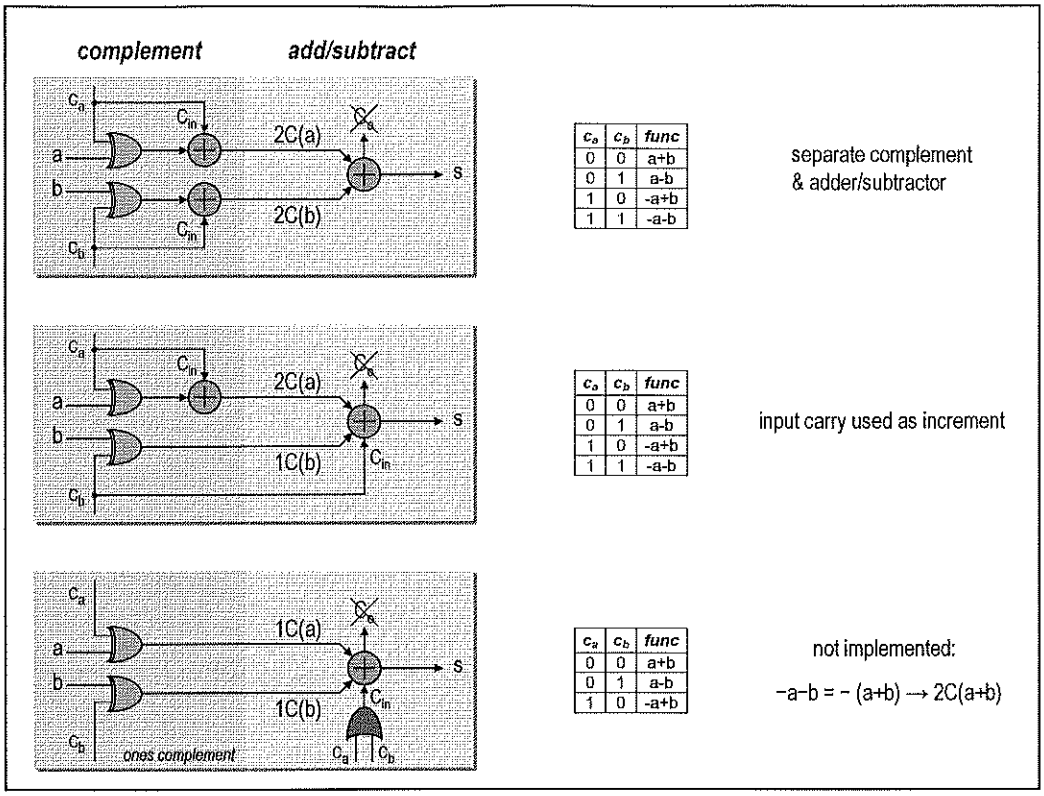
Ignoring the carry-out is equivalent to subtracting 2^n. Doing this to the foregoing expression yields the result b − a.

Consider the sum:(−a) + (−b) where a+ b is less than or equal to 2^n -1. This can be rewritten as:

$$2C(a) + 2C(b) = (2^n - a) + (2^n - b) = [2^n - (a + b)] + 2^n$$

By subtracting 2^n, the resulting form is exactly the representation of 2C(a + b), the desired twos complement representation of −(a + b).

The trade-off between twos complement and ones complement arithmetic should now be clearer. In the twos complement case, addition is simple but negation is more complex. For the ones complement system, it is easy to perform negation but addition becomes more complicated. Because twos complement only has one representation for zero, it is preferred for most digital systems.



c_a	c_b	func
0	0	a+b
0	1	a-b
1	0	-a+b
1	1	-a-b

separate complement & adder/subtractor

c_a	c_b	func
0	0	a+b
0	1	a-b
1	0	-a+b
1	1	-a-b

input carry used as increment

c_a	c_b	func
0	0	a+b
0	1	a-b
1	0	-a+b

not implemented:

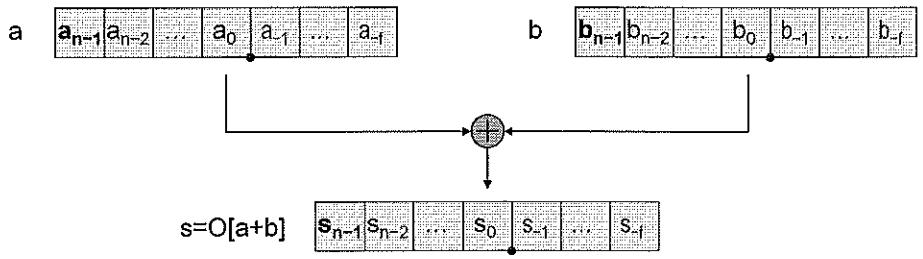
$-a-b = -(a+b) \rightarrow 2C(a+b)$

Complementing a twos complement number requires the negating of each bit (EXOR) and adding an LSB. When only one of the numbers (a or b) must be complemented, this addition of an LSB can be implemented as setting the input carry of the final addition to '1'.This results in hardware savings.

2.4 Overflow (2's C)

handling values that lie out of the dynamic range of an (n+f)-bit fixed-point number

→ *discards 'upper' bits*



e = O[a+b] – (a+b) = overflow error

Overflow occurs when the number of bits required to represent a number exceed the number of bits available.So, an overflow occurs when the value of a signal exceeds the dynamic range available.T he dynamic range is the range of numbers which can be represented within the arithmetic used. In twos complement fixed-point arithmetic, this range is given by:

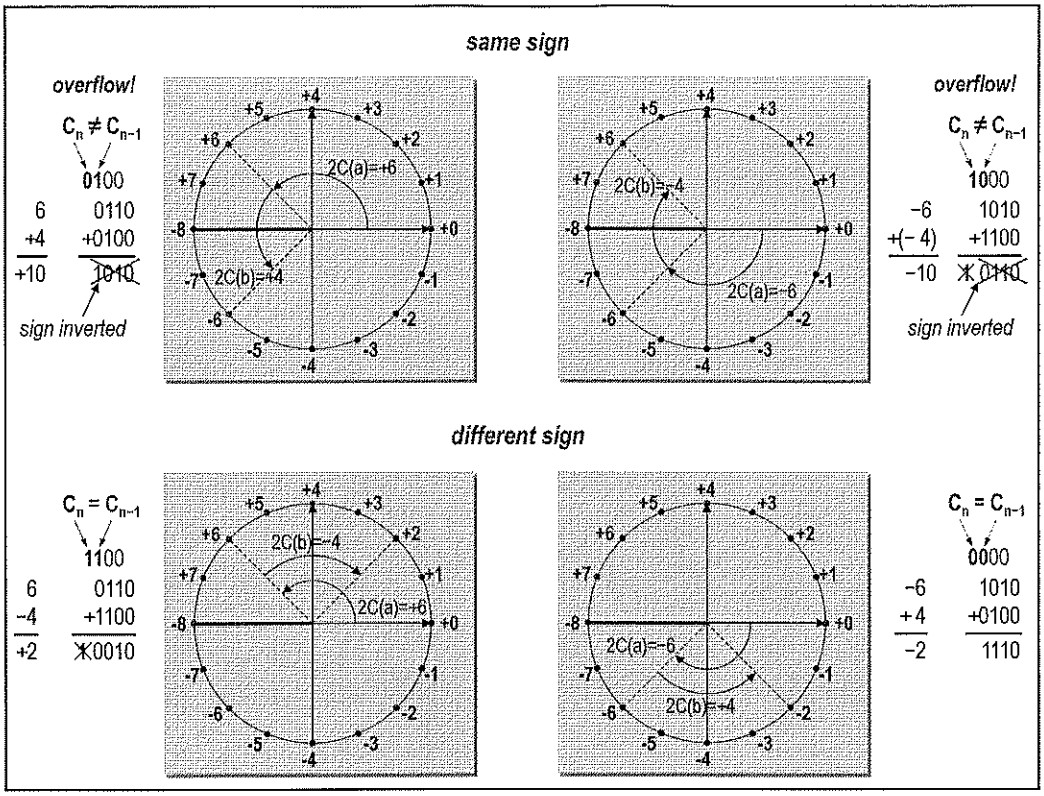
$$[\min, \max] = [\, -2^{n-1}, 2^{n-1} - 2^{n-f}\,]$$

Obviously, overflow errors are bad because they introduce big errors.

Overflow occurs whenever the sum of two positive numbers yields a negative result or when two negative numbers are summed and the result is positive.

Remark:

Scaling is the process of readjusting some internal gain parameters to avoid overflows. For a fixed total number of bits, there is a trade-off between decreasing the probability of overflow and increasing the relative quantisation error. Therefore, scaling is usually applied only to minimize the probability of overflow to a reasonable extent and not to preclude it entirely. When an overflow occurs, the resulting distortion is minimized by using overflow arithmetic.



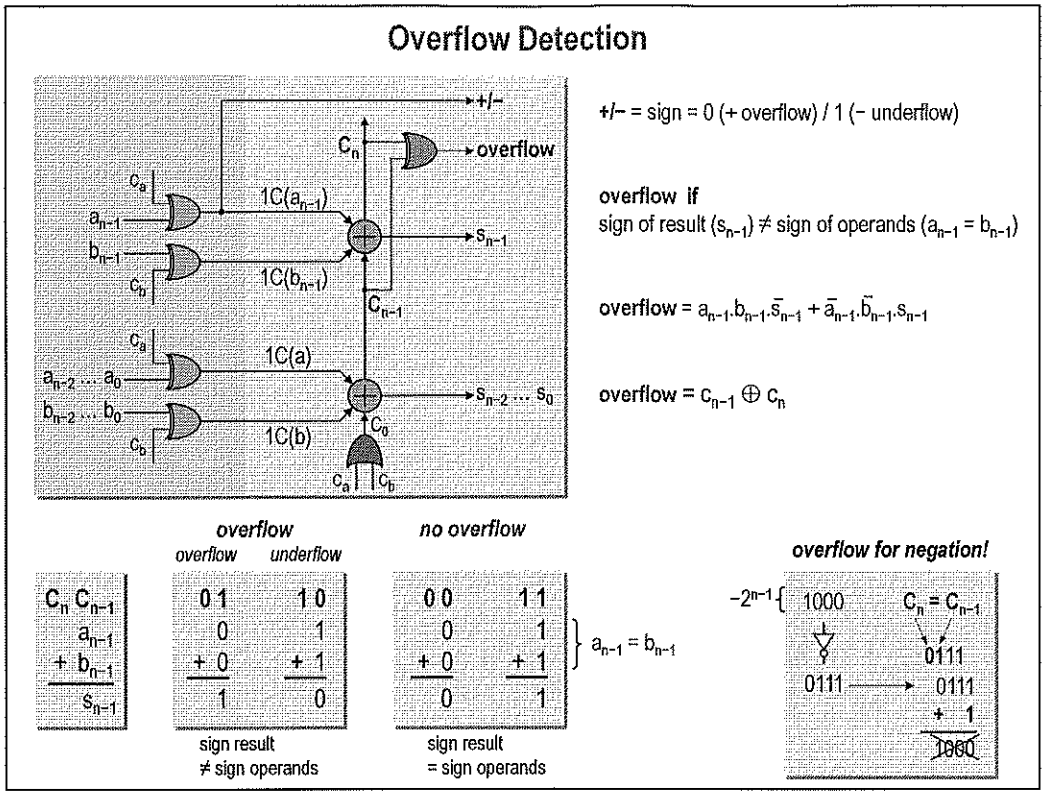
The number wheel can be used to illustrate overflow.T hink of addition as moving clockwise around the number wheel. Subtraction moves counter clockwise. Using the twos complement number representation, the number wheel can be divided into two halves, one representing positive numbers (and zero), the other representing the negative numbers. Whenever addition or subtraction crosses the positive/negative line, an overflow has occurred.

This concept is illustrated in the figure, with the two example calculations 6 + 4 and −6 + (−4).

For the sum 6 + 4, start on the number wheel with the representation for +6, advance for numbers in the clockwise direction. This yields −6; an overflow has occurred.

Similarly for subtraction.For the sum −6 + (−4), start with the representation for −6,move for numbers in the counter clockwise direction, obtaining the representation for +6. Once again, an overflow has occurred.

In general, overflow occurs when the carry-in and carry-out of the sign bit are different.



An overflow can be detected in two ways.

First, an overflow has occurred whenever the signs of the operands are the same and the sign of the sum does not agree with the signs of the operands. In an n-bit adder, overflow can be defined as:

$$\text{overflow} = a_{n-1} \cdot b_{n-1} \cdot \bar{s}_{n-1} + \bar{a}_{n-1} \cdot \bar{b}_{n-1} \cdot s_{n-1}$$

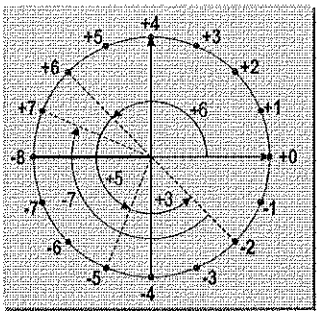
So, overflows occurs if the two operands are positive and the sum is negative, or if the two operands are negative and the sum is positive.

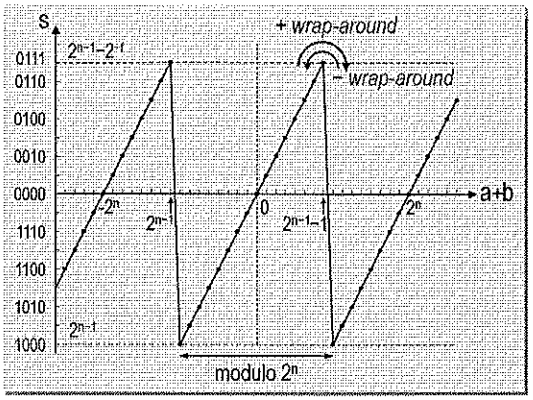
Secondly, if the carry-out of the high-order numeric (magnitude) position of the sum and the carry-out of the sign position of the sum agree, the sum is satisfactory; if they disagree, an overflow has occurred:

$$\text{overflow} = c_{n-1} \oplus c_n$$

A parallel adder adds the two operands, including the sign bits. An overflow from the magnitude part will tend to change the sign of the sum.

Intermediate Overflow (Wrap-Around)





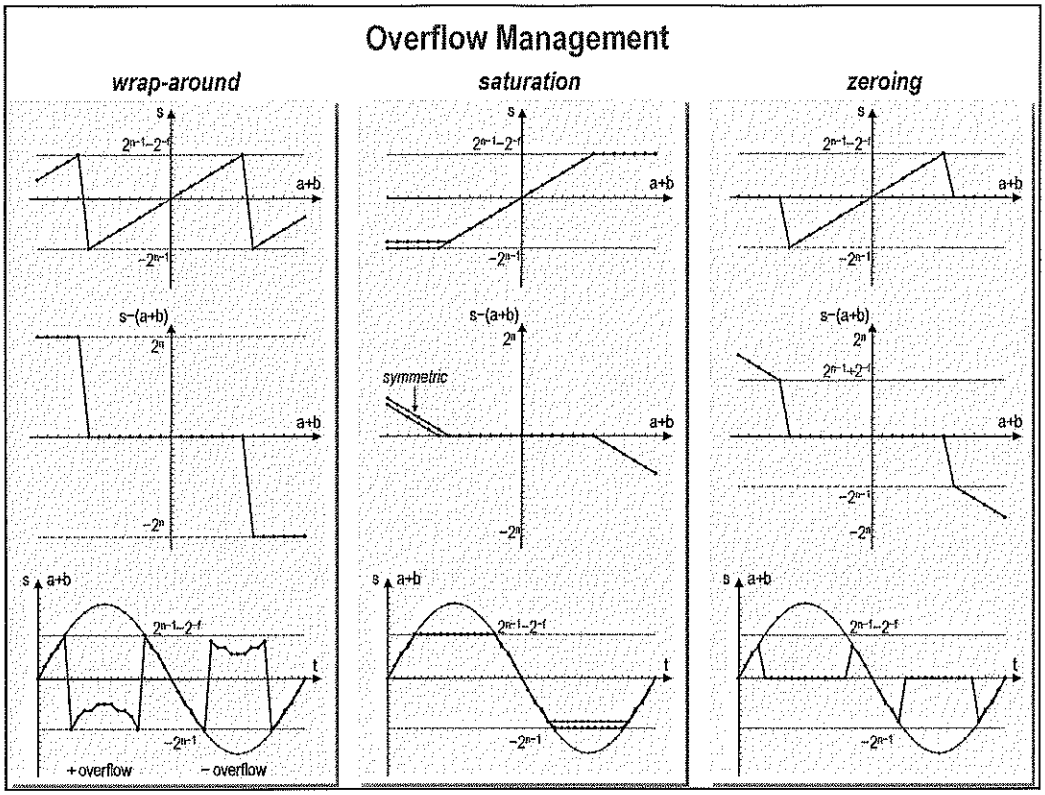
$C_n = C_{n-1}$	
	1111
6	0110
+5	+0101
+3	+0011
−7	+1001
+7	✕0111

$C_n \neq C_{n-1}$		
	0100	
6	0110	
+5	+0101	
+11	✕1011	← *intermediate overflow*
+3	+0011	
+14	1110	
−7	+1001	
+7	0111	← *correct result*

Any chain of arithmetic operations
whose final result is in the *representable range*
can be calculated *correctly*
even if *intermediate overflow* occurred.
A positive wrap-around is cancelled by
a negative wrap-around.

Two's complement wrap-around introduces large errors, but it has the advantage that if the sum of several numbers is within the dynamic range [min, max], the final result will be correct, even if intermediate sums overflow!

However, wrap-around overflow leaves IIR systems susceptible to zero-input large-scale limit cycles.



In twos complement fixed-point arithmetic, the dynamic range is given by:

$$[\min, \max] = [-2^{n-1}, 2^{n-1} - 2^{n-f}]$$

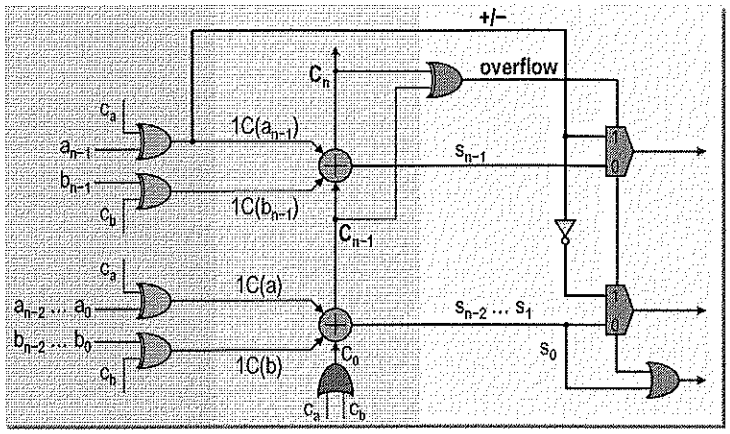
Saturation (or clipping)

When the result of an arithmetic operation exceeds the range of the destination type, saturation chooses the nearest representable value (as opposed to a nonsense wrap-around value obtained by dropping upper bits).

The overflow result is substituted by the values min or max, according to its sign.

Two's complement wrap-around introduce more error than saturation, but is easier in terms of hardware.

Overflow Management Implementation



$C_n \neq C_{n-1}$		$C_n \neq C_{n-1}$		$C_n \neq C_{n-1}$			
0100		1000		1000			
6	0110	−6	1010	−6	1010		
+4	+0100	+(−4)	+1100	+(−4)	+1100		
+10	1010	−10	✕0110	−10	✕0110		
+7	0111	−8	1000	−7	1001	*symmetric*	

An overflow has occurred if the carry-out of the high-order numeric (magnitude) position of the sum and the carry-out of the sign position of the sum disagree :

$$\text{overflow} = c_{n-1} \oplus c_n$$

The resulting sum is not satisfactory.

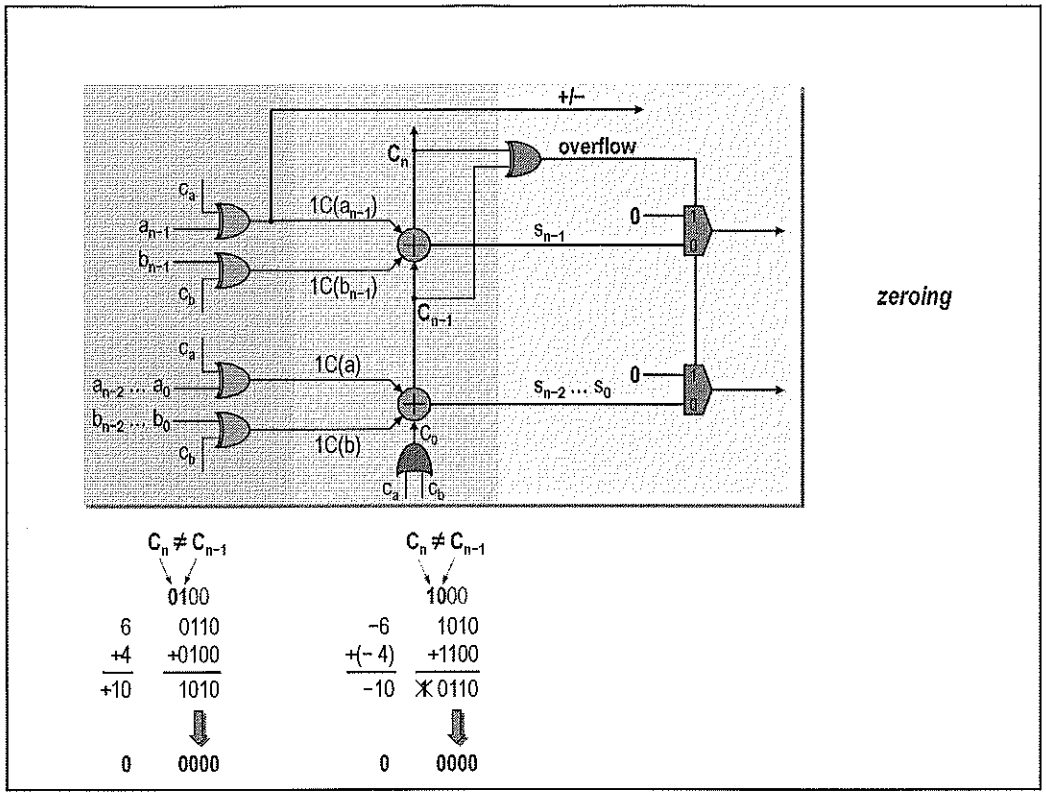
The saturating logic is activated when overflow is detected:

positive overflow: the sum is replaced by 011...11

negative overflow: the sum is replaced by 100...00 (asymmetric saturation)

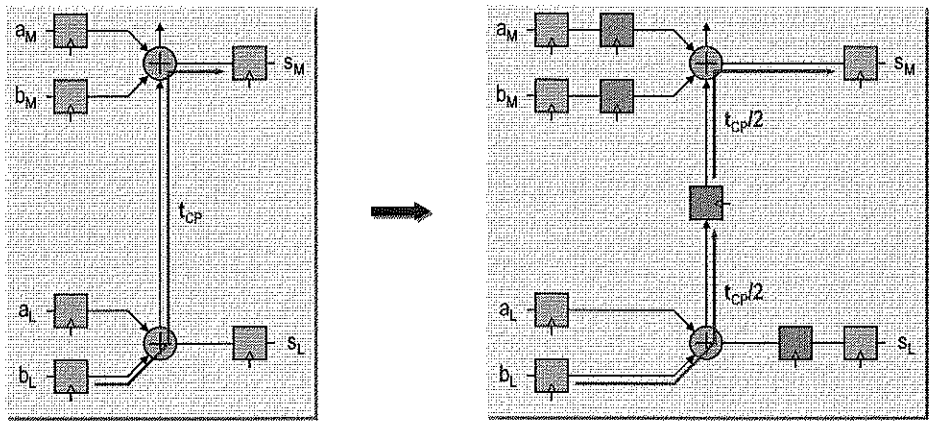
100...01 (symmetric saturation)

Remark: For the case that the result is 100...00, no overflow occurs. For symmetric saturation, a special detection circuit is needed to replace the result by 100...01. This circuit is not shown.



The zeroing logic is activated when overflow is detected. The sum is replaced by 000...00.

2.5 Pipelined Adder

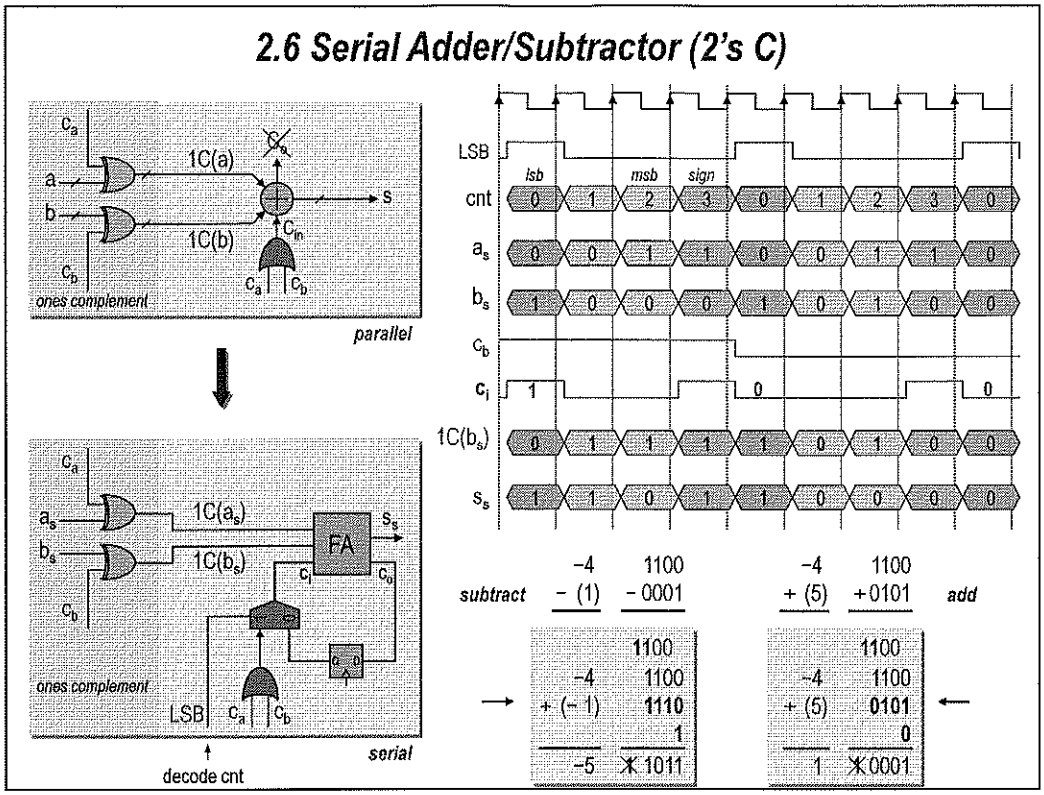


$a = [a_M\ a_L]$ M = most significant bits

$b = [b_M\ b_L]$ L = least significant bits

The critical path of an adder is the carry chain wit a delay t_{CP}.

This delay can be divided by two, by placing a pipeline register in the middle of the carry chain. Extra registers are inserted to align the most (M) and least (L) significant parts of the operands (a and b) and the sum (s).This is called retiming .



A bit-serial adder/subtractor can reduce the needed hardware but with a more complex control structure.

Adder

The two serial inputs a_s and b_s are assumed to be n-bit twos complement numbers and are synchronized to a clock signal, with the least significant bits (LSB's) first. The full adder is a combinational logic circuit which adds the two present input bits and a carry bit to produce a sum bit and an output carry bit. The output carry is stored in a one-bit delay or flip-flop (D) to be input for the full adder during the next clock cycle. After the most significant bits and sign bits are added to complete the addition, a new addition may begin immediately in the next clock cycle. Since a nonzero carry bit may still be stored in D from the previous addition, a clear (LSB) timing signal, which is zero during each LSB clock cycle, forces the input carry to zero via the multiplexer.

Assuming that the data samples are (n+1)-bit twos-complement numbers, an addition requires n + 1 clock cycles. This is one clock cycle longer than the serial inputs (operands). Several methods can be used to make operands and result equal in length: sign extension of the operands, rounding of the result, limiting the range of the operands so that the result can be represented with n bits (as was done in the figure).

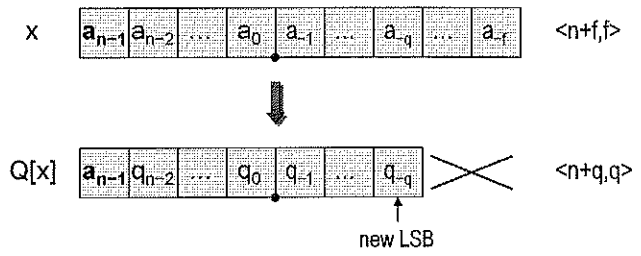
Subtractor

Subtraction is the negation of the subtrahend (b_s in the example shown), followed by addition. Negation of a twos-complement number is implemented by complementing all the bits of the number and adding '1' in the LSB position. Hence in the subtractor, an inverter complements all bits of the subtrahend, and the carry loop is modified to cause an initial carry input of '1' during the LSB cycle.

3. Quantization

reduction of the precision of a binary fixed-point number from n+f bits to n+q bits

⟶ discards 'lower' bits



e = Q[x] – x = quantization error

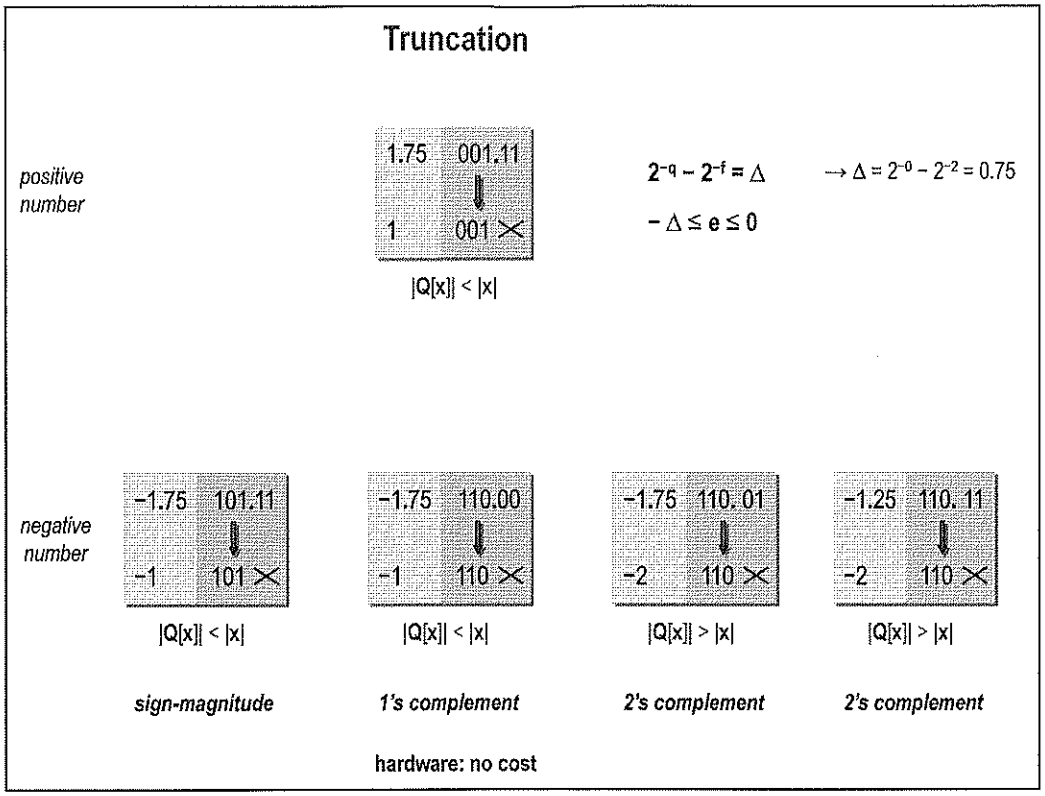
In performing fixed-point operations such as multiplications, it is often necessary to quantize a binary fixed-point number x, to another number Q[x], reducing the precision (number of fractional bits) from f to q, as shown. This can be done via truncation or rounding.

The effect of this reduction of precision is to introduce an error whose power depends on the values of f and q and whose statistical behaviour depends on the type of truncation or rounding used. This error is referred to by the generic name quantisation error and is given by:

$$e = Q[x] - x$$

Overflow handling versus Quantization

- Overflow handling discards upper bits.
- Quantization discards lower bits.
- Overflow handling is concerned with values that lie out of the range of the data type.
- Quantization is concerned with values that are more precise than the data type (i.e., between values in the data type).



In truncation, the least significant bits $a_{-q-1} \ldots a_{-f}$ are simply dropped, regardless of the sign and the convention to represent the negative numbers.

If x is a *positive number*, the 3 binary representations are identical.
If all the truncated bits are 1, the maximum quantisation error occurs, with absolute value Δ :

$$\Delta = 2^{-f} - 2^{-q}$$

The resulting number Q[x] is always smaller than the original number x. The corresponding quantisation error is always negative and limited:

$$-\Delta \leq e \leq 0 \quad \text{and} \quad Q[x]] \leq |x|$$

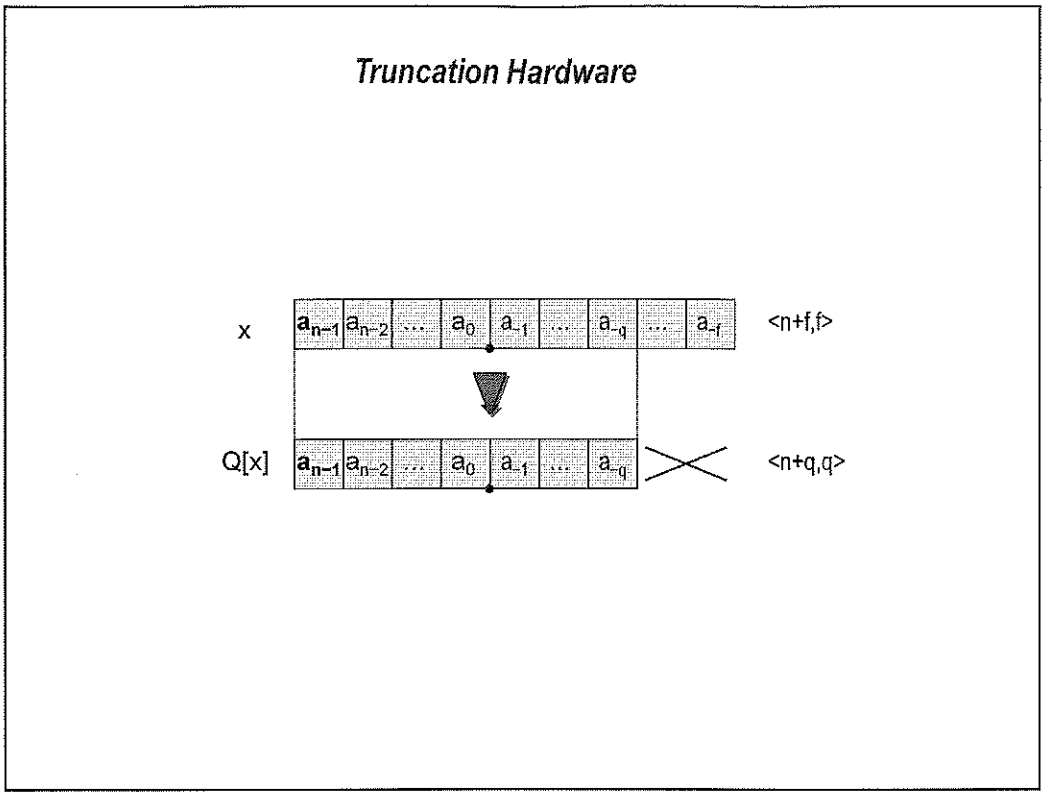
If x is a *negative number*, the error depends on which binary representation is used.

- sign-magnitude truncation and ones complement truncation:

$$0 \leq e \leq \Delta \quad \text{and} \quad |Q[x]| \leq |x|$$

- twos complement truncation:

$$-\Delta \leq e \leq 0 \quad \text{and} \quad |Q[x]| \geq |x|$$

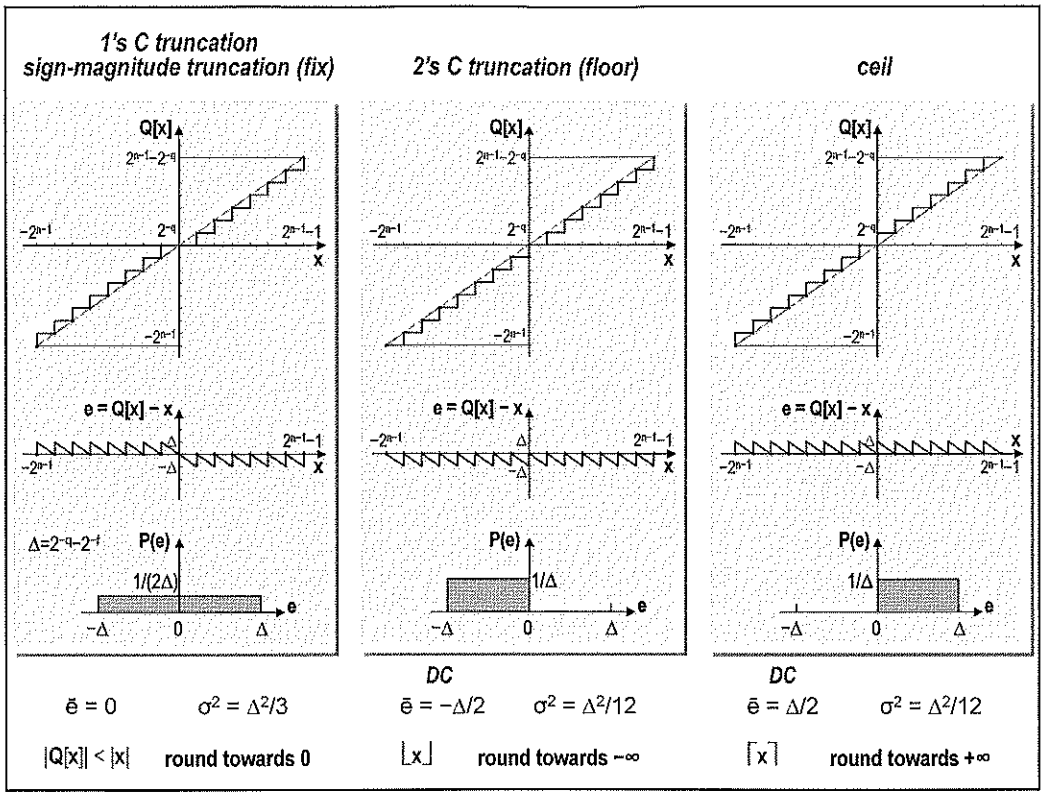


Truncation

- Simply discards the low (f-q) bits
- Quite imprecise

Example:

0000|1111 becomes 0 by truncation of the lower 4 bits.



The quantisation curve, the quantization error and the statistical behaviour of the quantisation error (error probability density function) for truncation and ceil are shown.

Round towards zero (1's C truncation, sign-magnitude truncation = fix)

The sign of the quantisation error depends on the sign of the number to be quantised. The mean value for random numbers is zero.

Notice that with round towards zero, the probability density function is twice as wide as the others. For this reason, the variance is four times that of the others.

Round towards minus infinity (2's C truncation = floor)

Floor is often called truncation when used with integers and fixed-point numbers that are represented in two's complement. It is the most common quantisation mode of DSP processors because it requires no hardware to implement.

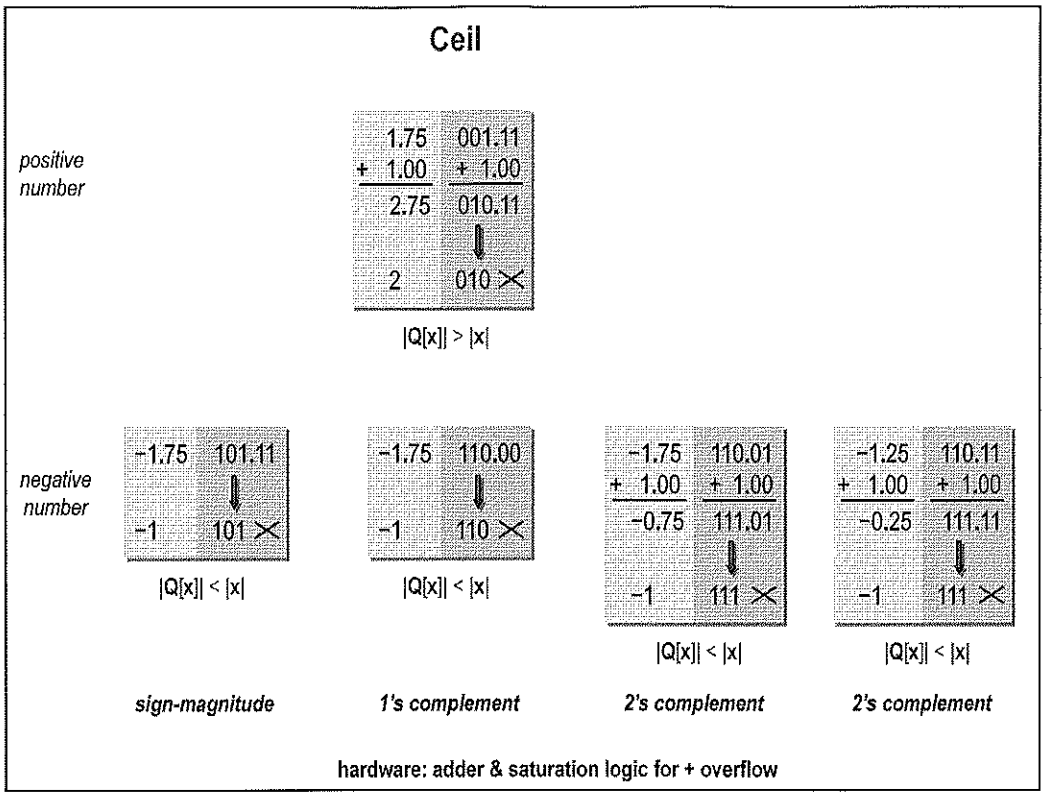
The sign of the quantisation error is always negative. So for random numbers the mean value is also negative. This results in a DC shift in digital filters implemented with 2's C truncation!

Floor does not produce quantized values that are as close to the true values as round will, but it has the same variance, and small signals that vary in sign will be detected, whereas in round they will be lost.

Round towards plus infinity (ceil)

The sign of the quantisation error is always positive. So for random numbers the mean value is also positive. This results in a DC shift in digital filters!

Ceil does not produce quantized values that are as close to the true values as round will, but it has the same variance, and small signals that vary in sign will be detected, whereas in round they will be lost.



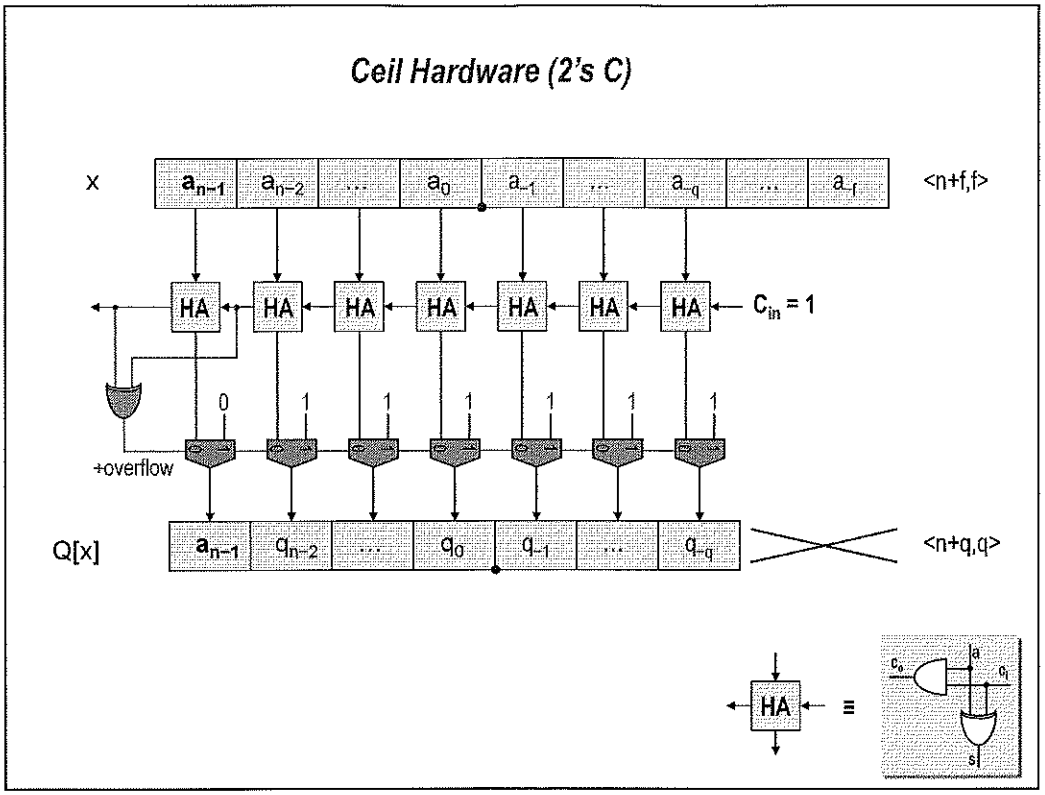
Round towards minus infinity (2's C truncation = floor)

2's C truncation requires no hardware to implement. The least significant bits are simply ignored.

Round towards plus infinity (ceil)

2's C ceiling requires extra hardware, without reducing the quantisation error.

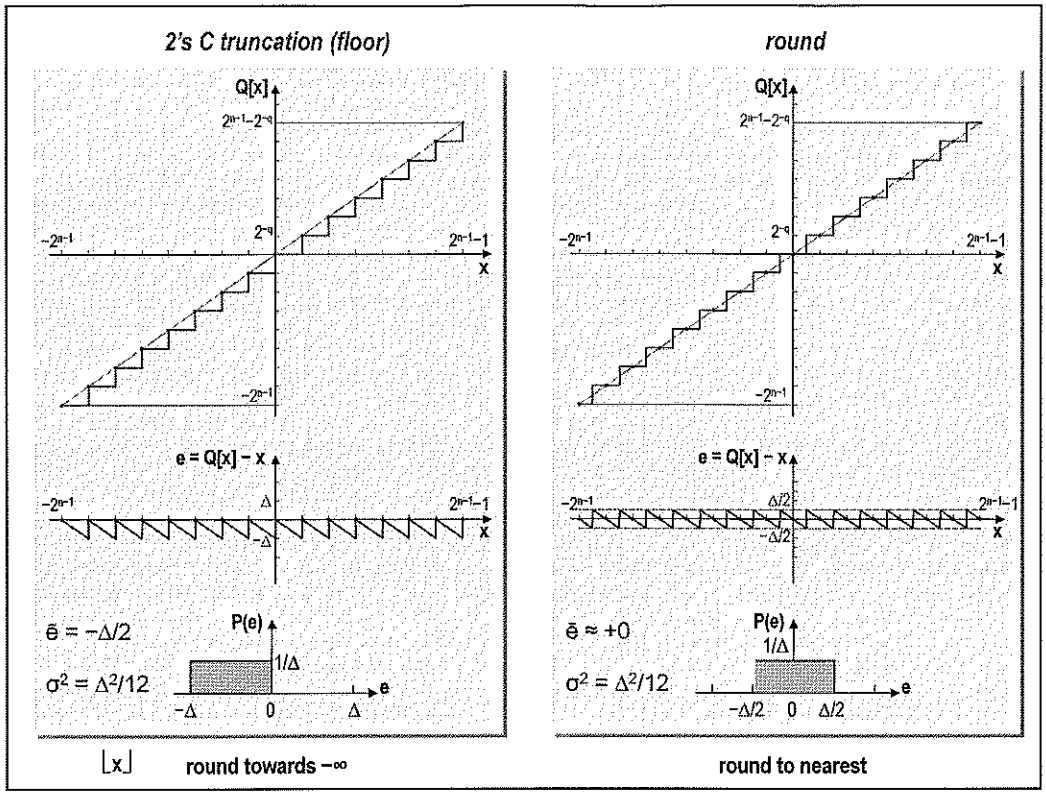
First, a '1' has to be added at the LSB position of the new (quantized) number before the least significant bits can be ignored. Due to this addition, overflow is possible.



Round towards plus infinity (ceil)

2's C ceiling requires extra hardware.

- First, a '1' is added at the LSB position (bit a_{-q}) of the new (quantized) number.
- Then, the least significant bits can be ignored.
- When positive overflow occurs due to the addition, saturation logic can be activated.



The quantisation curve, the quantization error and the statistical behaviour of the quantisation error (error probability density function) for truncation and rounding are shown.

Floor = Round towards minus infinity

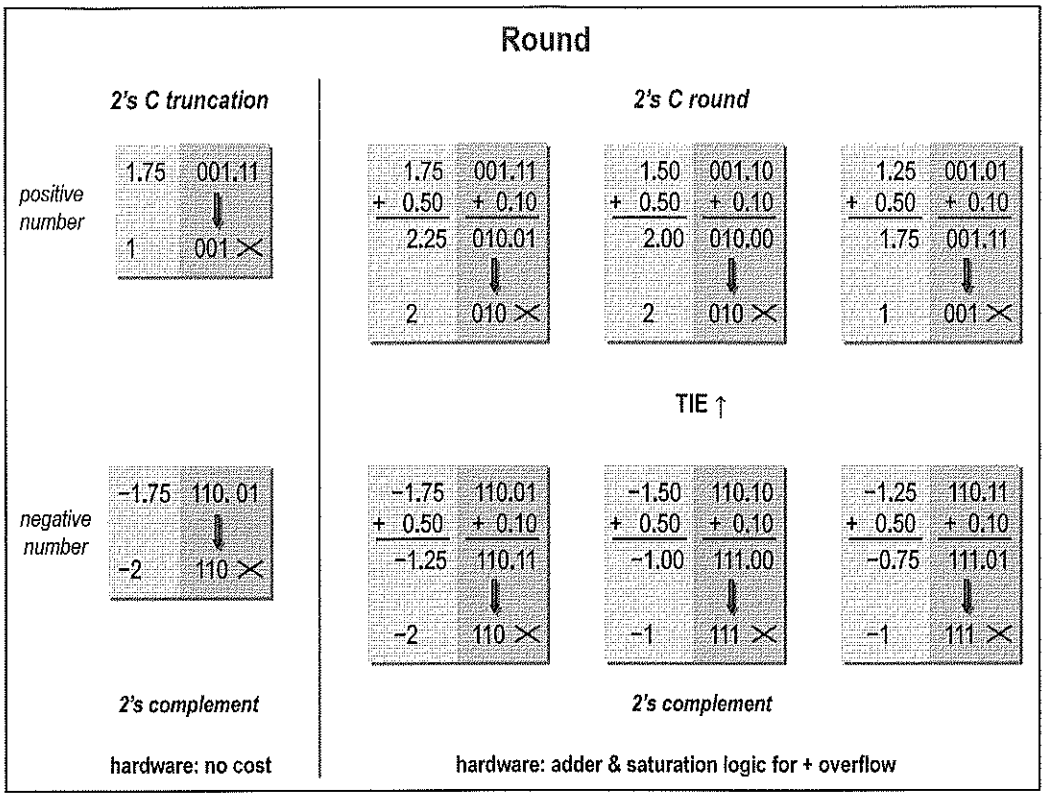
The sign of the truncation error is always negative. So for random numbers the mean value is also negative. This results in a DC shift in digital filters implemented with 2's C truncation!

The worst case truncation error is twice as large as the worst case rounding error. Also, the truncation error is always negative, so on average it has a non-zero mean.

Round = Round-to-nearest. In a tie, round to plus infinity

The sign of the quantisation error is alternating, depending on the precise magnitude of the number. A small bias is introduced by ordinary "round" caused by always rounding the tie in the same direction. So for random numbers the mean value is nearly zero (slightly positive).

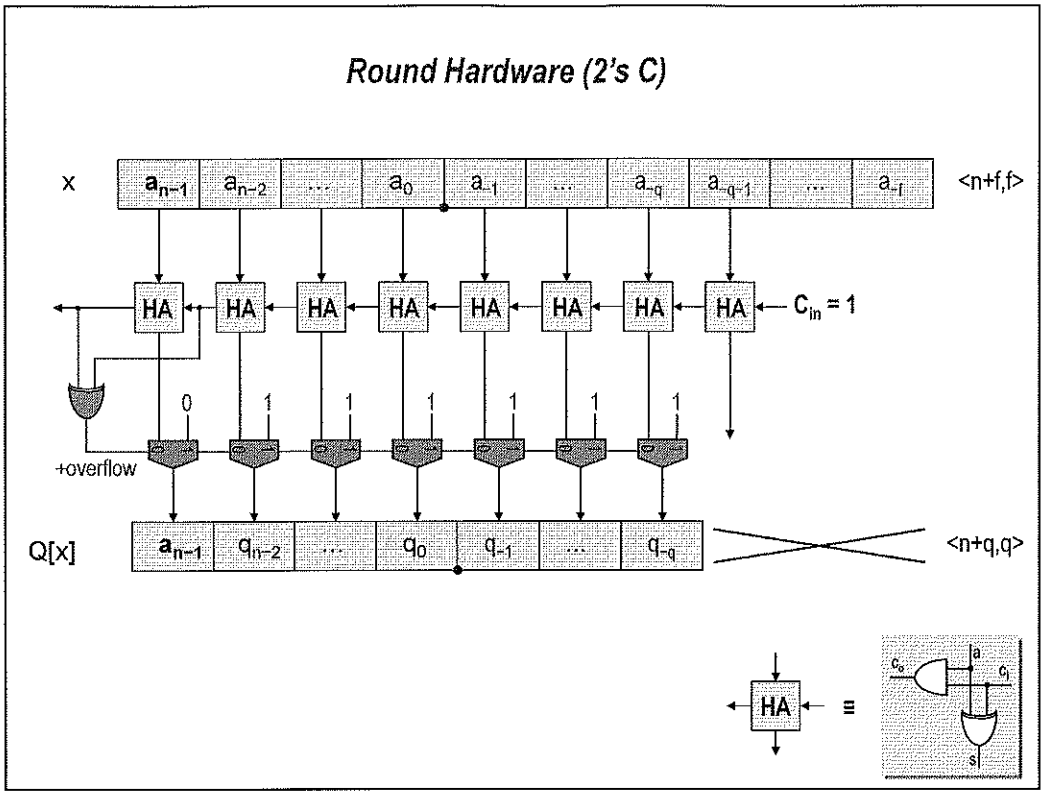
Round is more accurate than floor, but all values smaller than 2^{-q} get rounded to zero and so are lost.



In rounding, the value Q[x] is taken to be the nearest possible number to x. The quantisation error is limited by:

$$-(2^{-f} - 2^{-q})/2 < e \leq -(2^{-f} - 2^{-q})/2$$

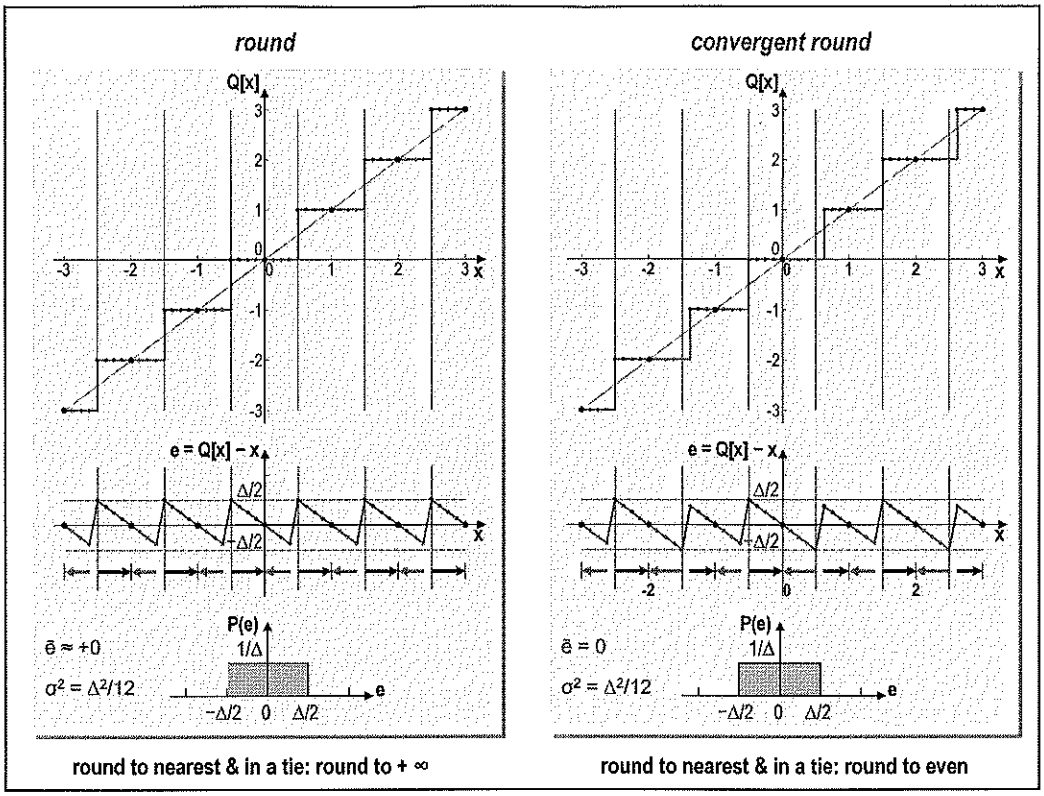
Rounding is more accurate than truncation, but requires more effort in its implementation.



Round = Round-to-nearest. In a tie, round to plus infinity

2's C rounding requires extra hardware.

- First, a '1' is added at the LSB−1 position (bit a_{-q-1}) of the new (quantized) number.
- Then, the least significant bits can be ignored.
- When positive overflow occurs due to the addition, saturation logic can be activated.



Convergent rounding eliminates the bias introduced by ordinary "round" caused by always rounding the tie in the same direction.

Round = Round-to-nearest. In a tie, round to plus infinity

- Chooses the nearest representable value.
- In case of a tie, rounds up.
- Sometimes called *biased* rounding.
- The error probability density function:

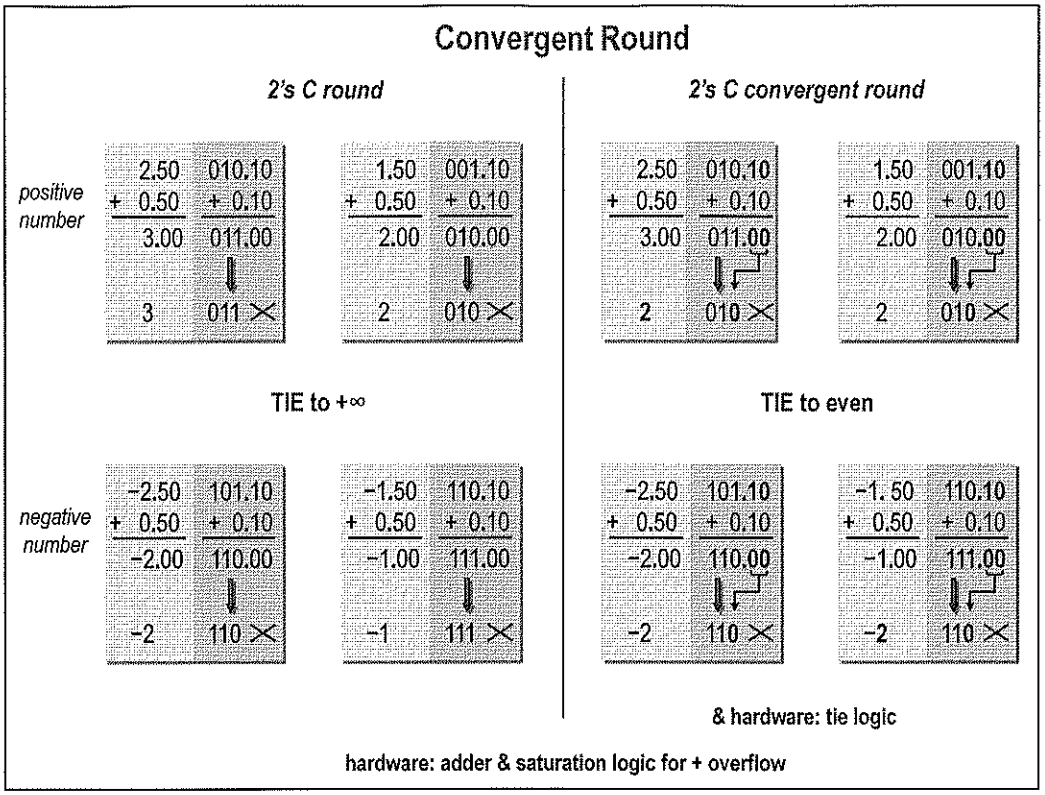
$$P(e) = 1/\Delta \quad \text{for} \quad -\Delta/2 < e \leq \Delta/2$$
$$= 0 \quad \text{otherwise}$$

Convergent rounding = Round-to-nearest. In a tie, round to even

- Also chooses nearest representable value.
- In case of tie, chooses nearest *even* number.
- Sometimes called *unbiased* rounding.
- The error probability density function:

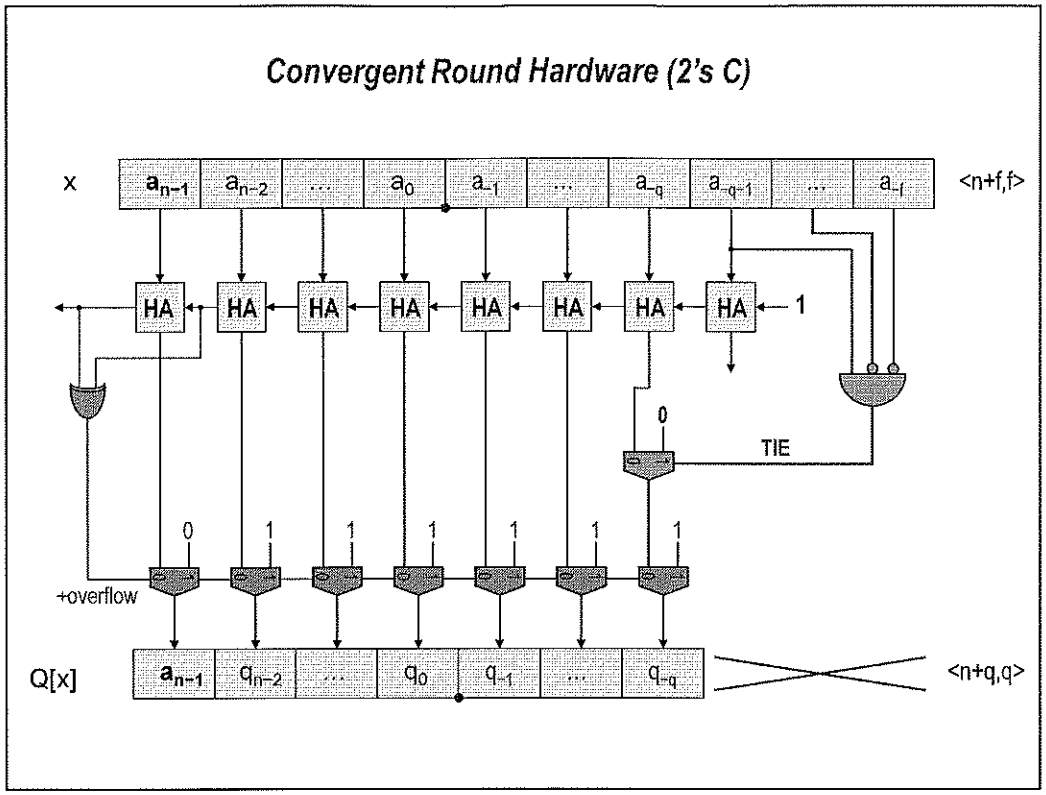
$$P(e) = 1/\Delta \quad \text{for} \quad -\Delta/2 \leq e \leq \Delta/2$$
$$= 0 \quad \text{otherwise}$$

The error probability density function for convergent rounding is difficult to distinguish from that of round-to-nearest by looking at the plot. Note that the error p.d.f. of convergent is symmetric, while round is slightly biased towards the positive.The only difference is the direction of rounding in a tie.



The conventional rounding rounds up any value above one-half the quantisation interval and rounds down any value below one-half. Because one-half is always rounded up, the result Q[x] will be biased in upward direction.

Convergent rounding solves the problem by rounding down if the number is odd and rounding up if the number is even.



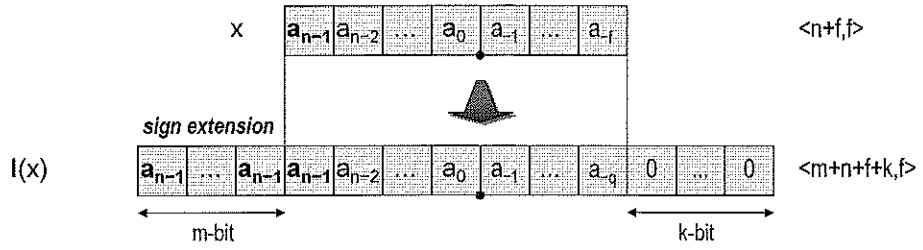
Round = Round-to-nearest. In a tie, round to plus infinity

2's C rounding requires extra hardware.

- First, a '1' is added at the LSB−1 position (bit a_{-q-1}) of the new (quantized) number.
- When a tie is detected (all bits that will be ignored are zero except the bit a_{-q-1}), the LSB position of the new (quantized) number is set to 0.
- Then, the least significant bits can be ignored.
- When positive overflow occurs due to the addition, saturation logic can be activated.

4. Increasing Size (Casting)

increasing the number of bits in a 2's complement fixed-point number without changing its value



I(x) = x

(no error)

casting = changing the number of bits in a twos complement representation.

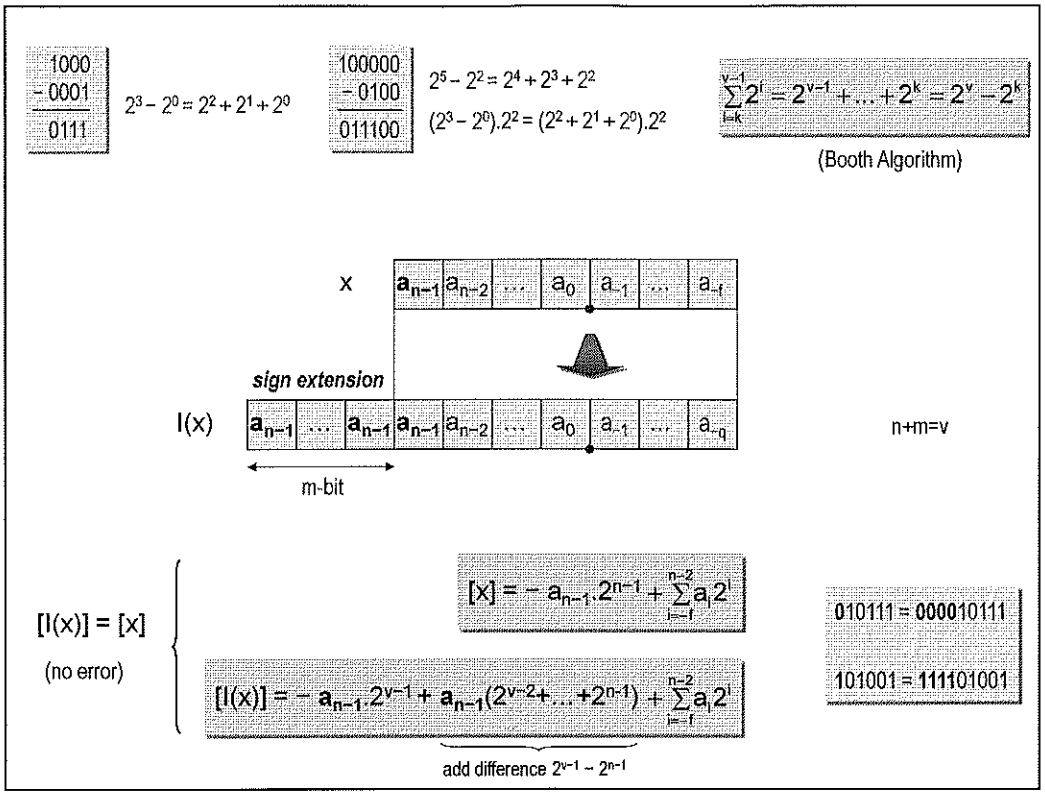
The number of bits in a twos complement representation can be increased, without changing its value.

- *To the right of the LSB*

  Append positions to the right of the LSB and fill them with 0's.

- *To the left of the MSB* = sign extension

  Replicates the sign bit in an uninterrupted sequence.



When a twos complement number with a n-bit integer part is extended to a number with a v-bit integer part, the complementation constant (= weight of the sign bit when the value is calculated) increases from 2^{n-1} to 2^{v-1}.

The difference of the two constants:

$$2^{v-1} - 2^{n-1}$$

must be added to the representation of any negative number. Using the Booth algorithm:

$$2^{v-1} - 2^{n-1} = 2^{v-2} + 2^{v-3} + \ldots + 2^{n-1}$$

this is equal to (v−n−1) 1's followed by (n−1) 0's.

Example:

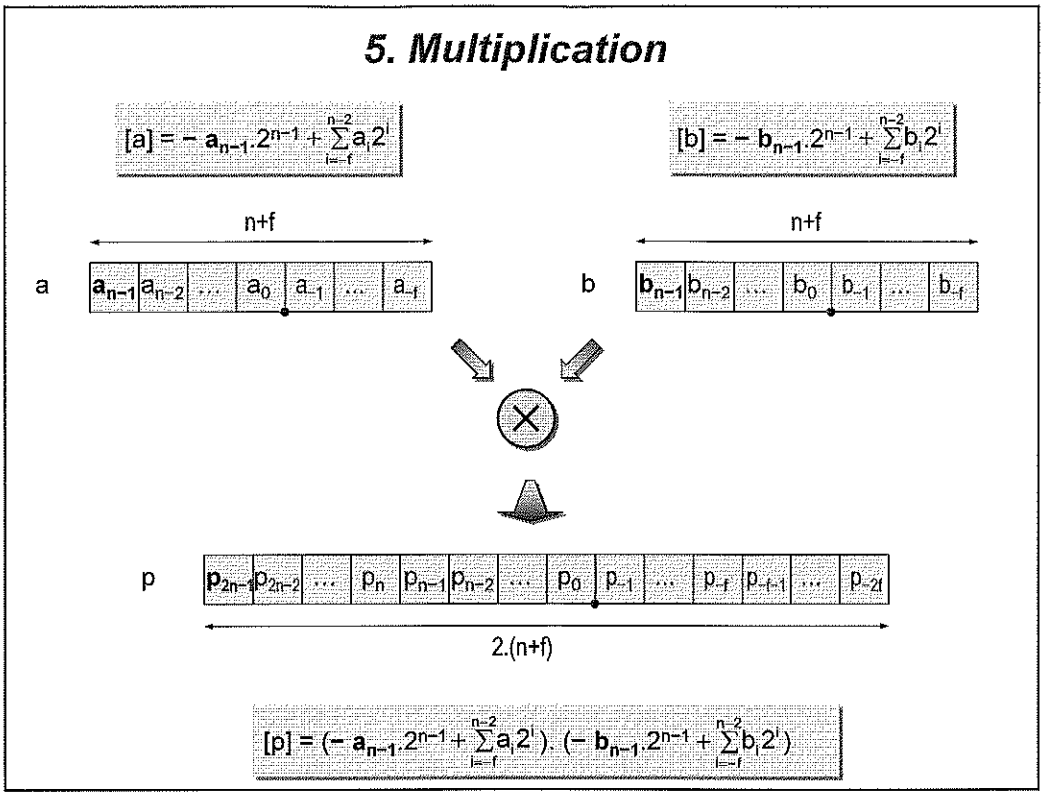
876543210		
100000000	2^8	256
- 000010000	- 2^4	- 16
011110000	$2^7 + 2^6 + 2^5 + 2^4$	240 = 128 + 64 + 32 + 16

To extend the integer part of a twos complement number with p=v-n bits, the sign bit must be extended (replicated) p times.

Example (p=3):

010111 = **000**10111

101001 = **111**101001



Multiplications are expensive and slow operations. The performance of many computational problems often is dominated by the speed at which a multiplication operation can be executed. This observation has, for instance, prompted the integration of complete multiplication units in state-of-the-art digital signal processors and microprocessors. Multipliers are, in effect, complex adder arrays. The analysis of the multiplier gives some further insight into how to optimize the performance (or the area) of complex circuit topologies.

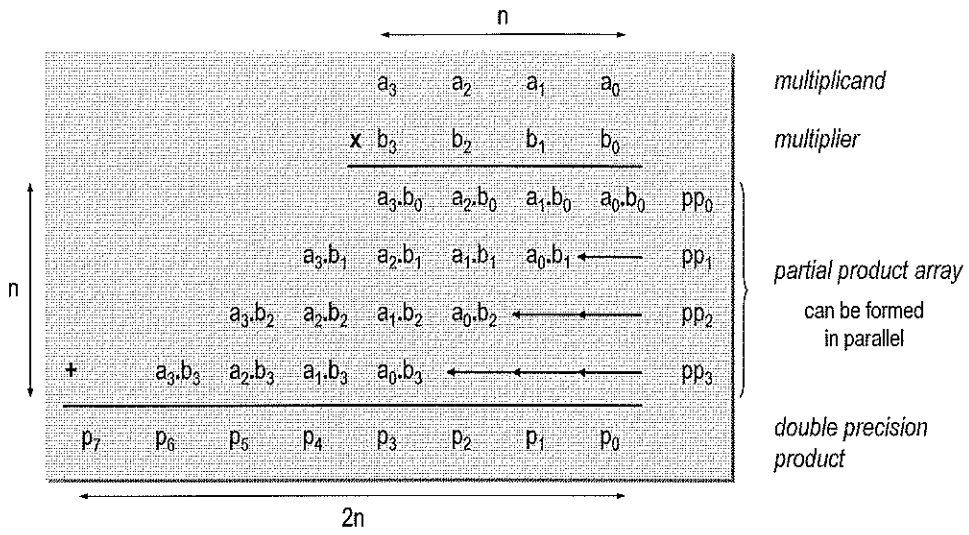
Consider two signed fixed-point binary numbers a and b that are n+f bits wide. To introduce the multiplication operation, it is useful to express a and b in the binary representation:

$$[a] = -a_{n-1}.2^{n-1} + \sum_{i=-f}^{n-2} a_i.2^i \qquad [b] = -b_{n-1}.2^{n-1} + \sum_{i=-f}^{n-2} b_i.2^i$$

with a_i, bj $\in \{0,1\}$. The multiplication operation is then defined as follows:

$$[p] = (-a_{n-1}.2^{n-1} + \sum_{i=-f}^{n-2} a_i.2^i).(-b_{n-1}.2^{n-1} + \sum_{i=-f}^{n-2} b_i.2^i)$$

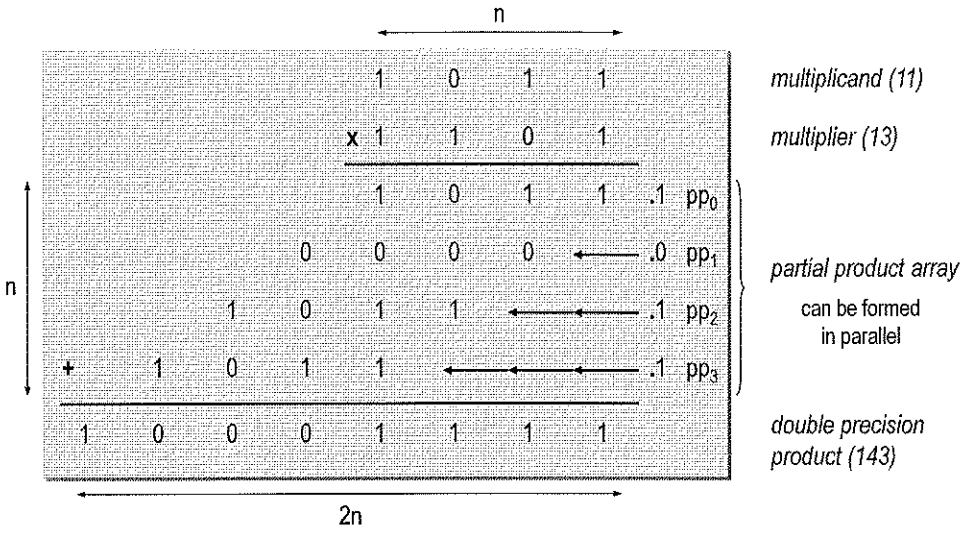
5.1 Unsigned Multiplication



Multiplication can be defined as repeated addition. The number to be added is the multiplicand, the number of times that its shifted version is added is the multiplier, and the result is the product. Each step of the addition generates a partial product. In most computers, the operands usually contain the same number of bits. When the operands are interpreted as integers, as in fixed-point arithmetic, the product is usually twice the length of the operands in order to preserve the information content.

The simplest way to perform a multiplication is to use a single two-input adder. For inputs that are n bits wide, the multiplication takes n cycles, using an n-bit adder. This shift-and-add algorithm for multiplication adds together n partial products. Each partial product is generated by multiplying the multiplicand with a bit of the multiplier - which, essentially, is an AND operation - and by shifting the result on the basis of the multiplier bit's position. A faster way to implement multiplication is to resort to an approach similar to manually computing a multiplication. All the partial products are generated at the same time and organized in an array. A multi-operand addition is applied to compute the final product. This set of operations can be mapped directly into hardware. The resulting structure is called an array multiplier and combines the following three functions: partial-product generation (and), partial-product shift, partial-product accumulation, and final addition.

Example of Unsigned Multiplication



The multiplication of two unsigned n-bit numbers results in a double precision product (2n-bit):

n bits x n bits = 2n bit product

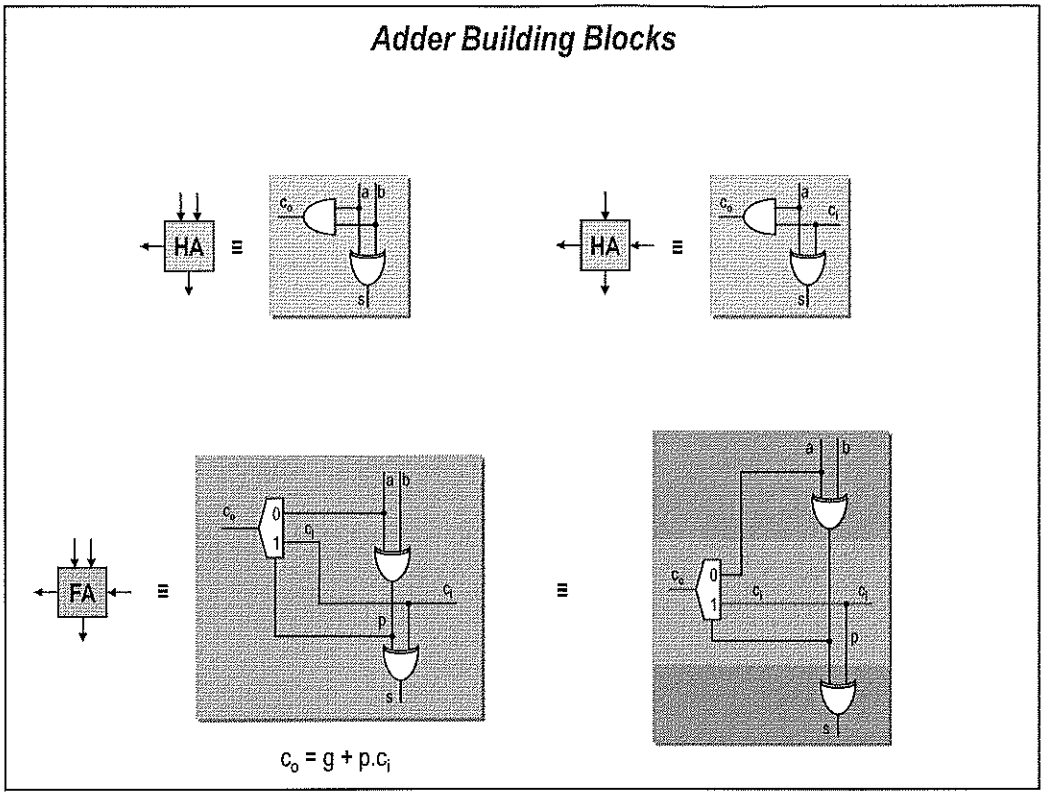
The binary representation of numbers makes the multiplication process easier:

0 → place 0 (0 x multiplicand)

1 → place a shifted copy (1 x multiplicand)

Partial products result from the logical AND of multiplicand a with a multiplier bit b_i. Each row in the partial-product array is either a copy of the multiplicand or a row of zeroes. Careful optimization of the partial-product generation can lead to some substantial delay and area reductions. Note that in most cases the partial-product array has many zero rows that have no impact on the result and thus represent a waste of effort when added. In the case of a multiplier consisting of all ones, all the partial products exist, while in the case of all zeros, there is none. Based on this observation the mean number of generated partial products is n/2.*Boo th recoding* is a technique that skips over 0's to reduce the number of additions in the multiplication process.

After the partial products are generated, they must be summed. This accumulation is essentially a *multi-operand addition*. A straightforward way to accumulate partial products is by using a number of adders that will form an array - hence, the name, array multiplier. A more sophisticated procedure performs the addition in a tree format (*Wallace*).



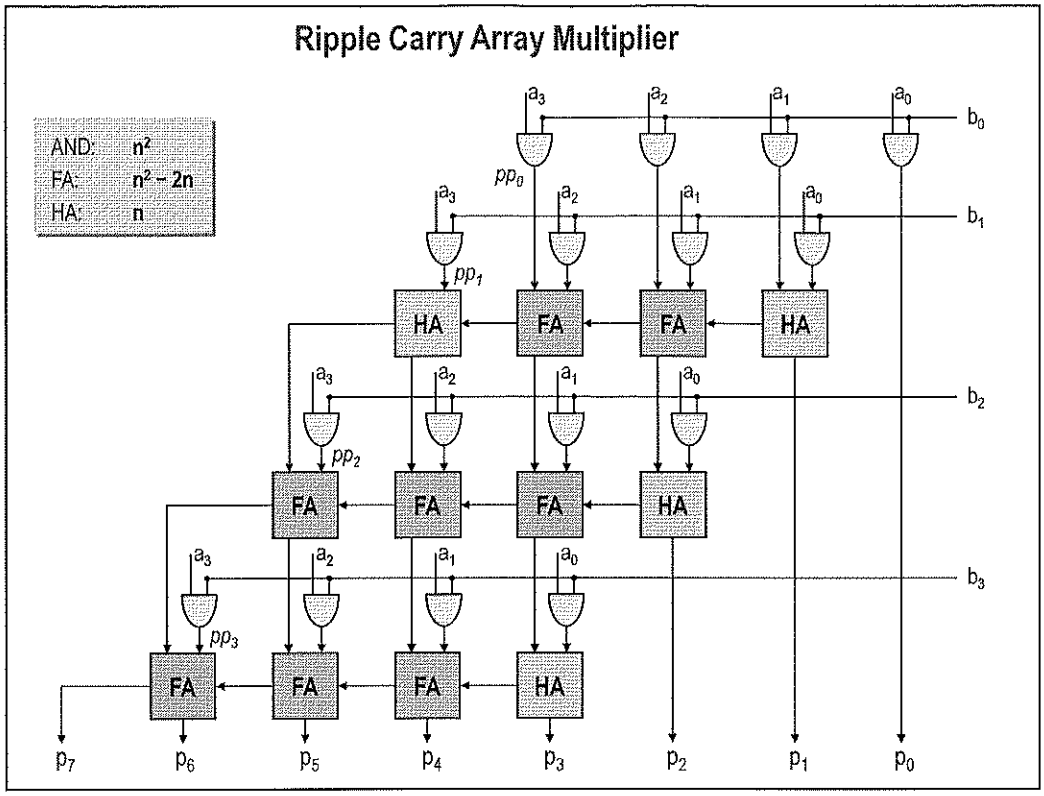
The basic building blocks for binary addition are the half adder (HA) and full adder (FA).

Half Adder (HA)

The half adder is the most primitive arithmetic circuit involved in the addition process. It has two inputs (the bits a and b to be added) and two outputs (the sum s and the output carry c_o).

Full Adder (FA)

In a multi-bit addition, the carry-out of a previous stage must be added to the sum of the present stage. This requires a third input: input carry c_i.

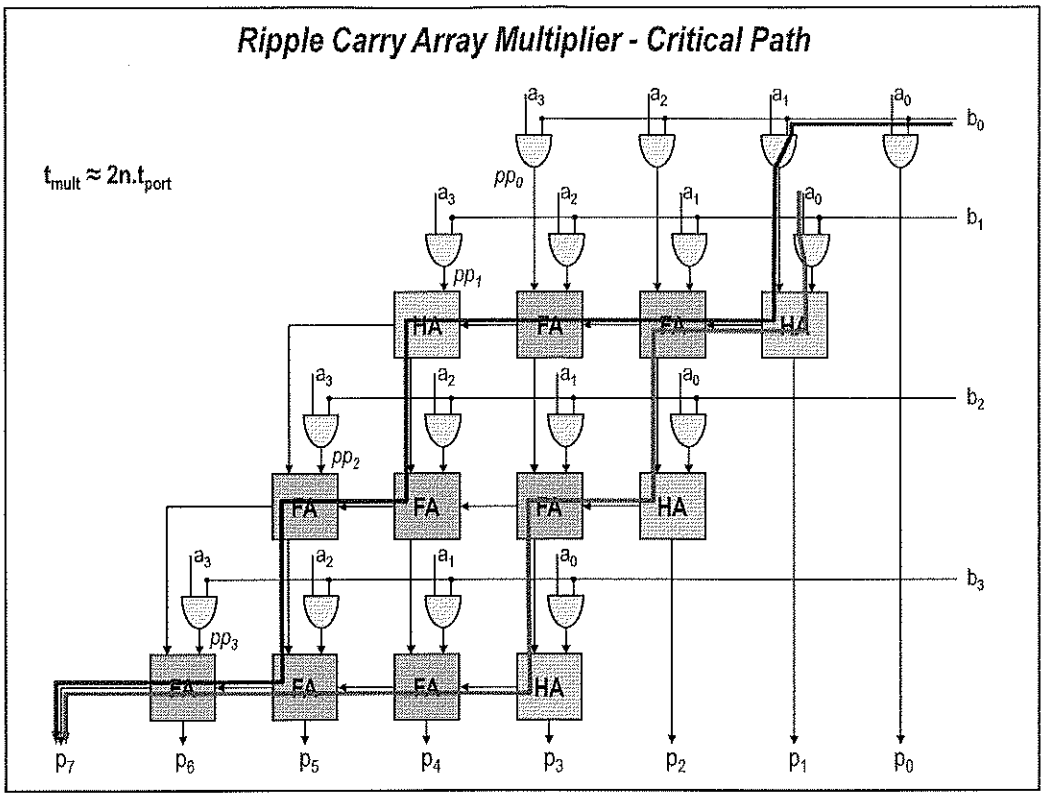


Ripple Carry Array Multiplier

A ripple carry array multiplier (also called row ripple form) is an unrolled embodiment of the classic shift-add multiplication algorithm. The illustration shows the adder structure used to combine all the partial products in a 4x4 multiplier. The partial products are the logical and of the bits of the multiplicand a witch each bit of the multiplier b.

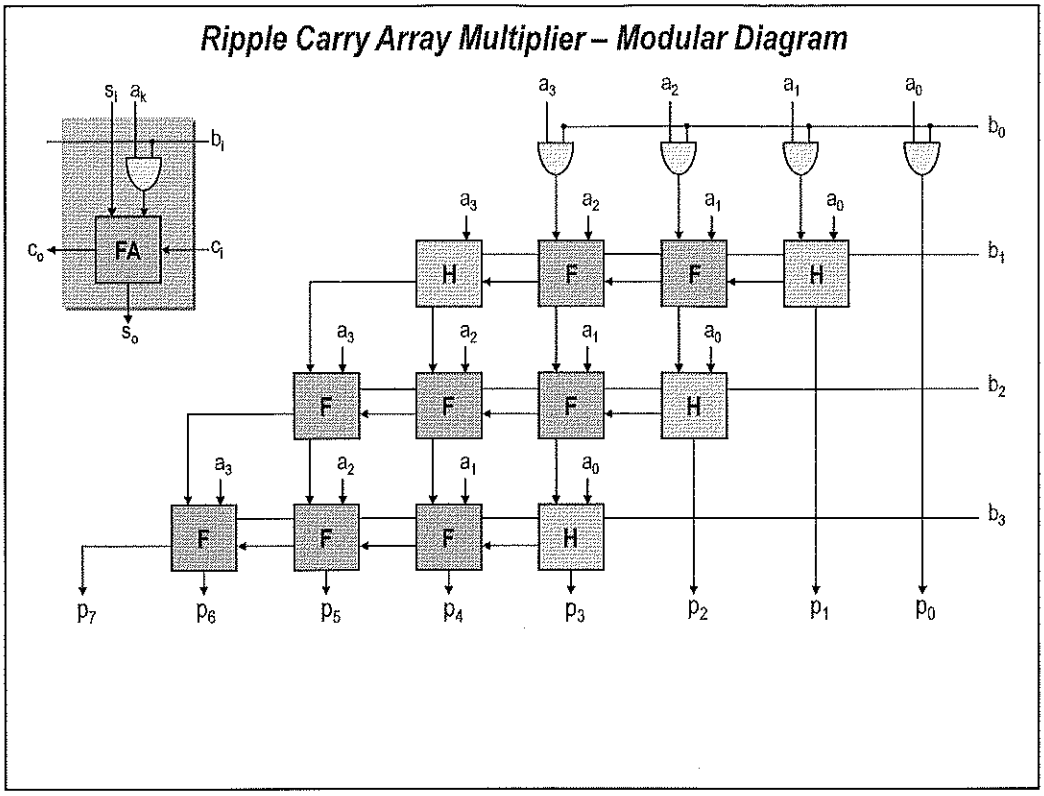
The hardware composition of an array multiplier is shown. There is a one-to-one topological correspondence between this hardware structure and the manual multiplication. The generation of n partial products requires n^2 two-bit AND gates. Most of the area of the multiplier is devoted to the adding of the n partial products, which requires (n−1) n−bit adders, of which n HA's and $(n-1)^2-1$ FA's. The shifting of the partial products for their proper alignment is performed by simple routing and does not require any logic.

This basic structure is simple to implement in FPGA's, but does not make efficient use of the logic in many FPGA's, and is therefore larger and slower than other implementations.



Due to the array organization, determining the propagation delay of this circuit is not straightforward. Consider the implementation shown. The partial sum adders are implemented as ripple-carry structures. Performance optimization requires that the *critical timing path* be identified first. This turns out to be nontrivial. In fact, a large number of paths of almost identical length can be identified. Two of those are highlighted in the figure. The maximum delay is the path from either LSB input (a_0 or b_0) to the MSB of the product (p_7), and is the same (ignoring routing delays) regardless of the path taken. The delay is approximately $2.n.t_{port}$.

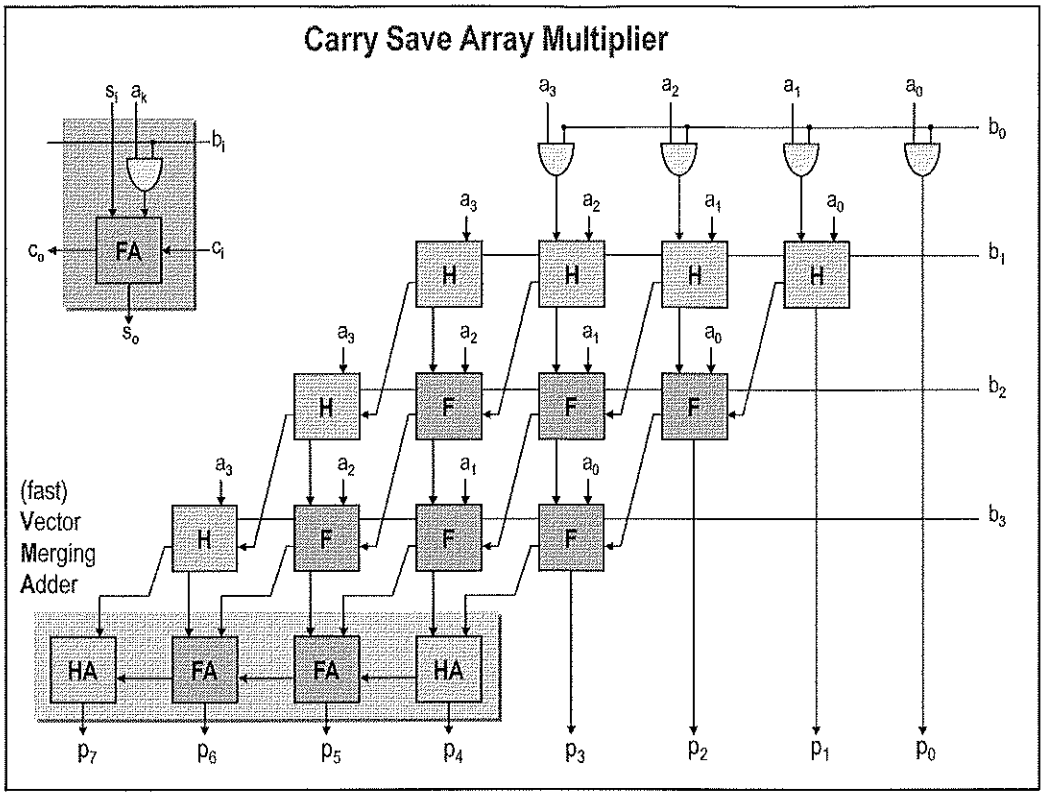
Since all critical paths have the same length, speeding up just one of them for instance, by replacing one adder by a faster one, does not make much sense from a design standpoint.All critical paths have to be attacked at the same time.



In the modular diagram of the ripple carry array multiplier, the partial product generation (and) and addition (HA or FA) are combined in a single block to simplify the structure of the multiplier.

Ripple Carry Array Multiplier (conclusion):

- row ripple form
- unrolled shift-add algorithm
- delay is proportional to n
- regular routing pattern

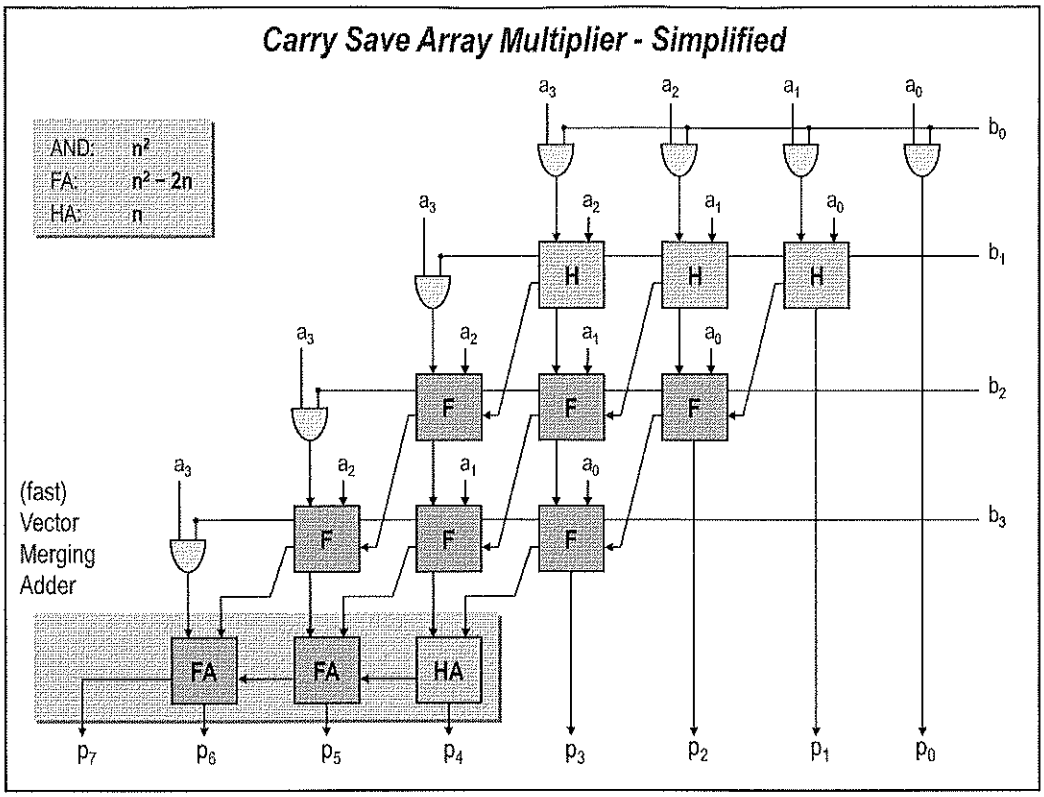


Carry Save Adder (CSA)

Multi-operand adder that does not propagate the carry to the following stage. Instead, it saves the carry propagation until all additions are completed and then take a final cycle to complete the carry propagation for all additions.

Carry Save Array Multiplier

Due to the large number of almost identical critical paths, increasing the performance of the structure of the ripple carry array through optimisation of the FA's yields marginal benefits. A more efficient realization can be obtained by noticing that the multiplication result does not change when the output carry bits are passed diagonally downwards instead of only to the right, as shown in the figure. An extra adder is included to generate the final result. This is called a Vector Merging Adder (VMA). The resulting multiplier is called a *carry save multiplier*, because the carry bits are not immediately added, but rather are "saved" for the next adder stage. In the final stage, carries and sums are merged in a fast carry-propagate (e.g., carry-look-ahead) adder stage: the Vector Merging Adder (VMA).



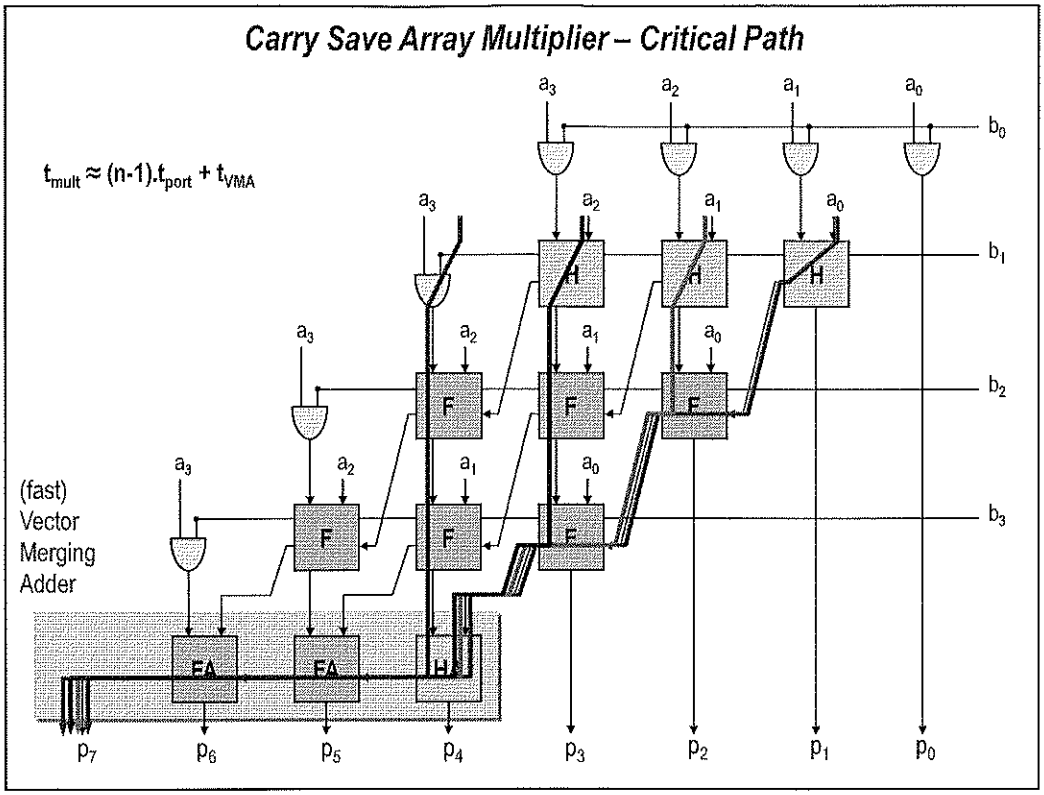
The half adders blocks (H) having the MSB of the multiplicand a as input only perform a partial product generation (no input carry, no output carry). They are replaced in the simplified structure by their equivalent and gate.

The carry save array multiplier (column ripple) has the same gate count as the carry ripple multiplier (row ripple):

AND: $n.n = n^2$

FA: $(n-1)^2-1 = n^2-2n$

HA: n

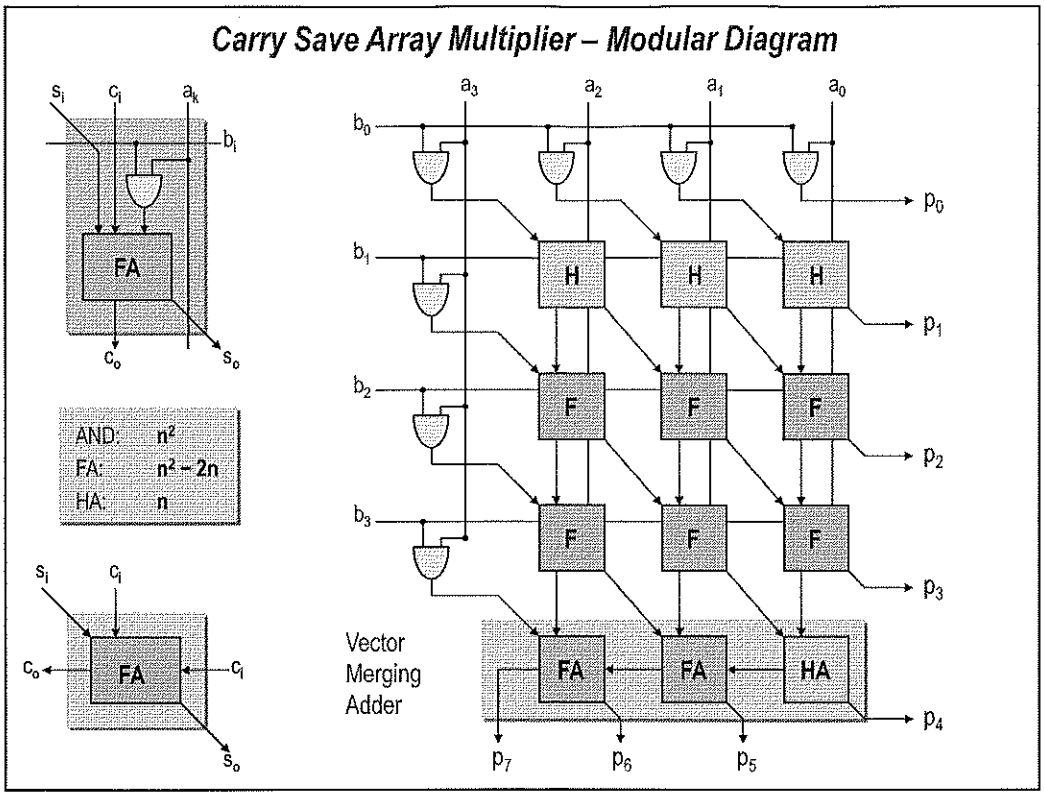


While the structure of the carry save multiplier has the same area cost, it has the advantage that its worst case critical path is shorter and uniquely defined, as highlighted in the figure and is expressed as:

$$t_{mult} \approx (n-1).t_{port} + t_{VMA}$$

This delay is almost the same as that from the carry ripple multiplier.

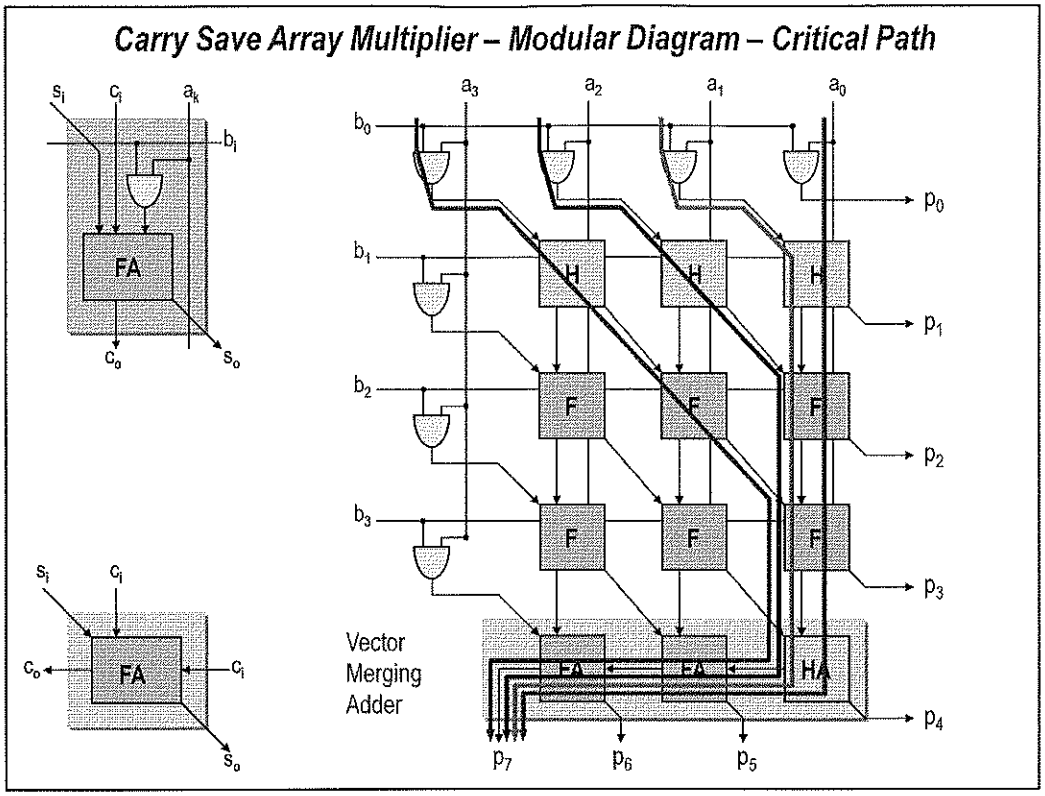
Because all the critical paths contain the delay of the Vector Merging Adder t_{VMA},t he critical path of the multiplier can be reduced by increasing only the speed of this Vector Merging Adder (carry look ahead, tree adder).



When mapping the carry save multiplier onto silicon, one has to take into account some other topological considerations. To ease the integration of the multiplier into the rest of the chip, itis advisable to make the outline of the module approximately rectangular.T he overall structure can easily be compacted into a rectangle, resulting in a very efficient layout. A floor plan for the carry save multiplier that achieves this goal is shown in the figure. Observe the regularity of the topology. This makes the generation of the structure amenable to automation.

The AND gates generate the partial products. Full adders and half adders add the generated partial products. Sum outputs are connected diagonally and carry outputs are connected vertically. The last row of adders (Vector Merging Adder) which are connected from left to right, generates the n most significant product bits.

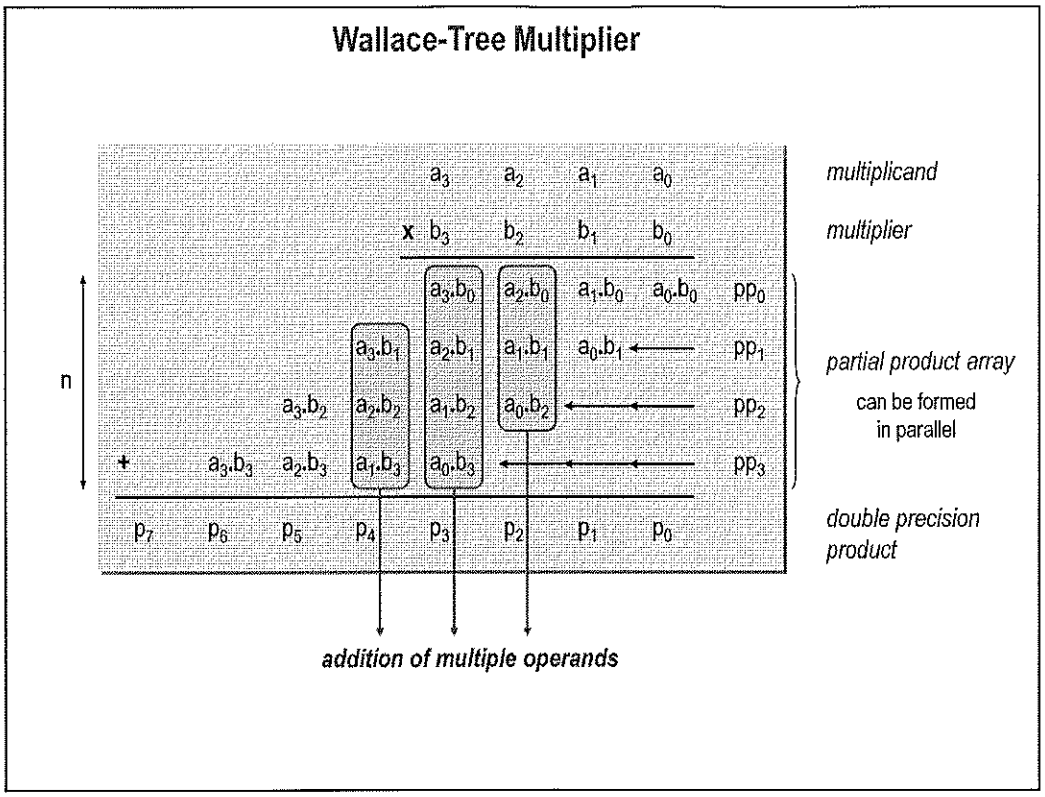
An n by n unsigned array multiplier uses n^2 AND gates, ($n^2 - 2n$) FA's and n HA's.



All the critical paths contain the delay of the Vector Merging Adder t_{VMA}. Almost half of the latency is due to the bottom row of adders. Therefore, the critical path of the multiplier can be reduced by increasing only the speed of this Vector Merging Adder (carry look ahead, tree adder). Although this decreases the overall delay it has a negative impact on the designs regularity.

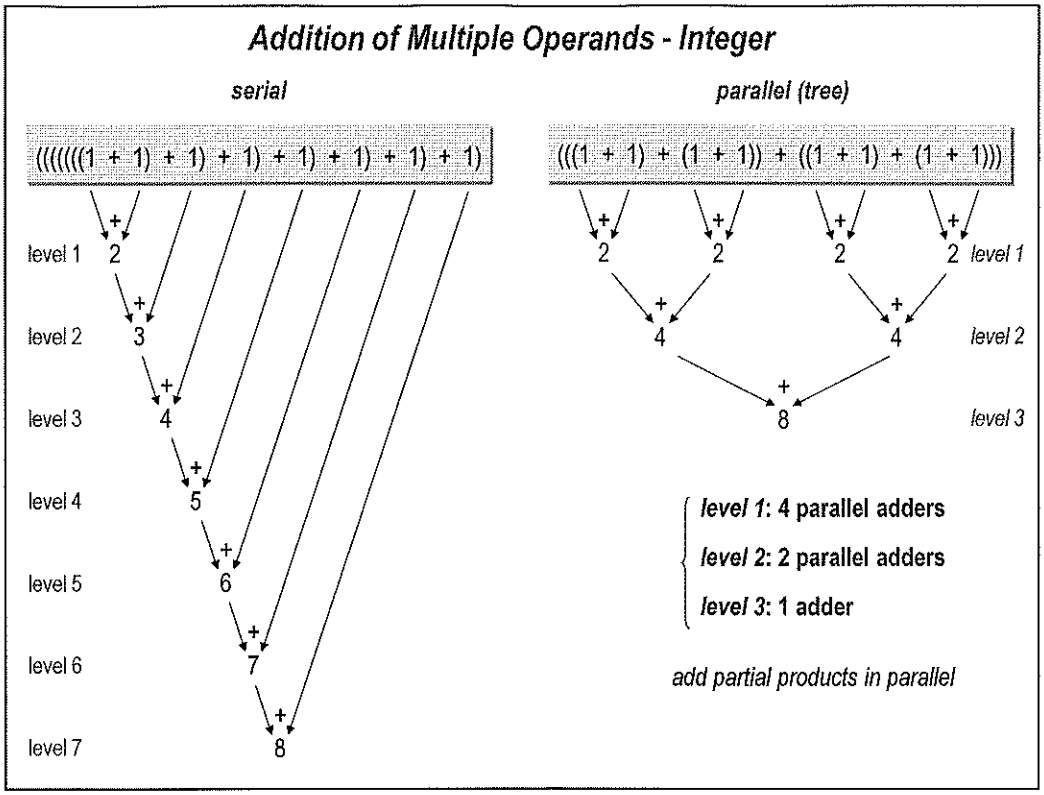
Carry Save Array Multiplier (conclusion):

- column ripple form
- fundamentally same delay and gate count as row ripple form
- ripple adder can be replaced with faster carry look ahead or carry tree adder
- regular routing pattern

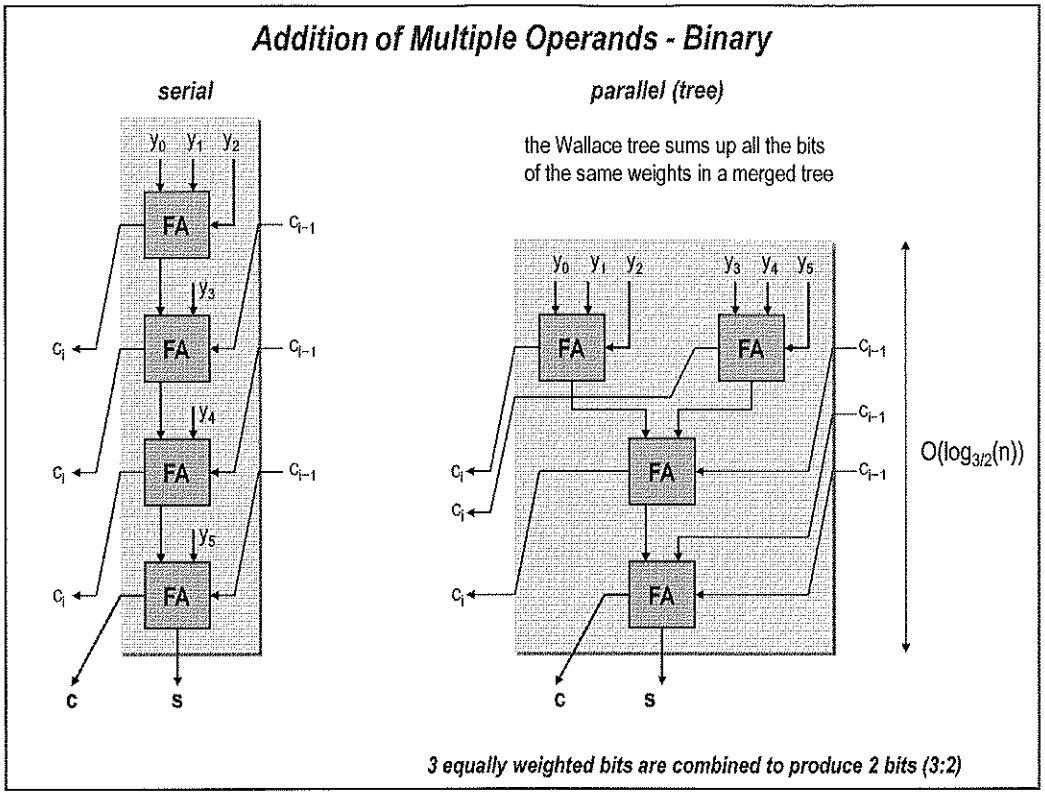


Wallace-Tree Multiplier

The partial-sum adders can also be rearranged in a treelike fashion, reducing both the critical path and the number of adder cells needed. Consider the simple example of four partial products each of which is four bits wide, as shown in the figure.T he full adders needed for this operation can be rearranged by observing that only column 3 in the array has to add four bits. All other columns are somewhat less complex.

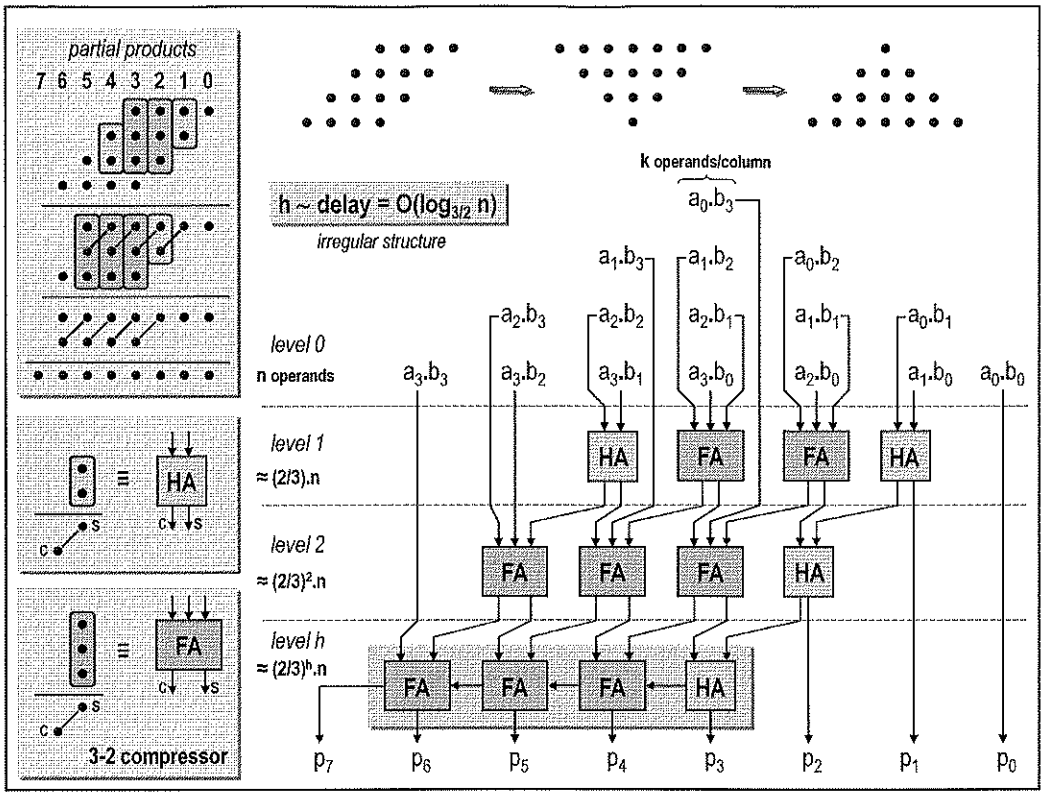


The addition is commutative and associative.By a reorganisation of the additions of multiple operands, the number of levels can be reduced (not the number of additions).



One can reduce the number of series carry save adders, by adding more of the partial products in parallel. A tree of adders is used.

A Wallace tree is an implementation of an adder tree designed for minimum propagation delay. Rather than completely adding the partial products in pairs like the ripple adder does, the Wallace tree sums up all the bits of the same weights in a merged tree. Usually full adders are used, so that *3 equally weighted bits are combined to produce 2 bits*: one (the carry) with weight of k+1 and the other (the sum) with weight k. Each layer of the tree therefore reduces the number of vectors by a factor of 3:2 (Another popular scheme obtains a 4:2 reduction using a different adder style that adds little delay in an ASIC implementation). The tree has as many layers as is necessary to reduce the number of vectors to two (a carry and a sum). A conventional adder is used to combine these to obtain the final product. The structure of the tree is shown. For a multiplier, this tree is pruned because the input partial products are shifted by varying amounts.

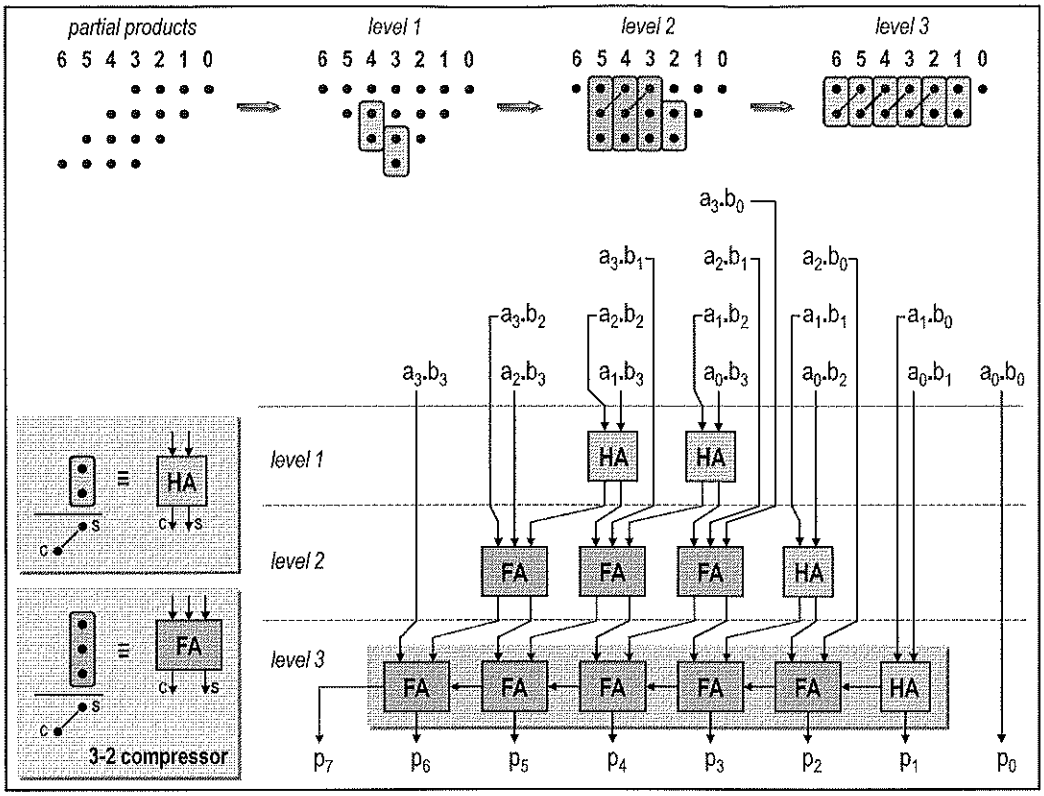


Tree multipliers generate all partial products in parallel using a tree of counters to reduce the partial products to sum and carry vectors and then sum these vectors using a fast carry-propagate adder. Tree multipliers offer a delay proportional to the logarithm of the operand length (n).

In the figure, the original matrix of partial products is reorganized into a tree shape to visually illustrate its varying depth. The challenge is to realize the complete matrix with a minimum depth and a minimum number of adder elements. The first type of operator that can be used to cover the array is a full adder, which takes three inputs and produces two outputs: the sum, located in the same column and the carry, located in the next one. For this reason, the FA is called a 3-2 *compressor.* It is denoted by a circle covering three bits. The other operator is the half-adder, which takes two input bits in a column and produces two outputs. The HA is denoted by a circle covering two bits.

The final step for completing the multiplication is to combine the result in the final adder.This can be a simple two-input adder but performance of this "vector-merging" operation is of key importance.

The presented structure is called the *Wallace-Tree multiplier*, and its implementation is shown. The tree multiplier realizes hardware savings for larger multipliers. The propagation delay is reduced as well. In fact, it can be shown that the propagation delay through the tree is equal to $O(\log_{3/2}(n))$ instead of O(n) for a carry save multiplier. While faster than the carry save structure for large multiplier word lengths, the Wallace multiplier has the disadvantage of being very *irregular*, which complicates the task of coming up with an efficient layout. Each tree has a different number of partial products and a different number of input carries.This irregularity is visible even in the four-bit implementation shown.



There are numerous other ways to accumulate the partial-product tree. A number of compression circuits has been proposed in the literature. They are all based on the concept that when full adders are used as 3:2 compressors, the number of partial products is reduced by two-thirds per multiplier stage. One can even go a step further and devise a 4-2 (or higher order) compressor.

The tree shown in the figure is iteratively covered with FA's and HA's, starting from its densest part (level 1).In a first step, HA's are introduced in columns 4 and 3.Th e reduced tree is shown (level 2).A second round of reductions creates a tree of depth 2 (level 3).

The final step for completing the multiplication is to combine the result in the final adder.This can be a simple two-input adder but performance of this "vector-merging" operation is of key importance.

A Wallace -Tree is often slower than a ripple carry multiplier in an FPGA

Many FPGA's have a highly optimized ripple carry chain connection. Regular logic connections are several times slower than the optimized carry chain, making it nearly impossible to improve on the performance of the ripple carry adders for reasonable data widths (at least 16 bits). Even in FPGA's without optimized carry chains, the delays caused by the complex routing can overshadow any gains attributed to the Wallace-Tree structure. For this reason, a Wallace-Tree multiplier does not provide any advantage over ripple carry multipliers in many FPGA's. In fact due to the irregular routing, they may actually be slower and are certainly more difficult to route.

5.2 Direct 2's Complement Integer Multiplication

$a = a_{n-1}\, a_{n-2} \cdots a_1\, a_0$ $\quad [a] = -a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$

x $\quad b = b_{n-1}\, b_{n-2} \cdots b_1\, b_0$ $\quad [b] = -b_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

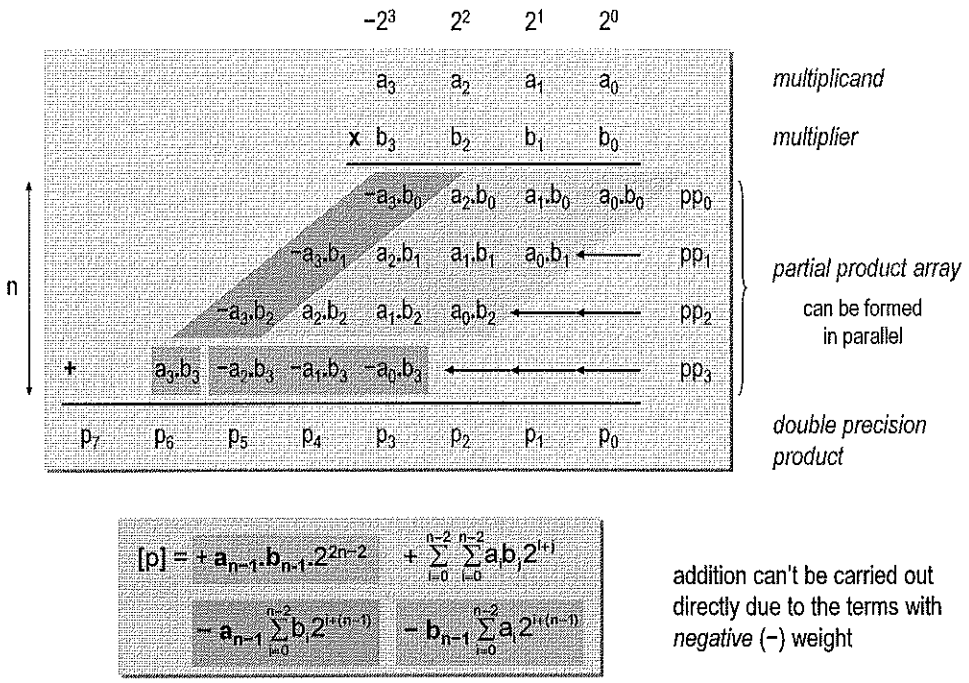
$axb = p = p_{2n-1}\, p_{2n-2} \cdots p_1\, p_0$ $\quad [p] = -p_{2n-1}.2^{2n-1} + \sum_{i=0}^{2n-2} p_i 2^i$

$$[p] = +a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} - a_{n-1}\sum_{i=0}^{n-2} b_i 2^{i+(n-1)} - b_{n-1}\sum_{i=0}^{n-2} a_i 2^{i+(n-1)}$$

$a_{n-1}.b_i$, $b_{n-1}.a_i$ — *negative terms* → subtractors

Multiplication of two's complement numbers generates signed partial products as shown in the figure. Since $a_{n-1}.b_i$ and $b_{n-1}.a_i$ have negative weights they should be subtracted rather than added. This makes the design difficult to implement because it requires adder and subtractor cells. Consequently several techniques have been proposed to handle partial products with negative and positive weight such as the Baugh-Wooley Algorithm.

Signed Multiplication

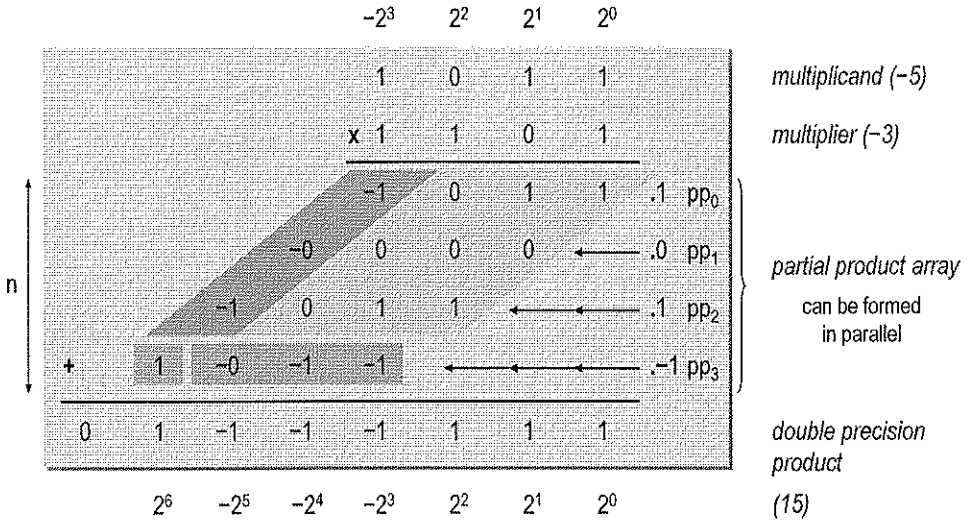


$$[p] = +a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{(i+j)} - a_{n-1}\sum_{i=0}^{n-2} b_i 2^{i+(n-1)} - b_{n-1}\sum_{i=0}^{n-2} a_i 2^{i+(n-1)}$$

addition can't be carried out directly due to the terms with *negative* (−) weight

Since $a_{n-1}.b_i$ and $b_{n-1}.a_i$ have negative weights, they should be subtracted rather than added. This makes the design difficult to implement because it requires adder and subtractor cells.

Example of Signed Multiplication



Addition can't be carried out directly due to the terms with negative weight.

This example of the multiplication of two's complement operands illustrates the problem of the terms with negative weight. They cannot be handled directly with conventional full adders.

Two's Complement Negative Number

$$[a] = -a_{n-1}.2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

$$[-a] = a_{n-1}.2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i$$

$$= a_{n-1}.2^{n-1} - 2^{n-1} + 2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i$$

(Booth Algorithm)

$$\sum_{i=k}^{v-1} 2^i = 2^{v-1} + \ldots + 2^k = 2^v - 2^k$$

$$= -(1-a_{n-1}).2^{n-1} + 2^{n-1} - \sum_{i=0}^{n-2} a_i 2^i$$

$$= -(1-a_{n-1}).2^{n-1} \left(+1 + \sum_{i=0}^{n-2} 2^i\right) - \sum_{i=0}^{n-2} a_i 2^i \quad \longleftarrow \quad \sum_{i=0}^{n-2} 2^i = 2^{n-2} + \ldots + 1 = 2^{n-1} - 1$$

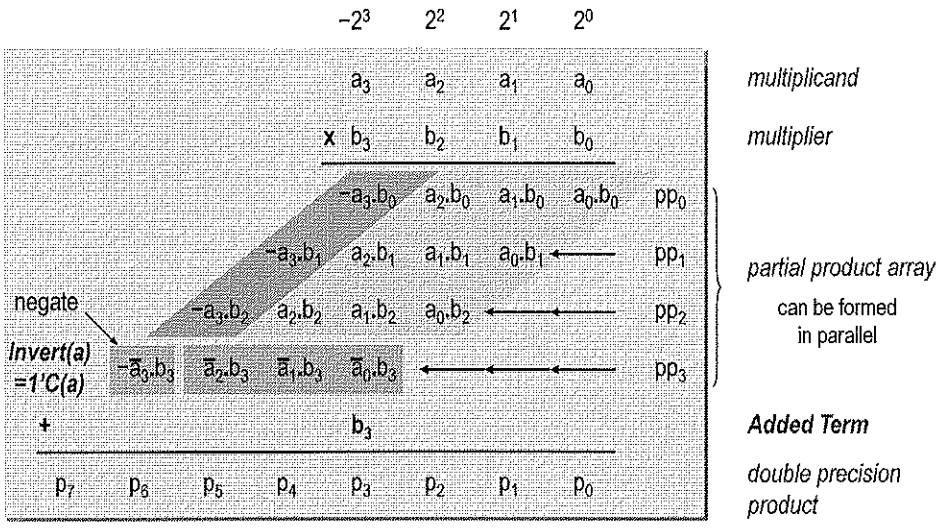
$$= -(1-a_{n-1}).2^{n-1} + \sum_{i=0}^{n-2} (1-a_i).2^i + 1$$

$$= \underbrace{-(\bar{a}_{n-1}).2^{n-1} + \sum_{i=0}^{n-2} (\bar{a}_i).2^i}_{\text{1's complement}} + 1$$

$$a = (\overline{-a}) + 1$$

The complement –a of a two's complement number a can be found by inverting the bits and adding an LSB bit.

Signed Multiplication with 2's Complement of pp_{n-1}



Addition can't be carried out directly due to the terms with negative weight.
Sign extension is used to solve this problem.

The last partial product results from the multiplication of the sign bit of the multiplier (b_3) with the multiplicand a.

$$pp_3 = a.(-b_3.2^3) = (-a).b_3.2^3$$

The complement –a of the multiplicand a is taken:

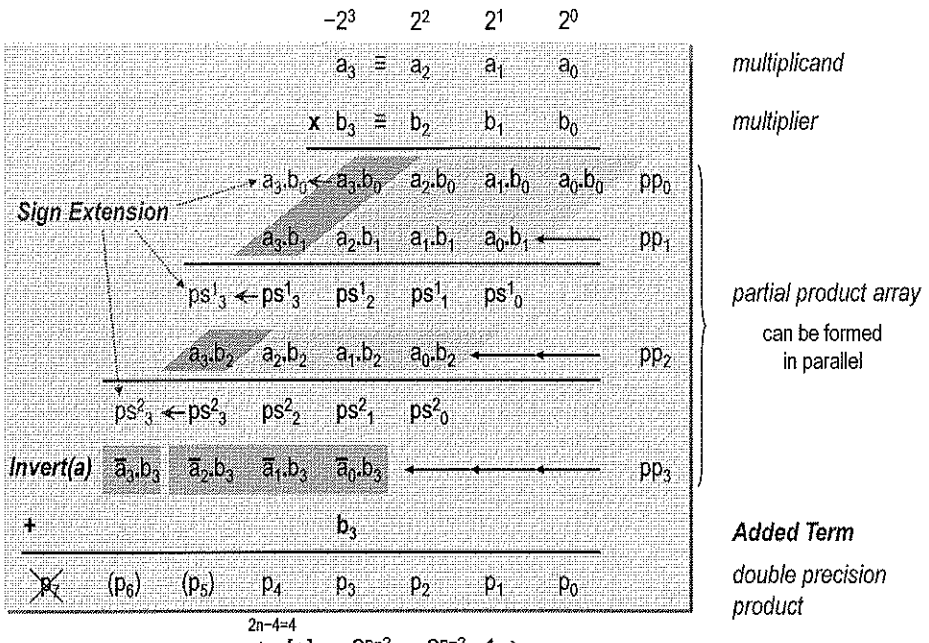
$$2'sC(a) = 1'sC(a) + 1$$

The complemented partial product becomes:

$$2'sC(pp_3) = (1'sC(a) + 1).b_3.2^3 = 1'sC(a).b_3.2^3 + b_3.2^3$$

This can be realised by inverting the a_i bits in the last partial product and adding the term $b_3.2^3$.

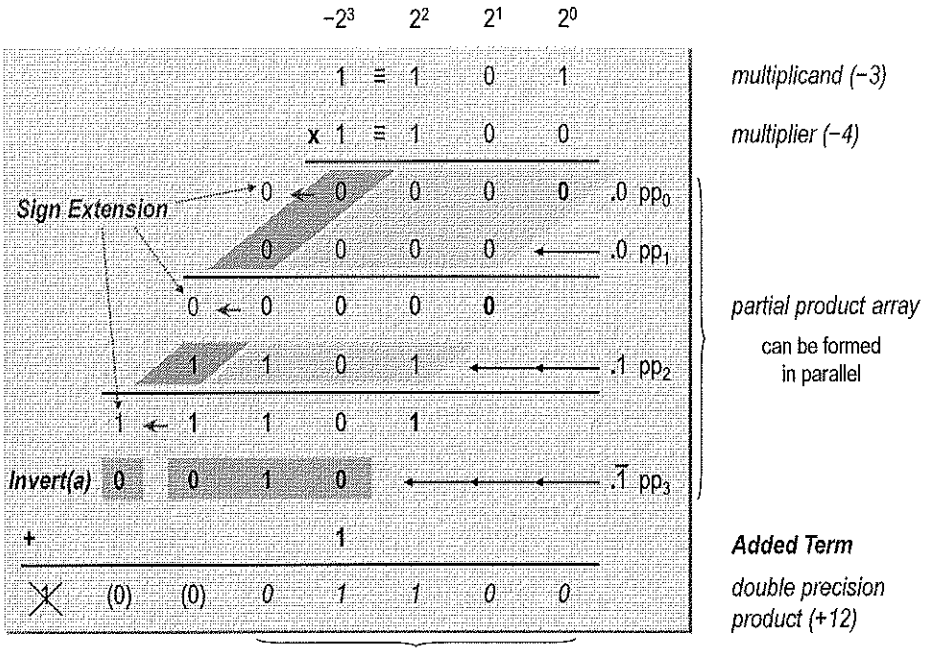
Signed Multiplication with 2's Complement of pp_{n-1} and Sign Extension



The partial products are not aligned. Due to the sign bits with negative weight, sign extension of the twos complement numbers is used. To limit the length of the partial products the are only extended with one bit. This reduces the hardware needed but also limits the range of the multiplicand and the multiplier.

To eliminate overflow, the MSB of the n-bit multiplier, b_{n-1}, is defined as a guard bit and is set equal to b_{n-2}. The multipliers must now lie in the range $[-2^{n-2}, 2^{n-2}-1[$. Also the multiplicand has a guard bit, so $a_{n-1} = a_{n-2}$.

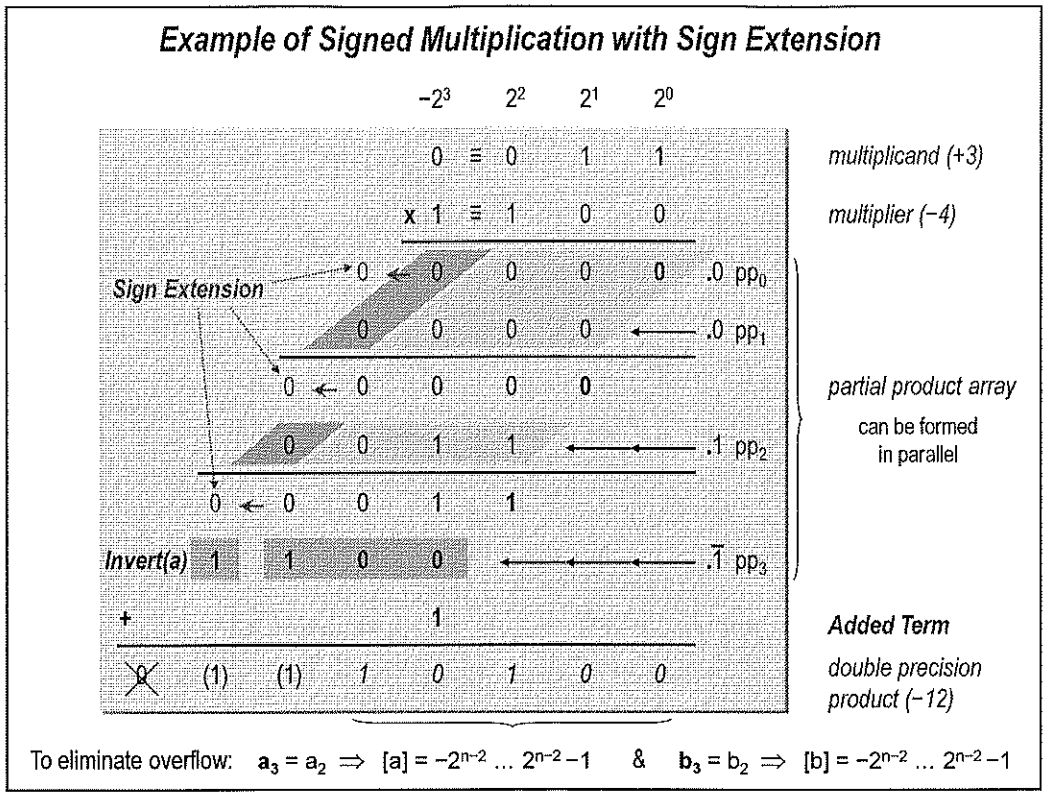
Example of Signed Multiplication with Sign Extension



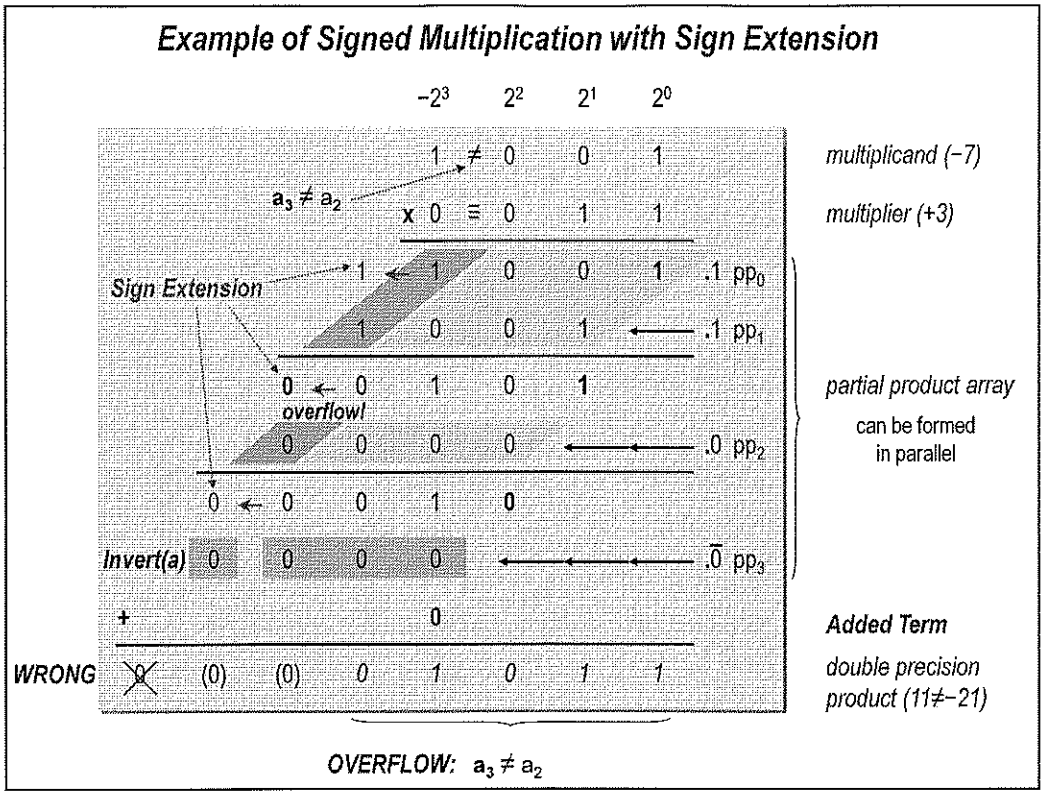
To eliminate overflow: $\mathbf{a_3} = a_2 \Rightarrow [a] = -2^{n-2} \ldots 2^{n-2}-1$ & $\mathbf{b_3} = b_2 \Rightarrow [b] = -2^{n-2} \ldots 2^{n-2}-1$

To eliminate overflow, the MSB of the n-bit multiplicand, a_{n-1}, is defined as a guard bit and is set equal to a_{n-2}. The multiplicands must now lie in the range $[-2^{n-2}, 2^{n-2}-1[$. Also $b_{n-1} = b_{n-2}$ for the multiplier.

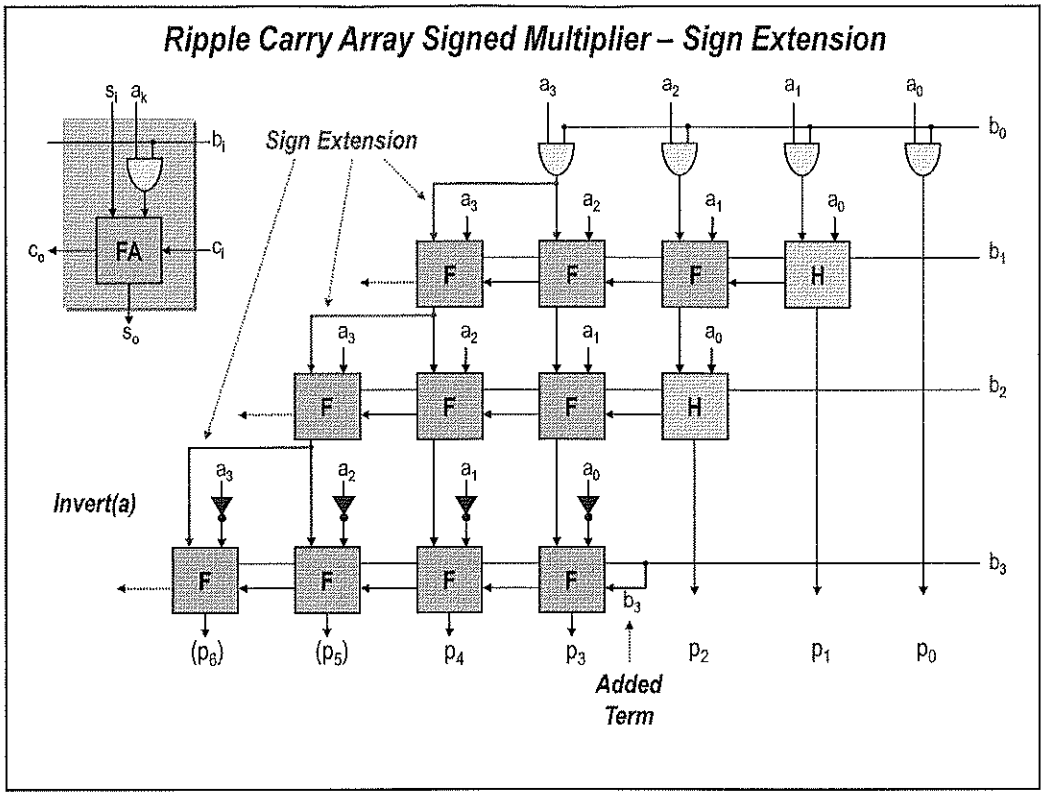
The example illustrates the principle of the complemented last partial product and the sign extension of the other partial products to implement the direct multiplication of signed integers.



To eliminate overflow, the MSB of the n-bit multiplicand, a_{n-1}, is defined as a guard bit and is set equal to a_{n-2}. The multiplicands must now lie in the range $[-2^{n-2}, 2^{n-2}-1[$. Also $b_{n-1} = b_{n-2}$ for the multiplier.



When the multiplicand (and/or multiplier) lies outside the range $[-2^{n-2}, 2^{n-2}-1[$, overflow occurs due to the limited sign extension (only one bit per partial product).



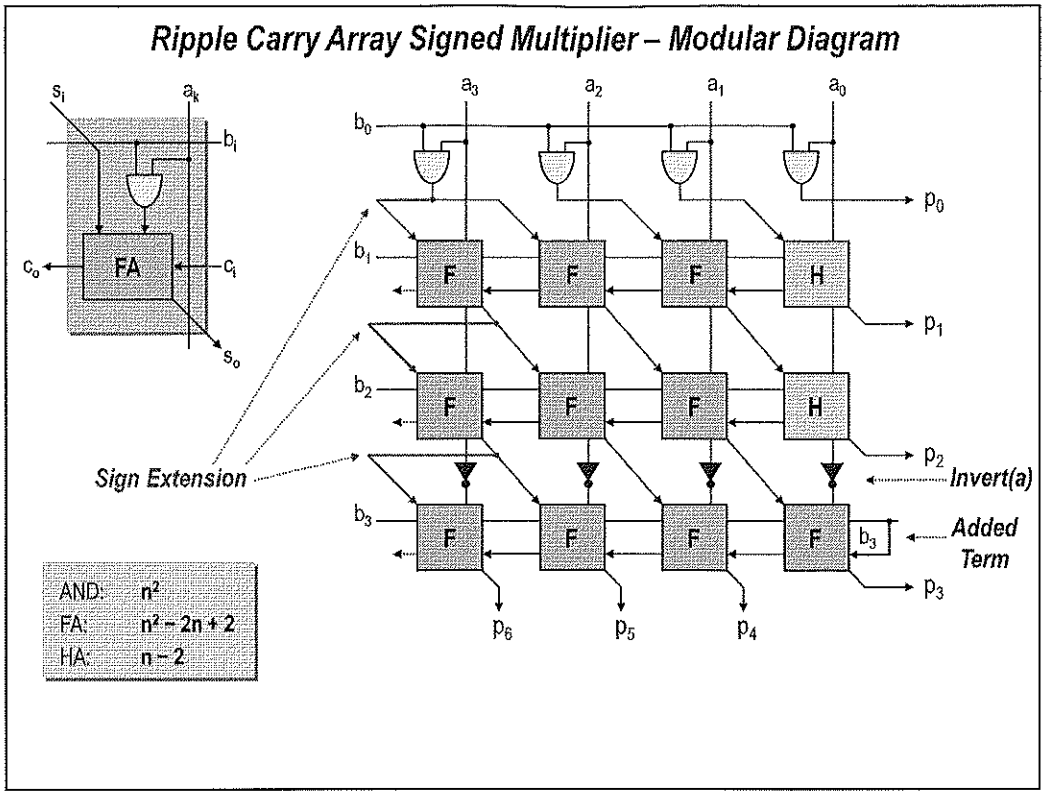
The array multiplier for signed numbers based on the complement of the last partial product, is illustrated.

The complemented partial product:

$$2'sC(pp_3) = (1'sC(a) + 1).b_3.2^3 = 1'sC(a).b_3.2^3 + b_3.2^3$$

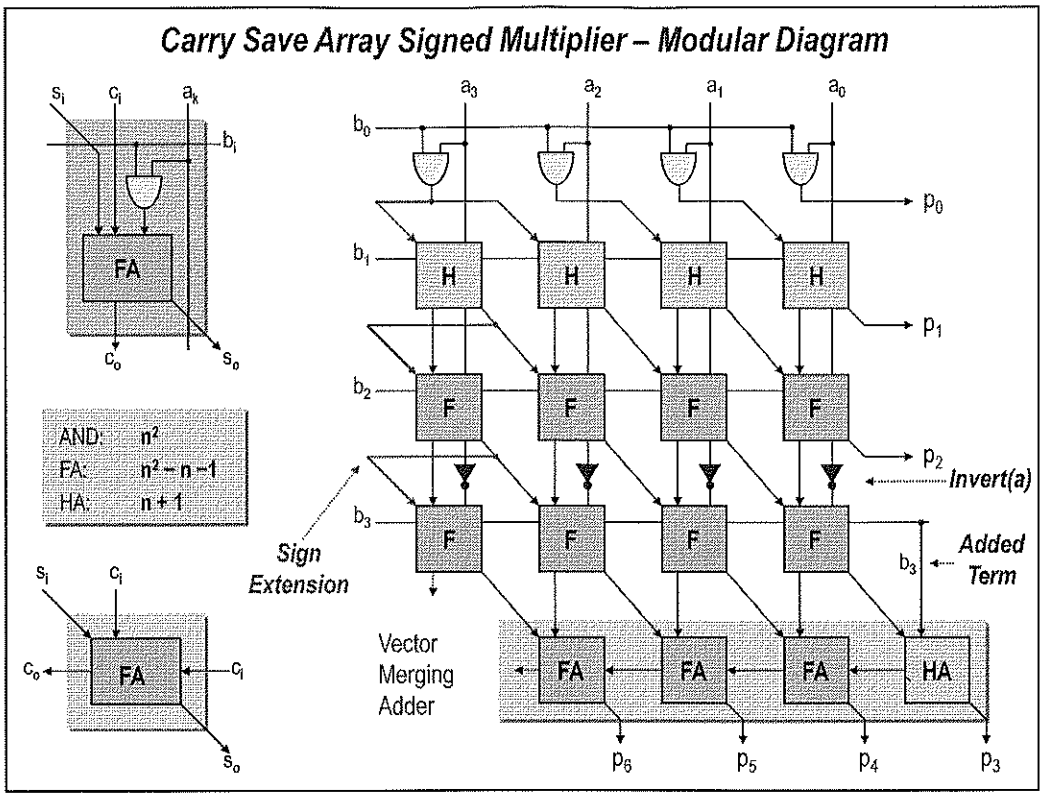
can be realised by inverting the a_i bits in the last partial product and adding the term $b_3.2^3$.

To align the partial products before addition, sign extension is used. Because only one bit sign extension is implemented, the range of the multiplicand and the multiplier must be limited to prevent internal overflow during addition of the partial products.



The overall structure can easily be compacted into a rectangle, resulting in a very efficient layout. A floor plan for the ripple carry array signed multiplier that achieves this goal is shown in the figure. Observe the regularity of the topology. This makes the generation of the structure amenable to automation.

An n by n array signed multiplier uses n^2 AND gates, (n^2 – 2n +2) FA's and (n – 2) HA's. The n inverters can be combined with the corresponding AND gates. Two extra Full Adders are needed due to the sign extension of the first partial product and the added term in the last partial product.

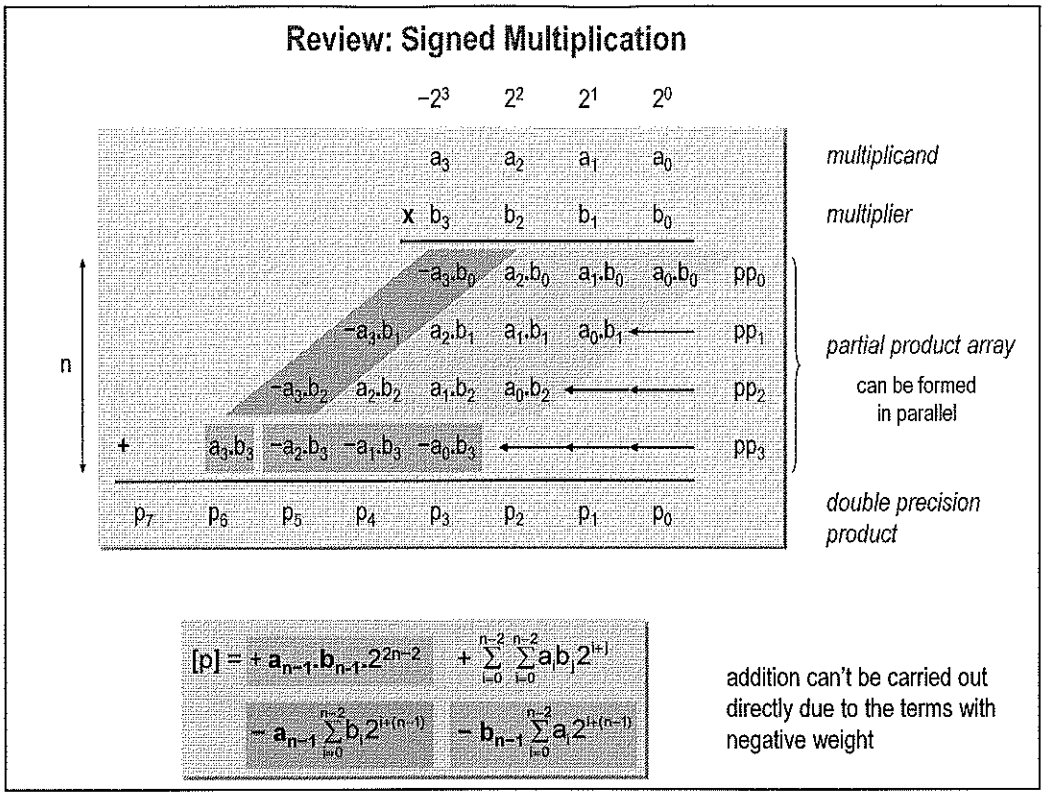


Carry Save Array Multiplier

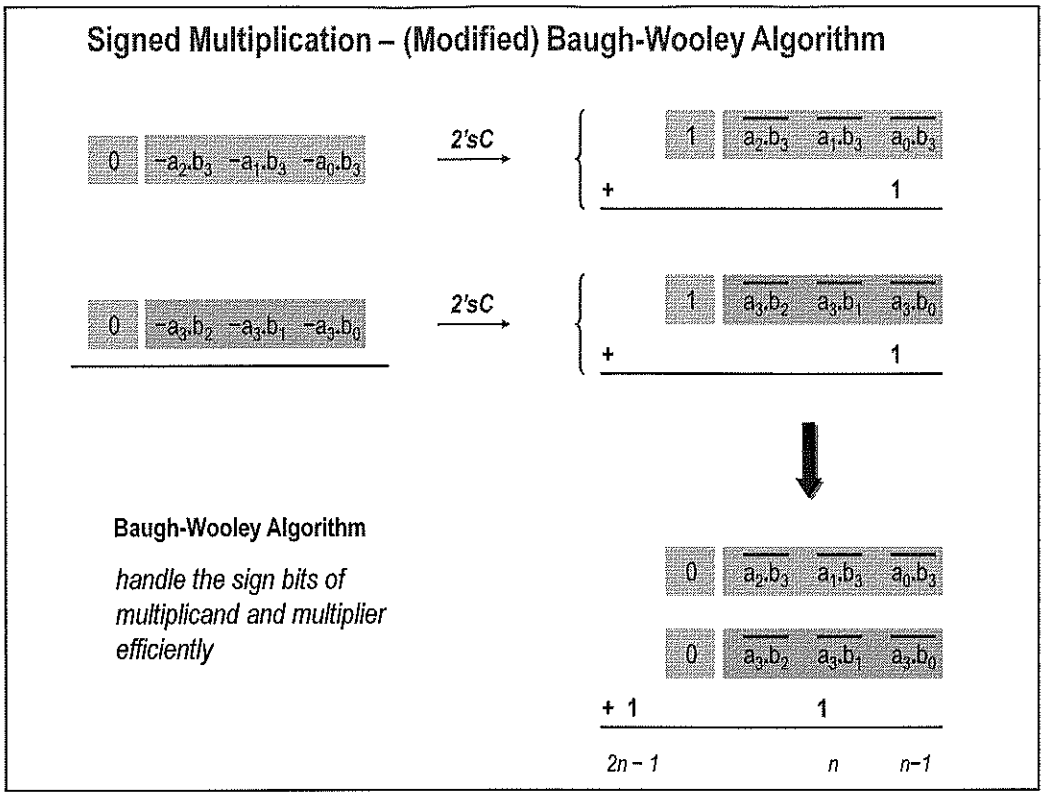
The multiplication result does not change when the output carry bits are passed vertically downwards to the next partial product instead of only to the left in the own partial product, as shown in the figure. An extra adder is included to generate the final result: the Vector Merging Adder (VMA). The resulting multiplier is called a *carry save multiplier*, because the carry bits are not immediately added, but rather are "saved" for the next adder stage. In the final stage, carries and sums are merged in a fast carry-propagate (e.g., carry-look-ahead) adder stage.

The AND gates generate the partial products. Full adders and half adders add the generated partial products. Sum outputs are connected diagonally and carry outputs are connected vertically. The last row of adders (Vector Merging Adder) which are connected from left to right, generates the n most significant product bits.

An n by n signed carry save array multiplier uses n^2 AND gates, $(n^2 - 2n - 1)$ FA's and $(n + 1)$ HA's.

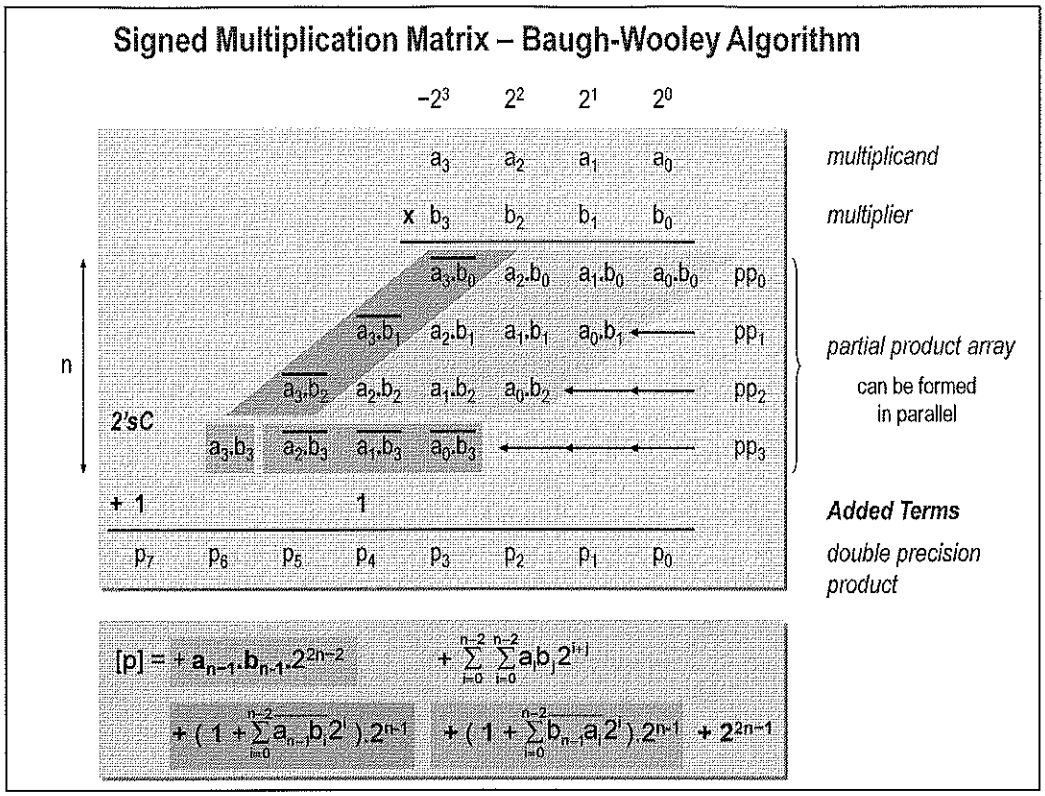


Since $a_{n-1}.b_i$ and $b_{n-1}.a_i$ have negative weights they should be subtracted rather than added. This makes the design difficult to implement because it requires adder and subtractor cells.



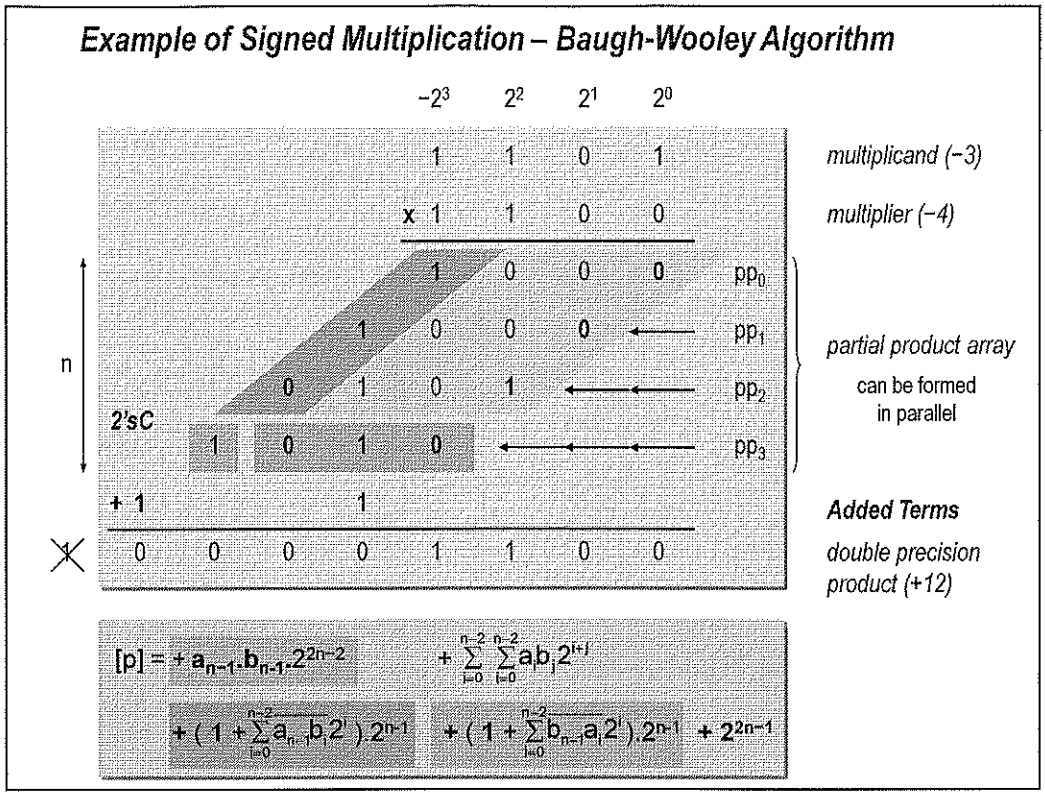
The BaughWooley algorithm provides a method for modifying the partial product matrix so that all the partial product bits have positive weights.

Two's complement multiplication is often realized using a variation of the Baugh-Wooley algorithm called the Complemented Partial Product Word Correction Algorithm. With this implementation partial product bits containing an a_{n-1} or b_{n-1},bu t not both, are complemented and ones are added to columns n and (2n − 1). This is equivalent to taking the two's complement of the two negative terms in the equation of the signed multiplication.

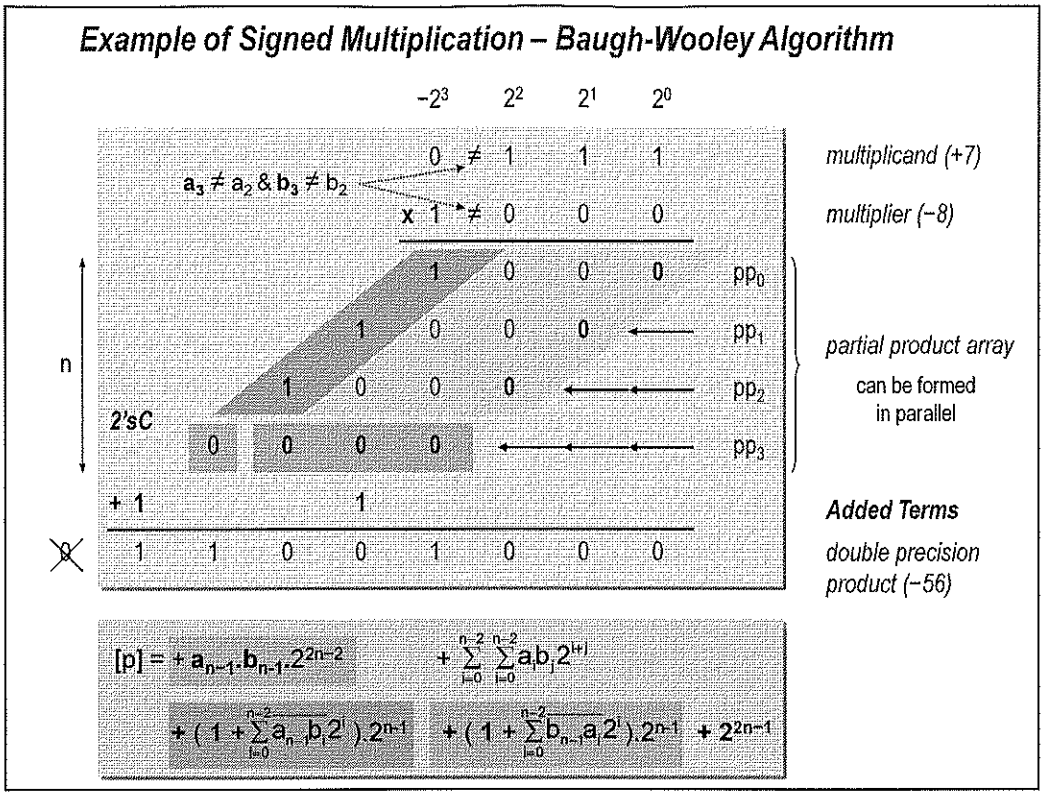


The multiplication matrix for two's complement multiplication based on the Complemented Partial Product Word Correction Algorithm is shown in the figure.

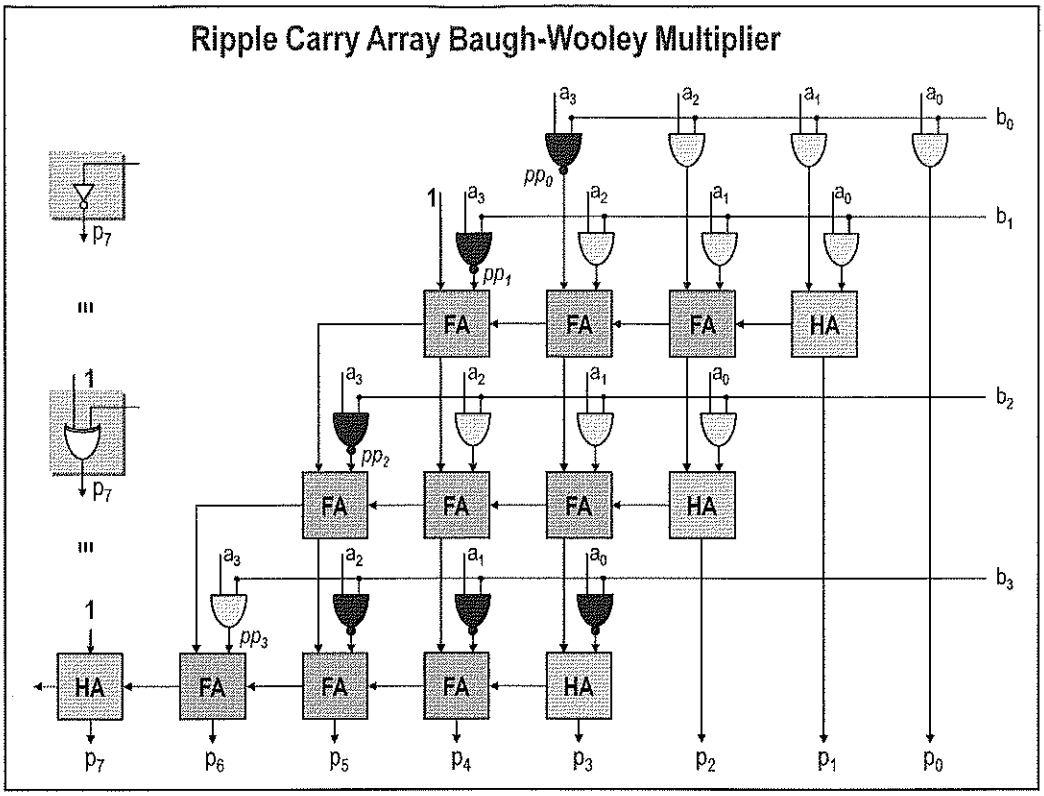
By taking the two's complement of the two negative terms in the equation of the signed multiplication results in a new equation with only positive terms.



An example illustrates the functionality of the multiplication matrix for two's complement multiplication based on the Complemented Partial Product Word Correction Algorithm.



There is no possibility for internal overflow, so the normal two's complement range of the multiplicand and multiplier can be used.

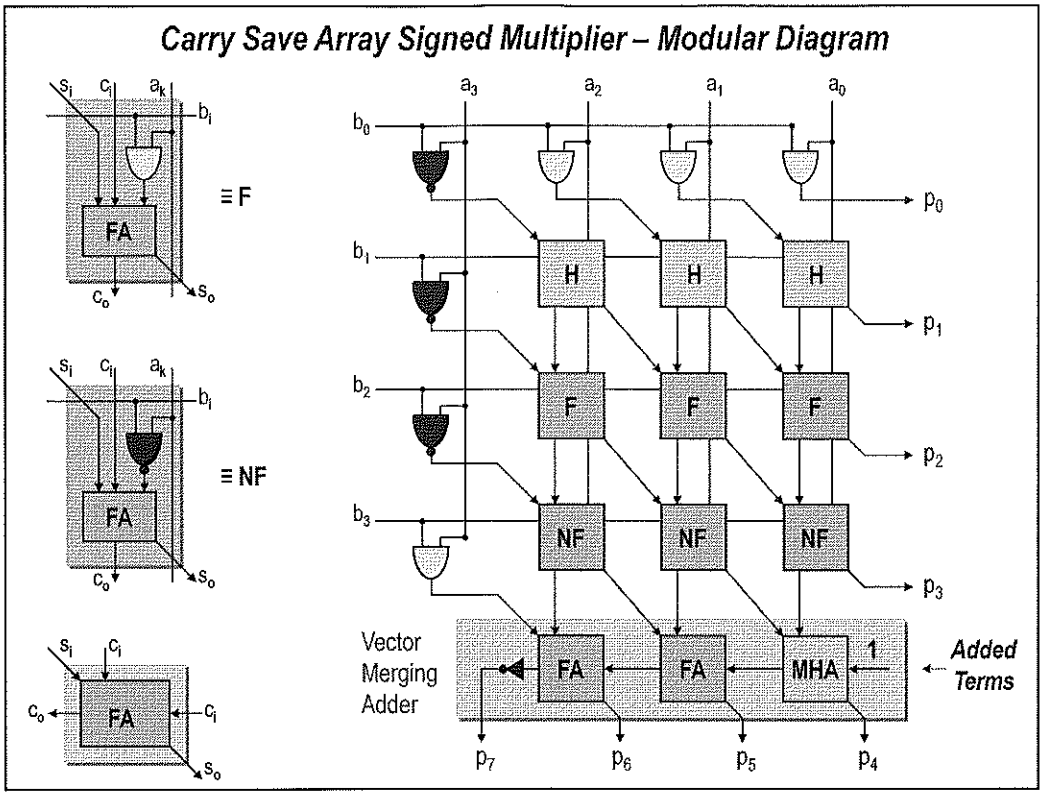


The design of an array multiplier that uses the Complemented Partial Product Word Correction Algorithm is shown. It is similar to the unsigned array multiplier design.

The AND gates in the leftmost column are replaced by NAND gates (except for the last row). Also in the last row the AND gates are replaced by NAND gates (except for the last column).

A '1' is added in column n and (2n − 1).

The last product bit p_{2n-1} is inverted to add the one in column n

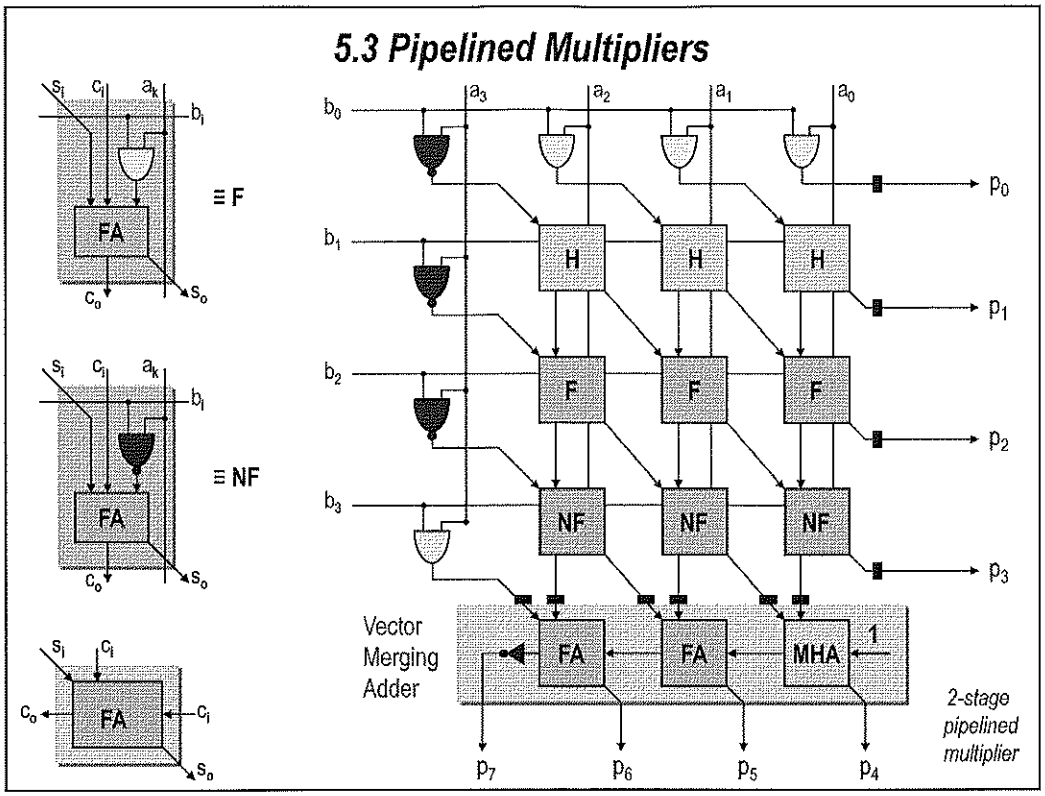


Also the design of an carry save array multiplier that uses the Complemented Partial Product Word Correction Algorithm is similar to the unsigned carry save array multiplier design.

AND gates in the leftmost column are replaced by NAND gates and the last row of modified full adder bocks F are replaced by negating modified full adder bocks NF.

The modified half adder MHA in the bottom right corner is a half adder that takes the sum and carry bits of the previous row and adds them with '1'. This cell has approximately the same area and delay as a regular half adder.

The last product bit p_{2n-1} is inverted to add the '1' in column $(2n - 1)$. Inverting p_{2n-1} has the same effect as adding '1' in column $(2n - 1)$, because the carry out from this column is ignored.



The delay in the multiplier poses a major limitation on the maximum clock rate that can be attained. Array multipliers can be configured to allow a pipelined mode of operation, where the execution of separate multiplications overlaps. If this mode of operation is applied, the long delay associated with the carry propagating addition in the array multiplier can be minimized, since it determines the throughput of the pipeline. This approach yields extremely high speed implementations.

2-Stage Pipelined Multiplier

The structure of the pipelined multiplier is shown. The basic cells shown here are identical to that in the unpipelined multiplier.

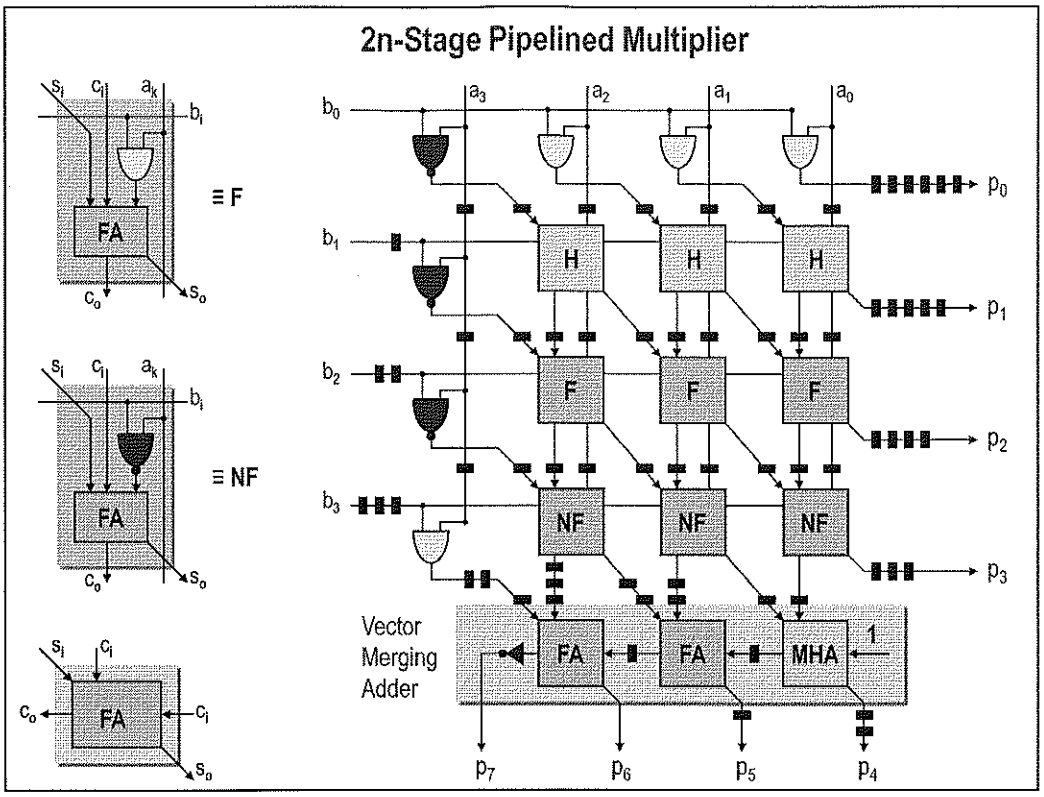
The delay of a multiplier is approximately $2.n.t_{FA}$, so 2 times t_{nadd}, the delay of an n-bit ripple carry adder.

The combinatorial logic of the carry save adder can be divided in two, by inserting registers a the inputs of the Vector Merging Adder. To align the product bits, the n LSB bits of the product must be delayed by 1 clock cycle.

The clock speed can be doubled (if both parts of the logic have approximately the same delay = balanced pipeline stages).

pipelined clock period = $T_{clk} > t_{nadd}$

latency = $2.T_{clk} \approx 2.t_{nadd}$



2n-Stage Pipelined Multiplier

The structure of the pipelined multiplier is shown. The basic cells shown here are identical to that in the unpipelined multiplier.

Registers are needed to propagate the multiplier and multiplicand bits to their destination and also to propagate the product bits that have been completed, which is done in parallel with the generation of new product bits.

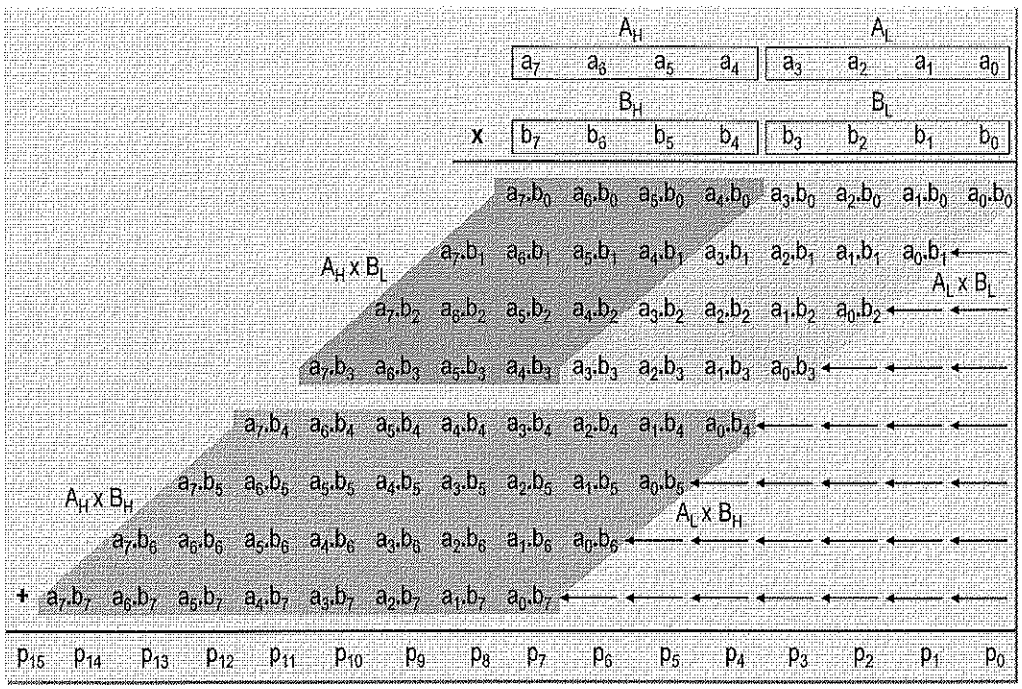
Pipeline registers are inserted between the rows. This allows a carry propagation of only one position between any two consecutive rows. Registers are inserted for the bits of the multiplier b, to align them with the timing of the partial products. MSB bits are scheduled to arrive later.

The clock speed depends only on the delay in the cells of the multiplier.

pipelined clock period = $T_{clk} > t_{AND} + t_{FA}$

latency = $(n-1).T_{clk} \approx (2n-2).(t_{AND} + t_{FA})$

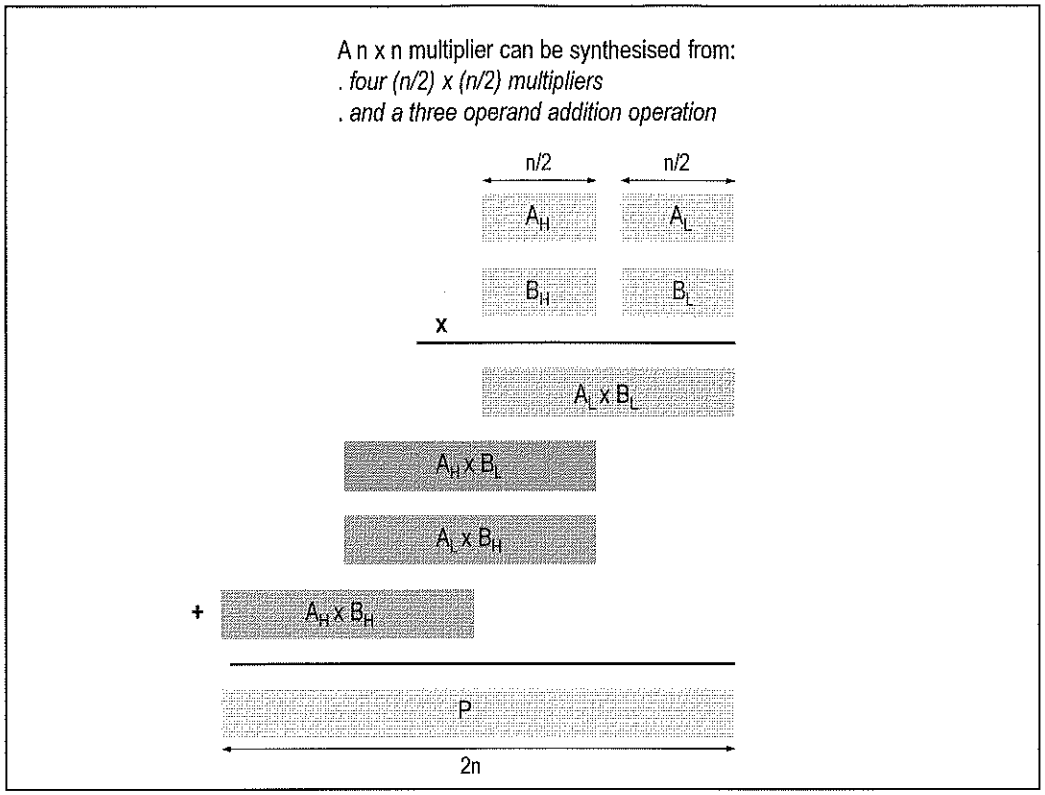
5.4 Modular Multipliers



Based on the equation:

$$(A_H + A_L)x(B_H + B_L) = A_HxB_H + A_LxB_H + A_LxB_H + A_LxB_L$$

larger multipliers can be realised based on the use of multiple smaller multipliers. The subproducts (A_HxB_H ,A_LxB_H , A_LxB_H , A_LxB_L) must be aligned to ensure the correctness of the result.



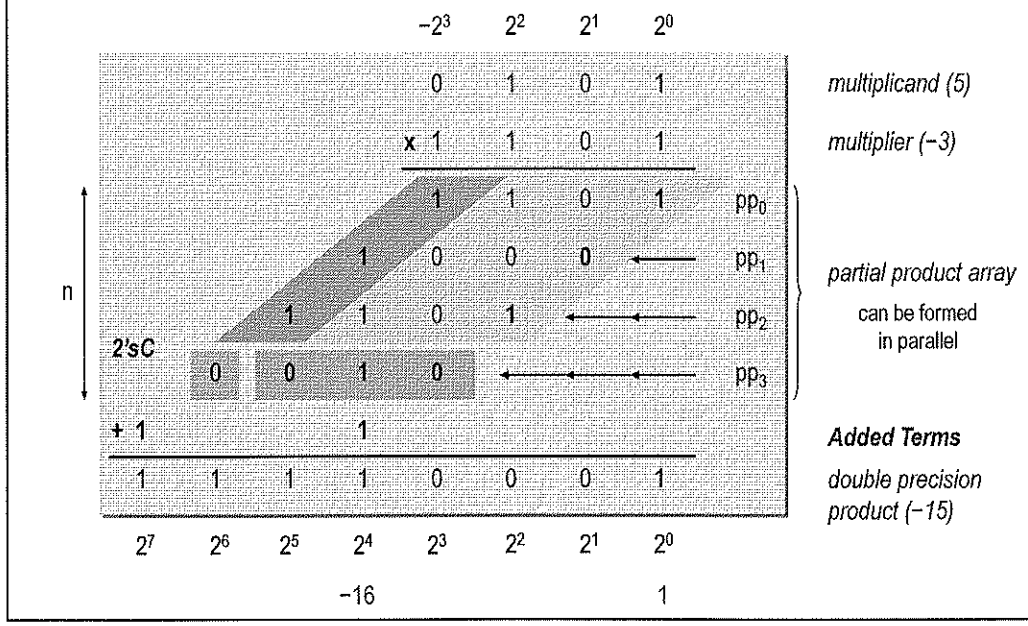
A n x n multiplier can be synthesised from:

. four (n/2) x (n/2) multipliers

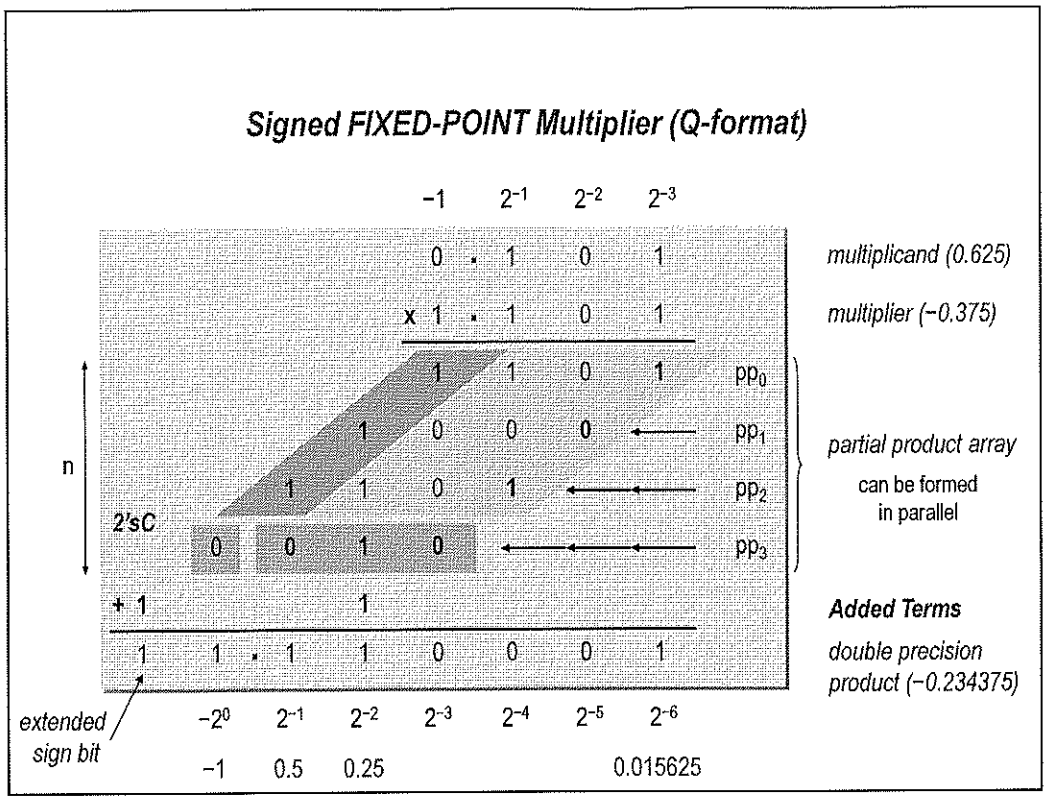
. and a three operand addition operation

5.5 Fixed-Point Multiplication vs Integer Multiplication

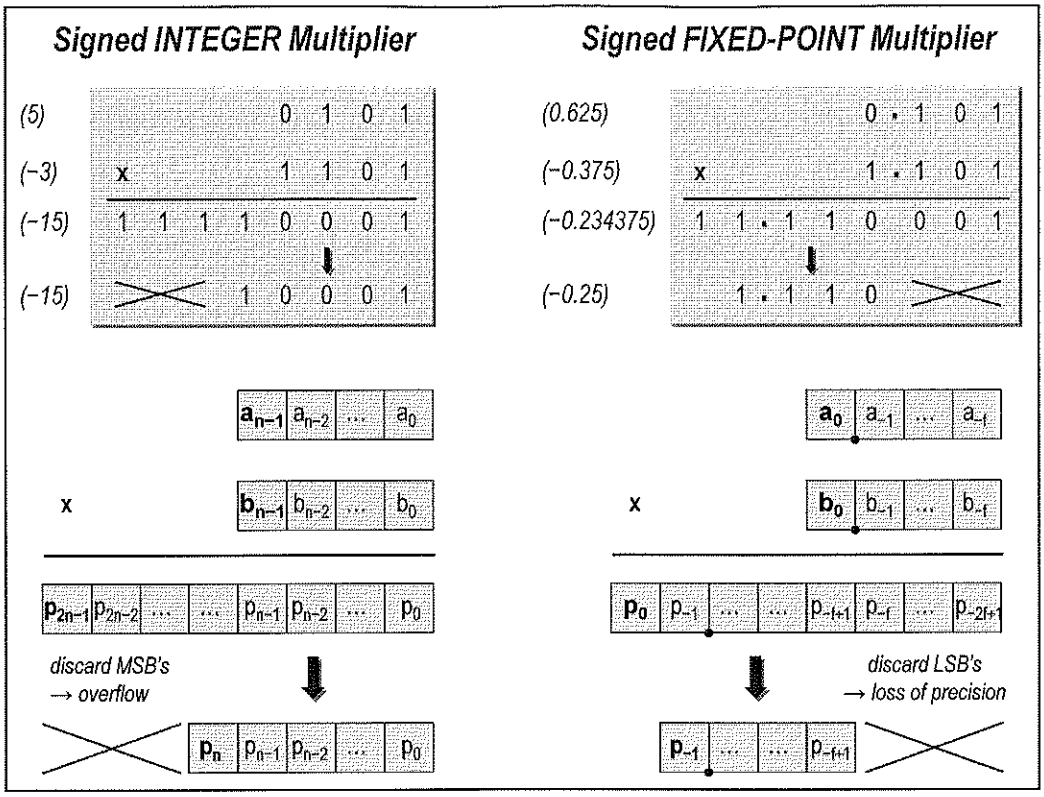
Signed INTEGER Multiplier



The example illustrates the functionality of the multiplication matrix for the multiplication of two's complement INTEGER numbers, based on the Complemented Partial Product Word Correction Algorithm.



The example illustrates the functionality of the multiplication matrix for the multiplication of two's complement FIXED-POINT numbers, based on the Complemented Partial Product Word Correction Algorithm.

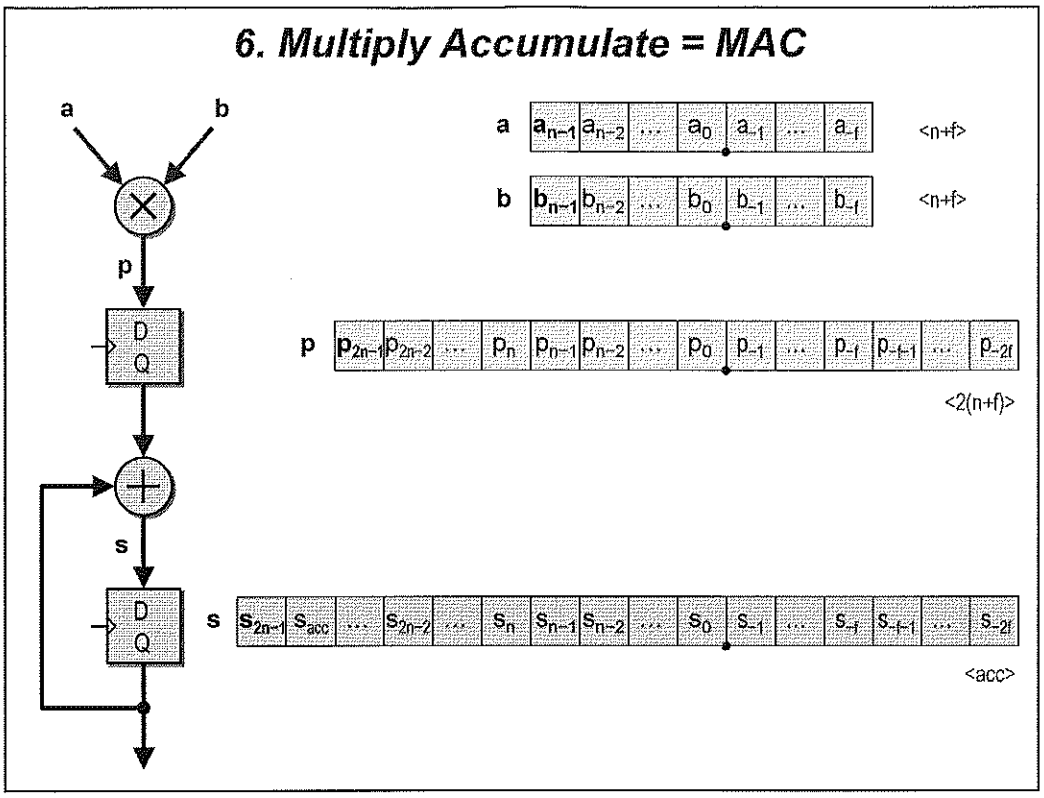


Signed Integer Multiplier

The MSB's are discarded. This can result in overflow errors.

Signed Fixed-Point Multiplier

The LSB's are discarded. This can result in loss of precision.



The multiply accumulate (MAC) operation is the kernel of various Digital Signal Processing algorithms.

The structure of a MAC unit is illustrated in the figure. The MAC unit presented consists of an (n+f)-bit multiplier and a acc-bit accumulator.

References

Baugh, Wooley, "A two's complement parallel array multiplication algorithm", IEEE Trans. Computers, 22: 1045-1047, Dec. 1973.

Parhami, "Computer Arithmetic", Oxford Univ. Press, 1999

HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

Digitale Synthese

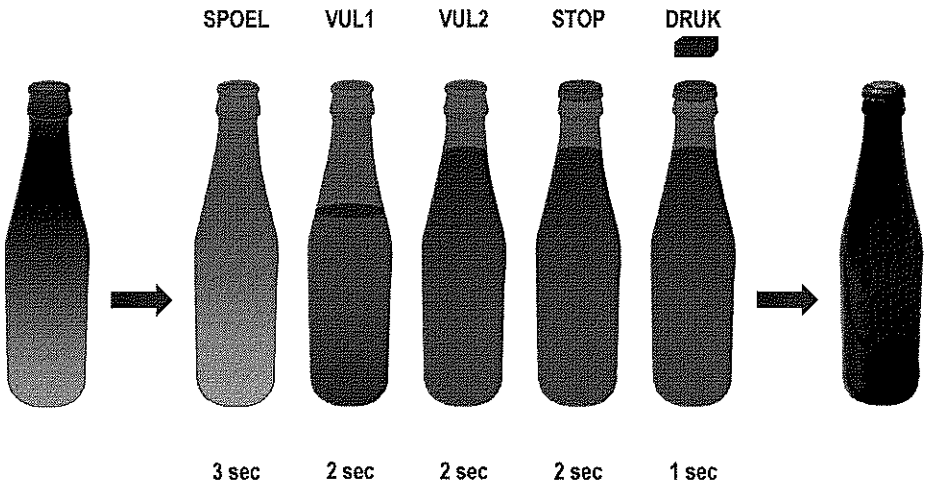
High Throughput Design

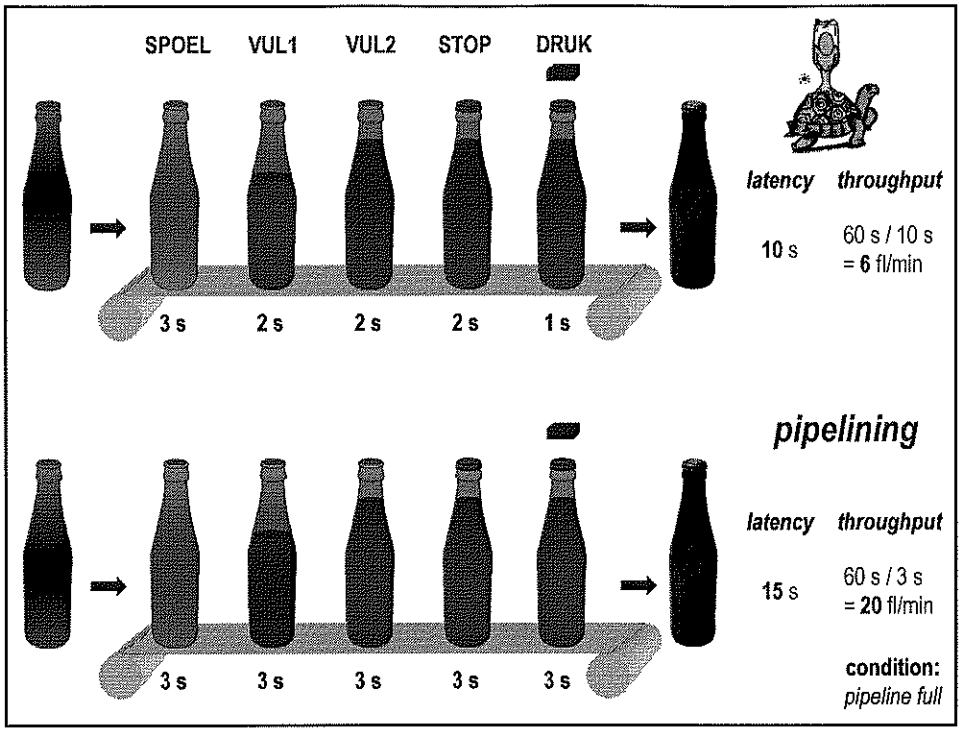
Analogie

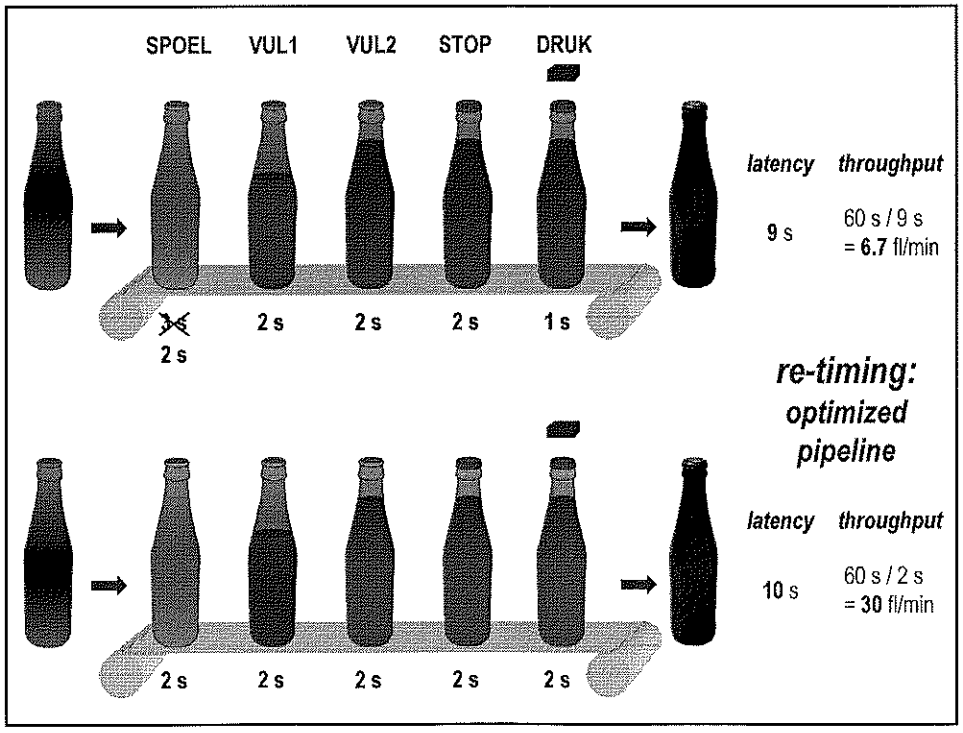
EmSD
Embedded System Design

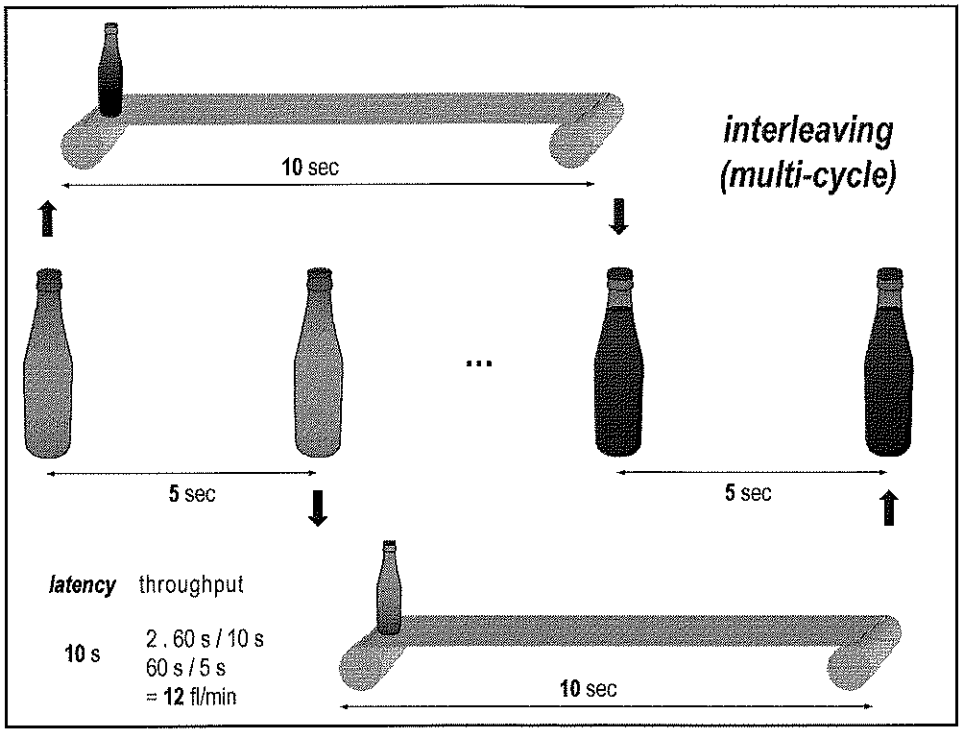
ir. J. Meel
april 2008

Stappen bij het Bottelen van Bier




SPOEL
VUL1
VUL2
STOP
DRUK
3 s
2 s
2 s
2 s
1 s
latency
throughput
10 s
60 s / 10 s
= 6 fl/min
pipelining
3 s
3 s
3 s
3 s
3 s
latency
throughput
15 s
60 s / 3 s
= 20 fl/min
condition:
pipeline full


SPOEL
VUL1
VUL2
STOP
DRUK
2 s
2 s
2 s
2 s
1 s
latency
throughput
9 s
60 s / 9 s
= 6.7 fl/min
re-timing:
optimized
pipeline
2 s
2 s
2 s
2 s
2 s
latency
throughput
10 s
60 s / 2 s
= 30 fl/min


10 sec
interleaving
(multi-cycle)
5 sec
5 sec
latency throughput
10 s
2 . 60 s / 10 s
60 s / 5 s
= 12 fl/min
10 sec

HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT SINT-KATELIJNE-WAVER

Digitale Synthese

High Throughput Design

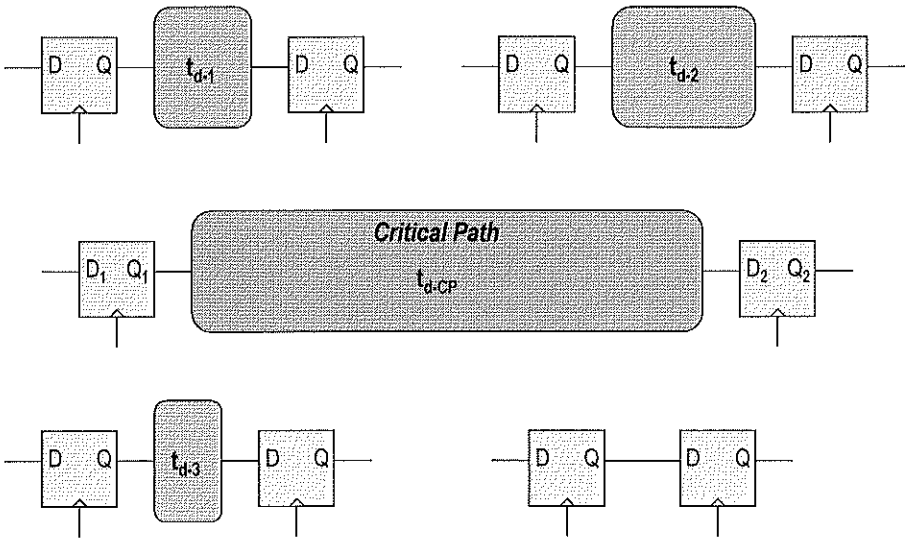
EmSD Embedded System Design

ir. J. Meel

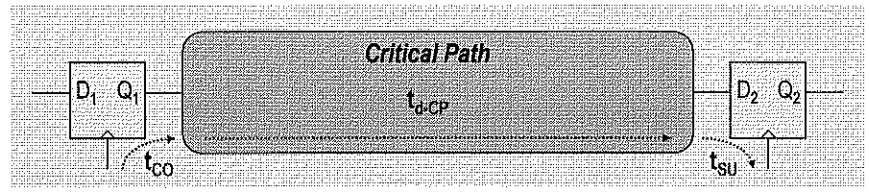
april 2008

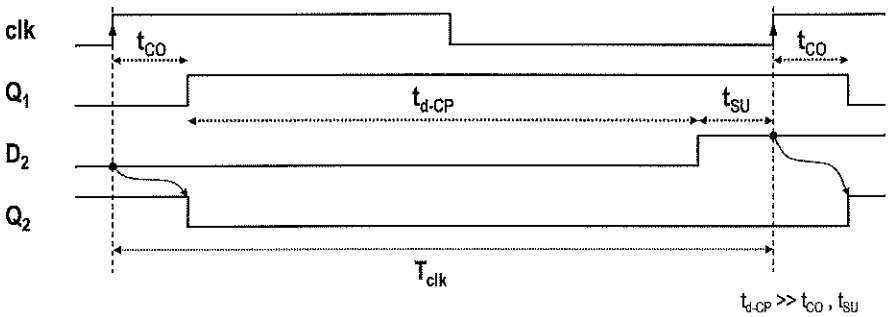
1. Maximum Clock Rate for a Synchronous Design

1.1 Critical Path



1.2 Maximum Clock Rate





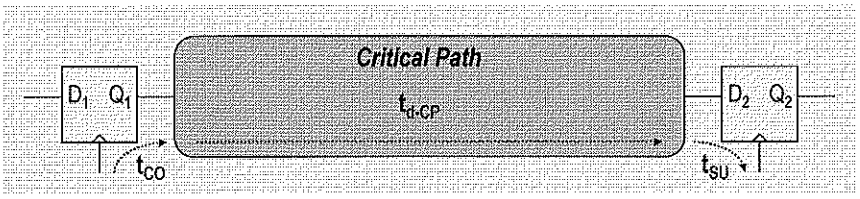
$T_{clk} > t_{CO} + t_{d\text{-}CP} + t_{SU}$

$f_{clk} < \frac{1}{t_{CO} + t_{d\text{-}CP} + t_{SU}}$

$t_{d\text{-}CP} >> t_{CO}, t_{SU}$

$f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$

1.3 Timing Definitions



latency

$T_L = T_{clk} \approx t_{d\text{-}CP}$

The **delay** from when an input is applied
until the output associated with that input becomes available.
[seconds] or [clock cycles]

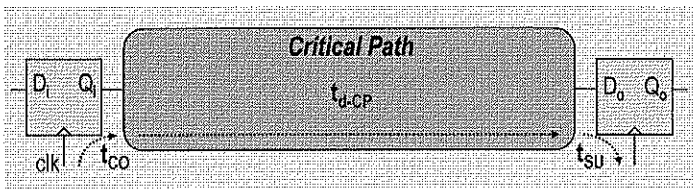
throughput

$f_T = f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$

The **rate** at which inputs are processed to create outputs.
[operators/second]

2. Pipelining

1.1 Pipelining – Optimal Case



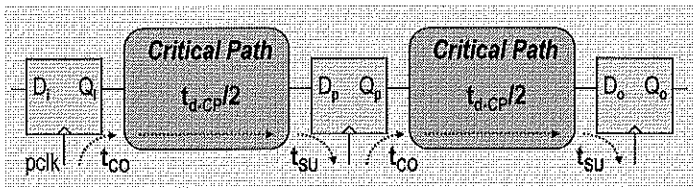
latency

$T_L = T_{clk} \approx t_{d\text{-}CP}$

throughput

$f_T = f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$

pipelining



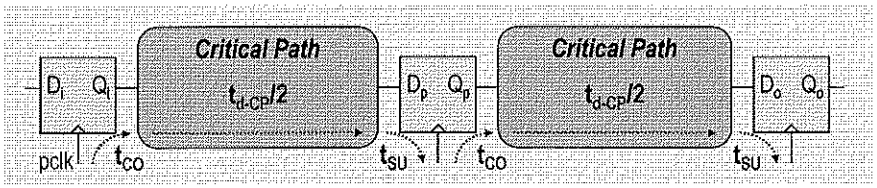
latency

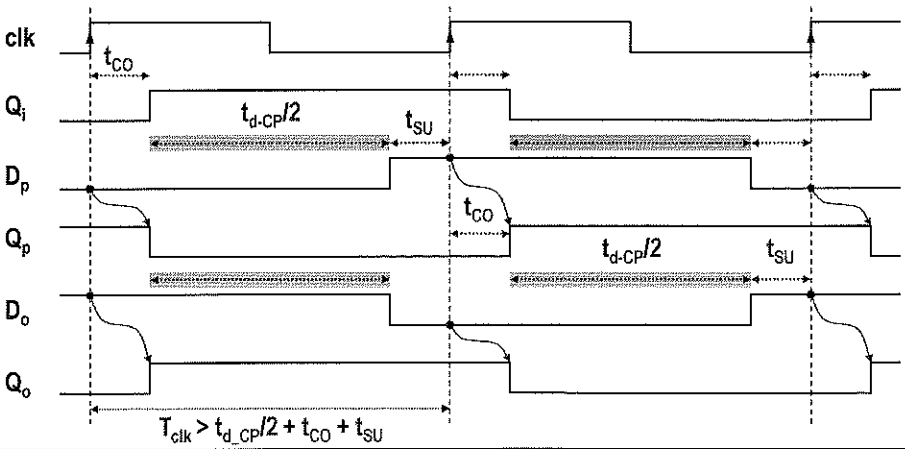
$T_L = 2.T_{pclk} \approx t_{d\text{-}CP}$

throughput

$f_T = f_{pclk} \approx 2.f_{clk} \approx \frac{2}{t_{d\text{-}CP}}$

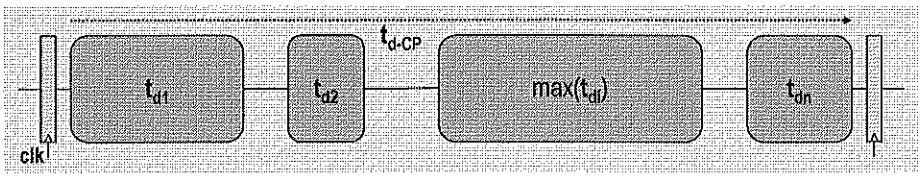
2 pipeline stages



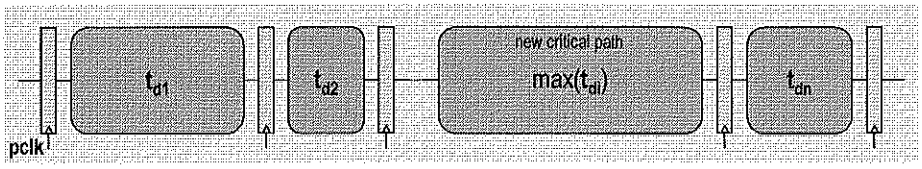


1.2 Pipelining – General Case

latency $T_L = T_{clk} \approx t_{d\text{-}CP} = \Sigma\, t_{di}$ *throughput* $f_T = f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$

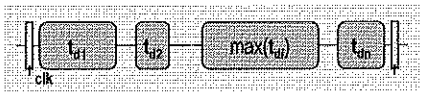
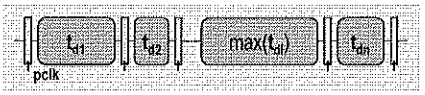


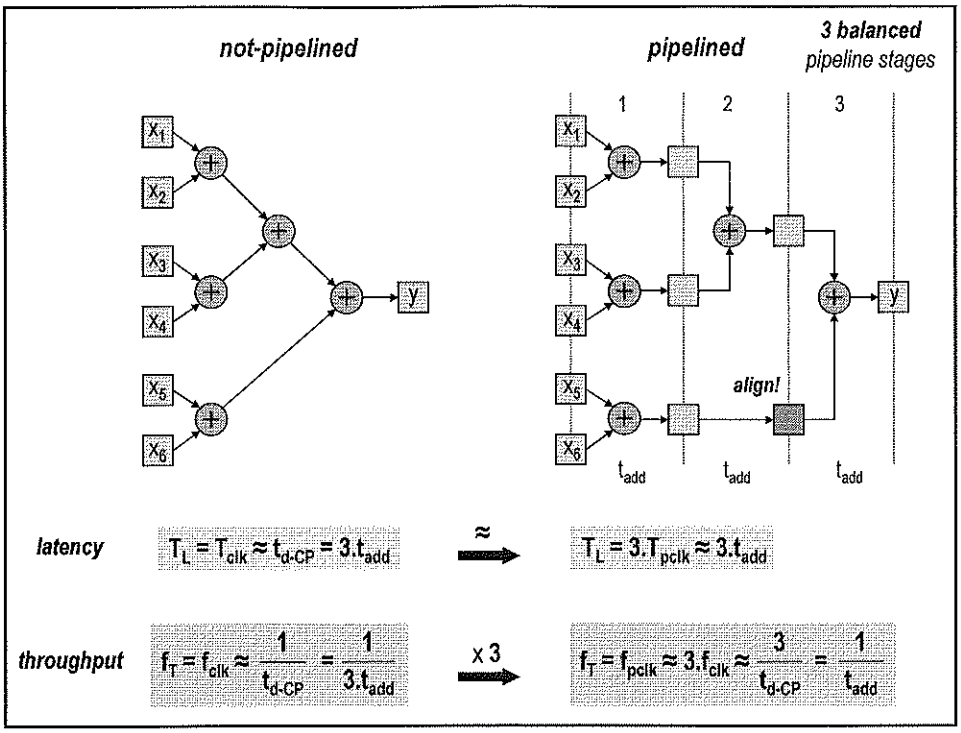
pipelining

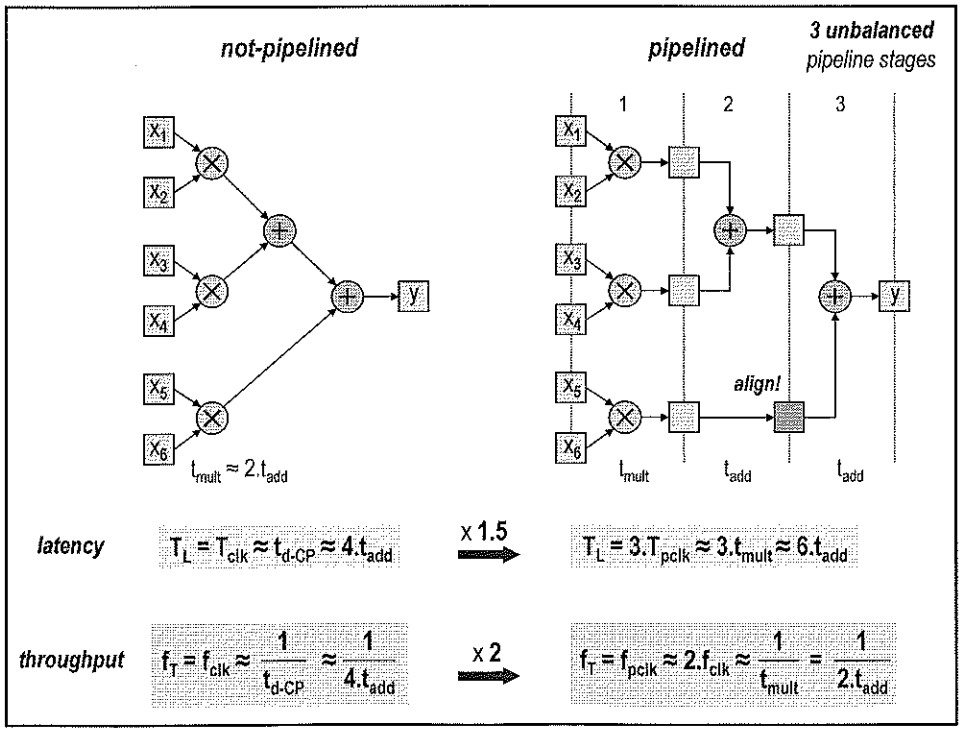


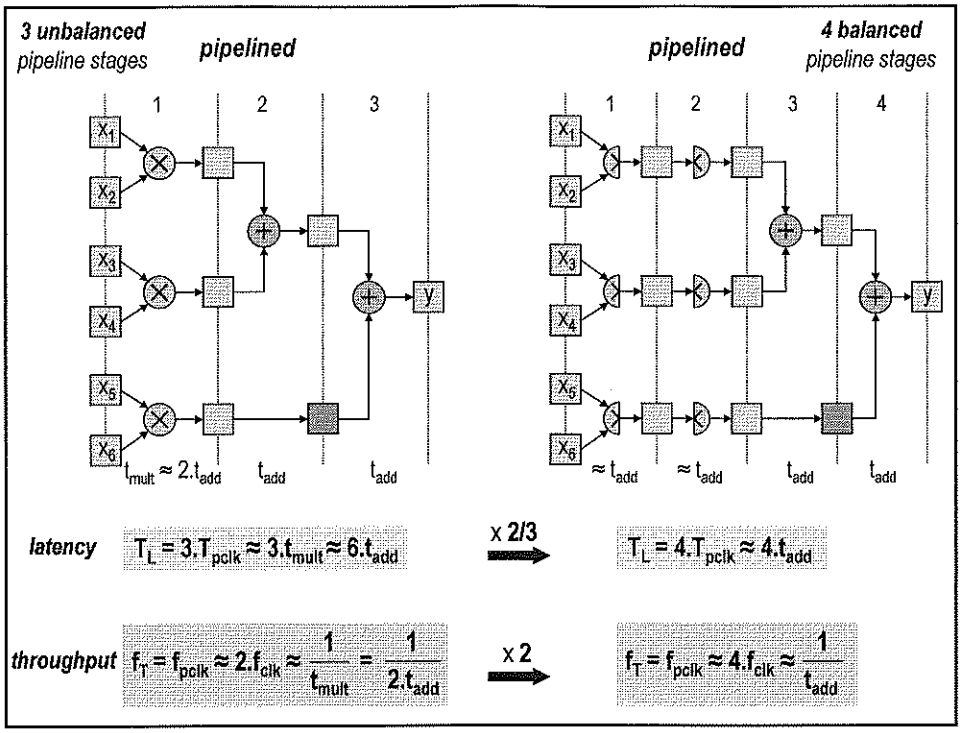
latency $T_L = n.T_{pclk} \approx n.[\max(t_{di})]$ *throughput* $f_T = f_{pclk} \approx \frac{1}{\max(t_{di})}$

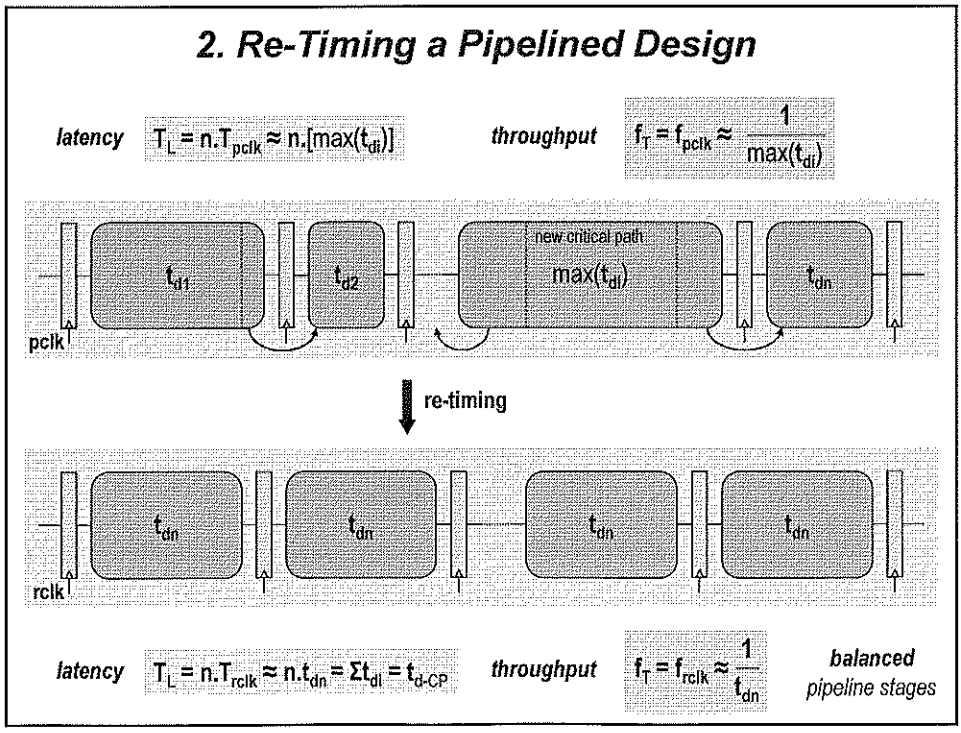
Pipelining Results

	not-pipelined	*pipelined*
		
latency	$T_L = T_{clk} \approx t_{d\text{-}CP} = \Sigma\, t_{di}$	$T_L = n.T_{pclk} \approx n.[\max(t_{di})]$
		Latency in cycles increases (x n) Latency in elapsed (real) time increases.
throughput	$f_T = f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$	$f_T = f_{pclk} \approx \frac{1}{\max(t_{di})}$
		Critical paths reduced. Clock frequency increases. Throughput increases.
area		Area increases: more flip-flops required.

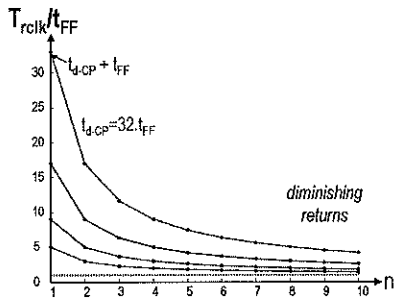

not-pipelined
pipelined
3 balanced
pipeline stages
1
2
3
x1
x2
x3
x4
x5
x6
y
align!
tadd
tadd
tadd
latency
TL = Tclk ≈ td-CP = 3.tadd
≈
TL = 3.Tpclk ≈ 3.tadd
throughput
fT = fclk ≈ 1 / td-CP = 1 / 3.tadd
x 3
fT = fpclk ≈ 3.fclk ≈ 3 / td-CP = 1 / tadd

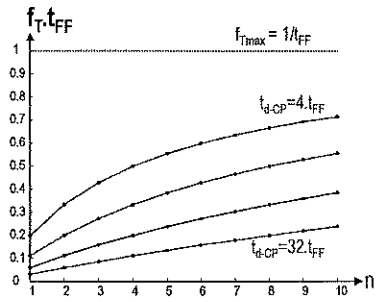

not-pipelined
pipelined
3 unbalanced
pipeline stages
1
2
3
x1
x2
x3
x4
x5
x6
y
align!
tmult ≈ 2.tadd
tmult
tadd
tadd
latency
TL = Tclk ≈ td-CP ≈ 4.tadd
x 1.5
TL = 3.Tpclk ≈ 3.tmult ≈ 6.tadd
throughput
fT = fclk ≈ 1 / td-CP ≈ 1 / 4.tadd
x 2
fT = fpclk ≈ 2.fclk ≈ 1 / tmult = 1 / 2.tadd


3 unbalanced pipeline stages
pipelined
pipelined
4 balanced pipeline stages
1
2
3
4
x1
x2
x3
x4
x5
x6
y
tmult ≈ 2.tadd
tadd
tadd
≈ tadd
≈ tadd
tadd
tadd
latency
TL = 3.Tpclk ≈ 3.tmult ≈ 6.tadd
x 2/3
TL = 4.Tpclk ≈ 4.tadd
throughput
fT = fpclk ≈ 2.fclk ≈ 1/tmult = 1/2.tadd
x 2
fT = fpclk ≈ 4.fclk ≈ 1/tadd

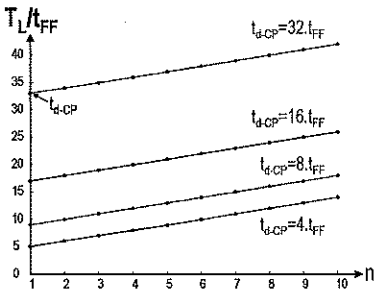

2. Re-Timing a Pipelined Design
latency
TL = n.Tpclk ≈ n.[max(tdi)]
throughput
fT = fpclk ≈ 1/max(tdi)
new critical path
max(tdi)
td1
td2
tdn
pclk
re-timing
rclk
latency
TL = n.Trclk ≈ n.tdn = Σtdi = td-CP
throughput
fT = frclk ≈ 1/tdn
balanced pipeline stages

Pipelining Limits (Balanced Pipeline Stages)





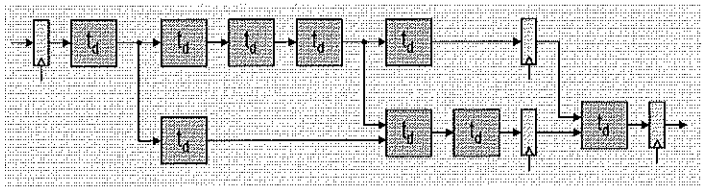
throughput $f_T = f_{rclk} = \frac{1}{t_{dn} + t_{FF}} = \frac{1}{t_{d\text{-}CP}/n + t_{FF}}$



latency $T_L = n.T_{rclk} = n.(t_{dn} + t_{FF})$

$T_L = t_{d\text{-}CP} + n.t_{FF}$

$t_{FF} = t_{SU} + t_{CO}$

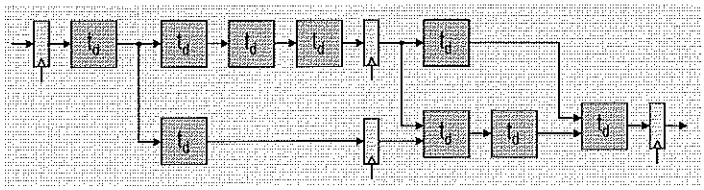


$T_{pclk} \approx 6.t_d$

$T_L = 2.T_{pclk} \approx 12.t_d$

$f_T \approx \frac{1}{6t_{di}}$

re-timing

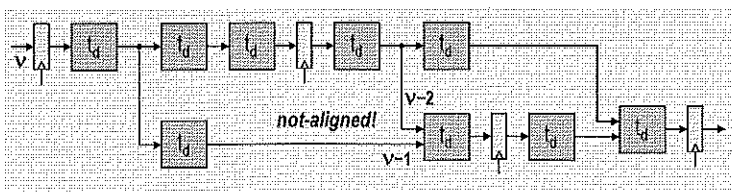


$T_{pclk} \approx 4.t_d$

$T_L = 2.T_{pclk} \approx 8.t_d$

$f_T \approx \frac{1}{4t_{di}}$

re-timing?

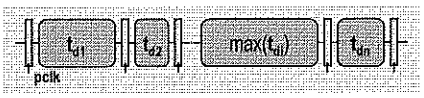
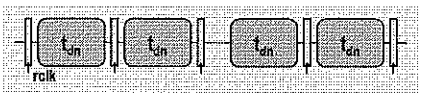


$T_{pclk} \approx 3.t_d$

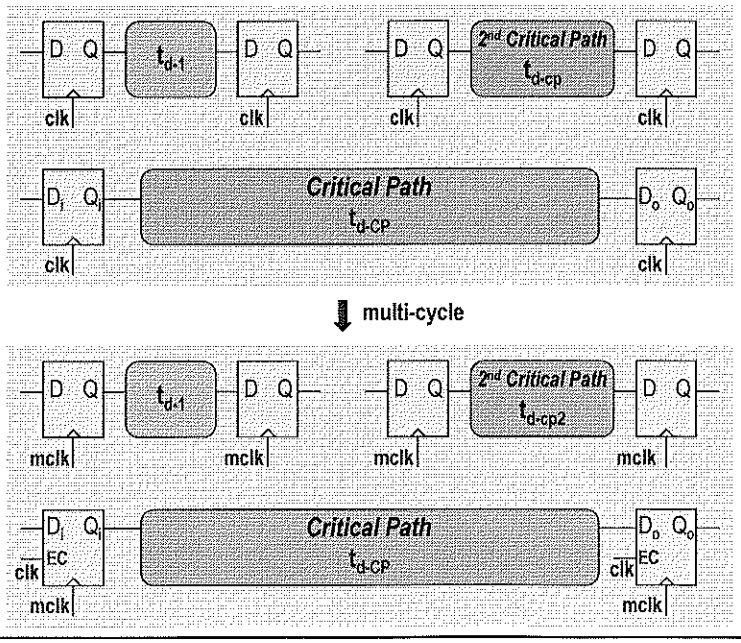
$T_L = 2.T_{pclk} \approx 6.t_d$

$f_T \approx \frac{1}{3t_{di}}$

Re-Timing Results

	not re-timed	*re-timed*
		
latency	$T_L = n.T_{pclk} \approx n.[\max(t_{di})]$	$T_L = n.T_{rclk} \approx n.t_{dn}$
		Latency in cycles constant (**n**) Latency in elapsed (real) time decreases, because the cycle time decreases.
throughput	$f_T = f_{pclk} \approx \frac{1}{\max(t_{di})}$	$f_T = f_{rclk} \approx \frac{1}{t_{dn}}$
		Critical paths reduced. Clock frequency increases. Throughput increases.
area		Area: depends on retimed path.

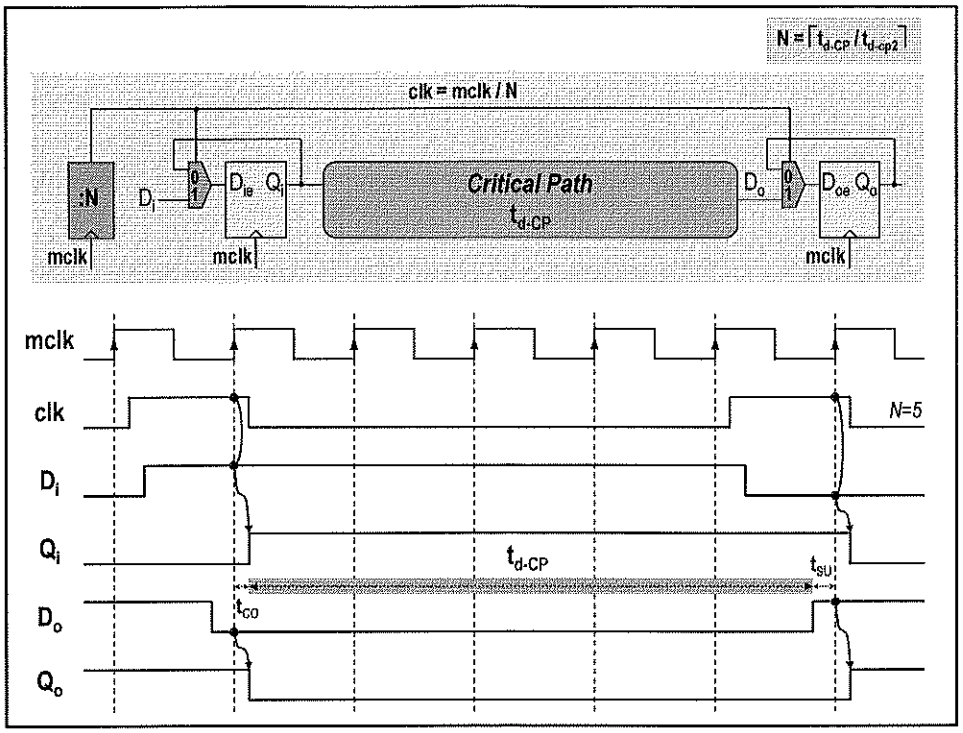
3. Multi-Cycle

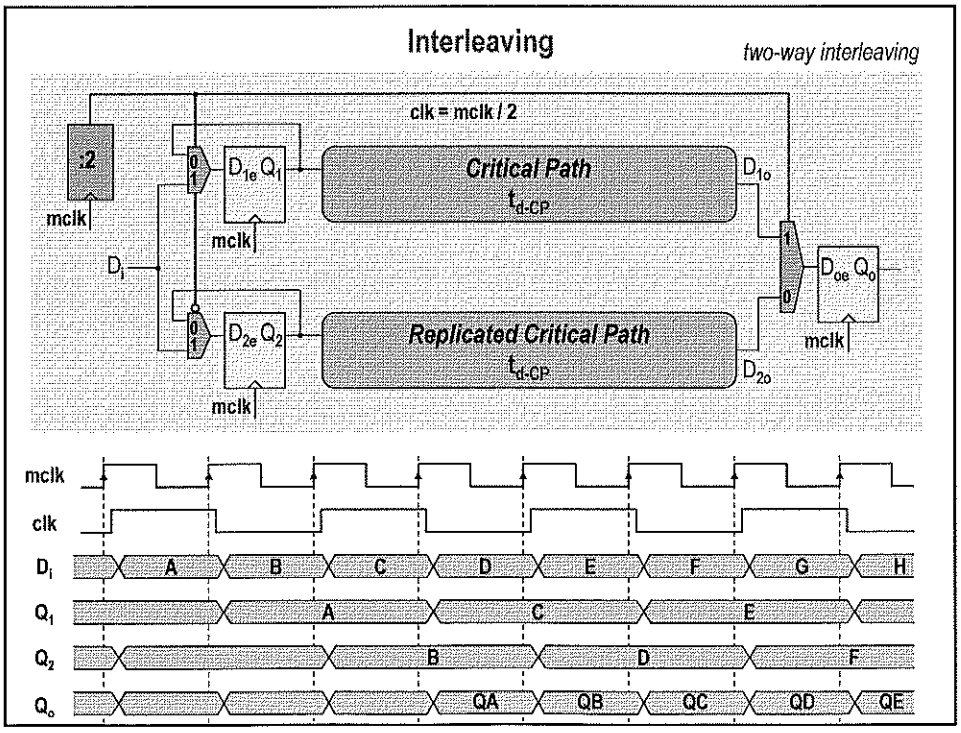


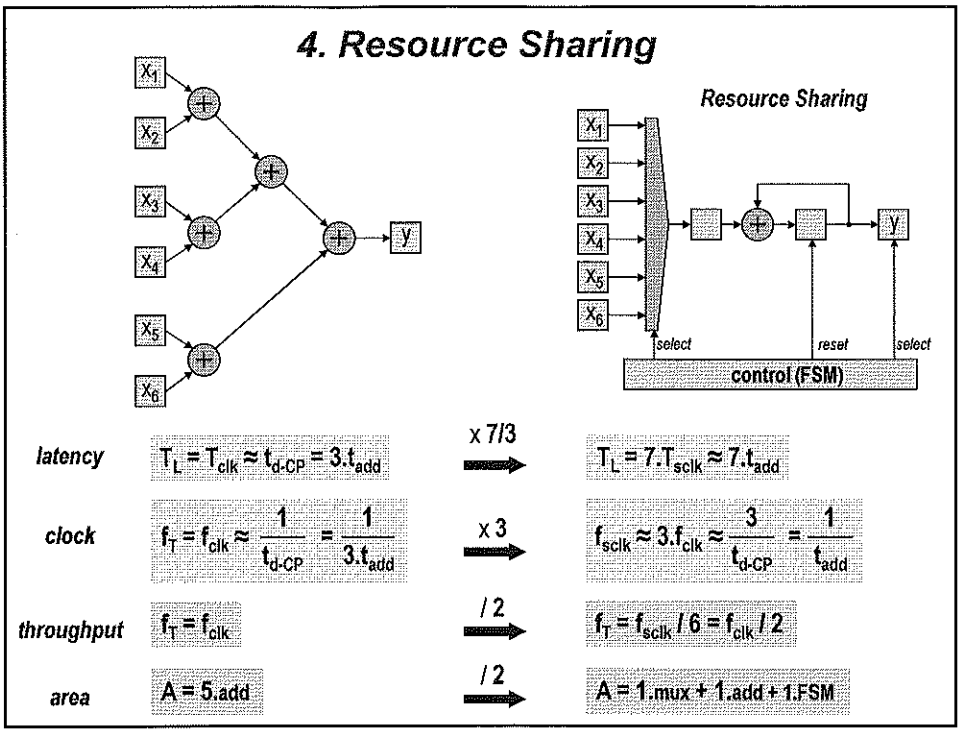
$$f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$$

$$f_{mclk} \approx \frac{1}{t_{d\text{-}cp2}}$$

$$f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$$


N = ⌈ t_d-CP / t_d-cp2 ⌉
clk = mclk / N
:N
mclk
D_i
D_ie Q_i
Critical Path
t_d-CP
D_o
D_oe Q_o
mclk
mclk
mclk
clk
N=5
D_i
Q_i
t_d-CP
t_SU
t_CO
D_o
Q_o


Interleaving
two-way interleaving
clk = mclk / 2
:2
mclk
D_i
D_1e Q_1
Critical Path
t_d-CP
D_1o
D_oe Q_o
mclk
mclk
D_2e Q_2
Replicated Critical Path
t_d-CP
D_2o
mclk
mclk
clk
D_i
A B C D E F G H
Q_1
A C E
Q_2
B D F
Q_o
QA QB QC QD QE


4. Resource Sharing
Resource Sharing
x1
x2
x3
x4
x5
x6
y
select
reset
select
control (FSM)
latency
$T_L = T_{clk} \approx t_{d\text{-}CP} = 3.t_{add}$
x 7/3
$T_L = 7.T_{sclk} \approx 7.t_{add}$
clock
$f_T = f_{clk} \approx \frac{1}{t_{d\text{-}CP}} = \frac{1}{3.t_{add}}$
x 3
$f_{sclk} \approx 3.f_{clk} \approx \frac{3}{t_{d\text{-}CP}} = \frac{1}{t_{add}}$
throughput
$f_T = f_{clk}$
/ 2
$f_T = f_{sclk} / 6 = f_{clk} / 2$
area
A = 5.add
/ 2
A = 1.mux + 1.add + 1.FSM

HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

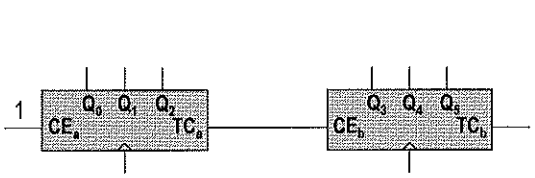
Digitale Synthese
High Speed Counters

EmSD
Embedded System Design

ir. J. Meel
oct 2009

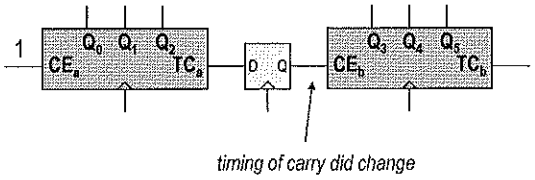
1. Pipelined Carry

1.1 Free-Running Counter



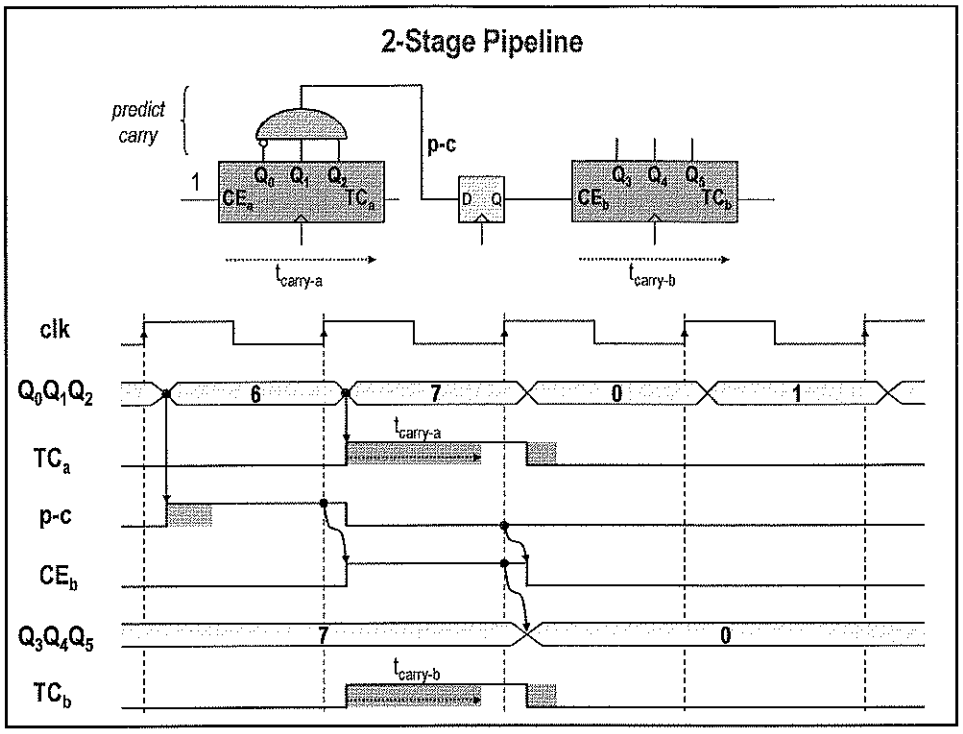
$Q_0Q_1Q_2$	$Q_3Q_4Q_5$
111	000
000	100
100	100
111	100
000	010
100	010

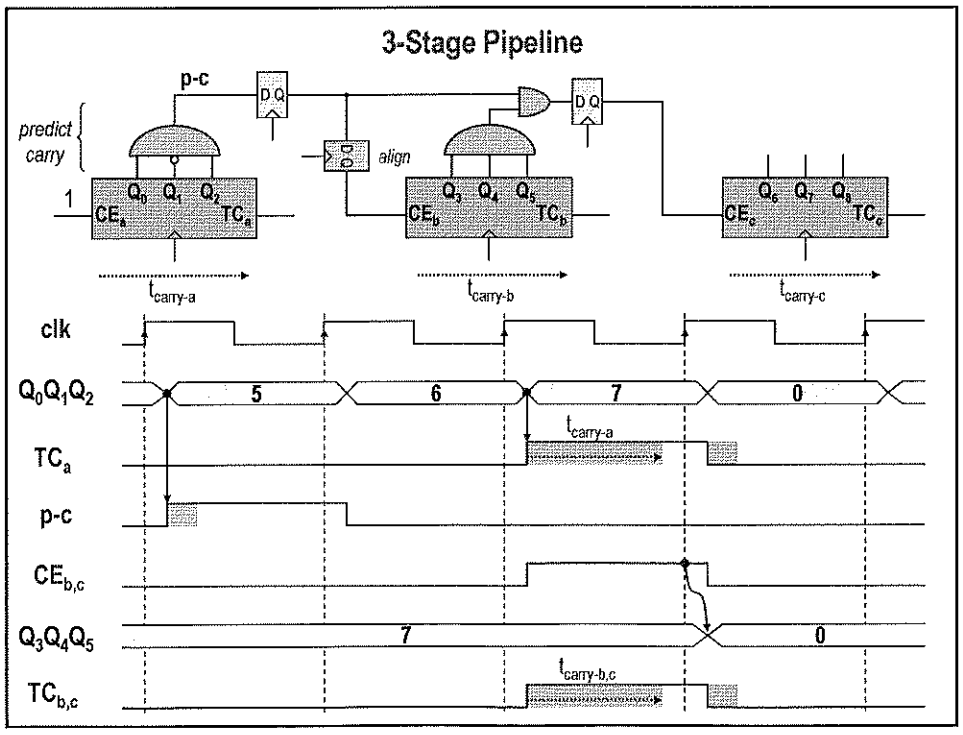
pipelining

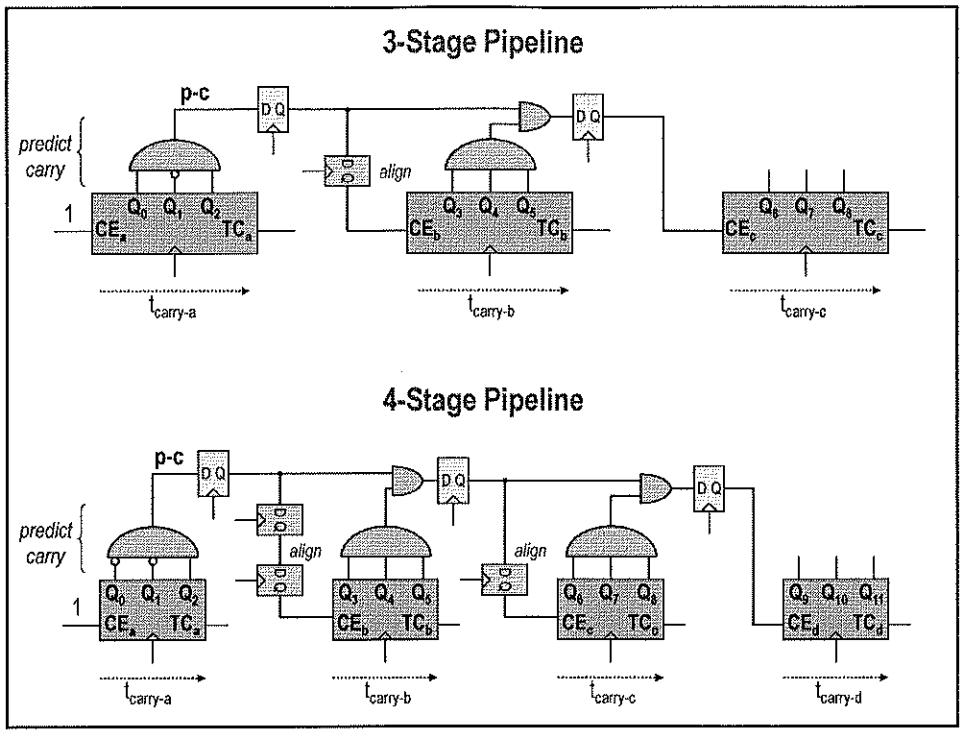


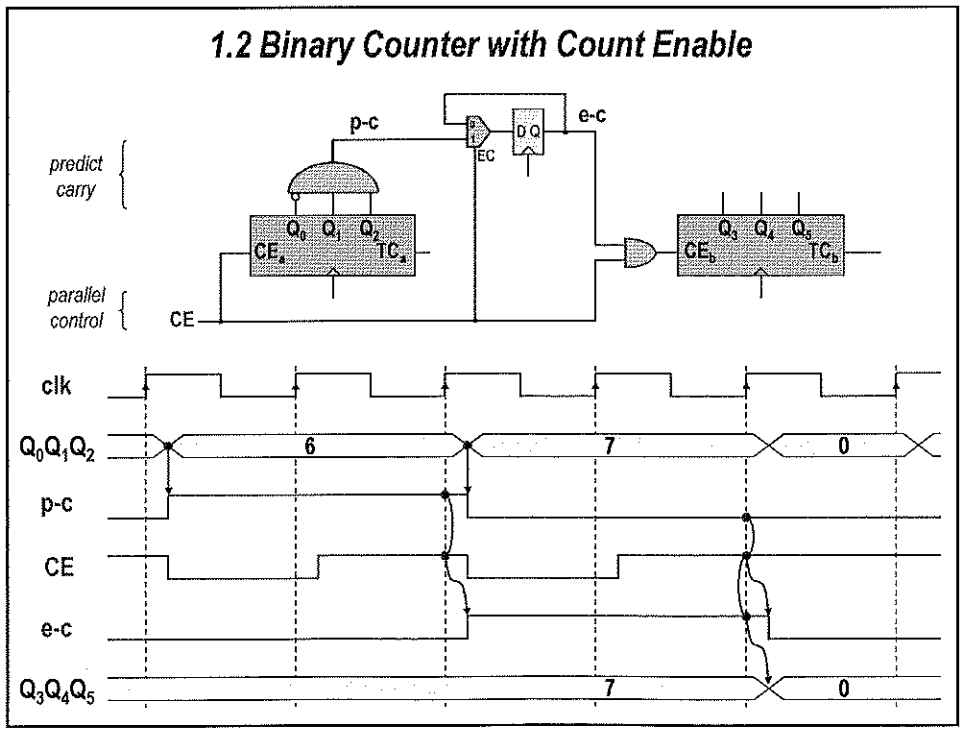
timing of carry did change

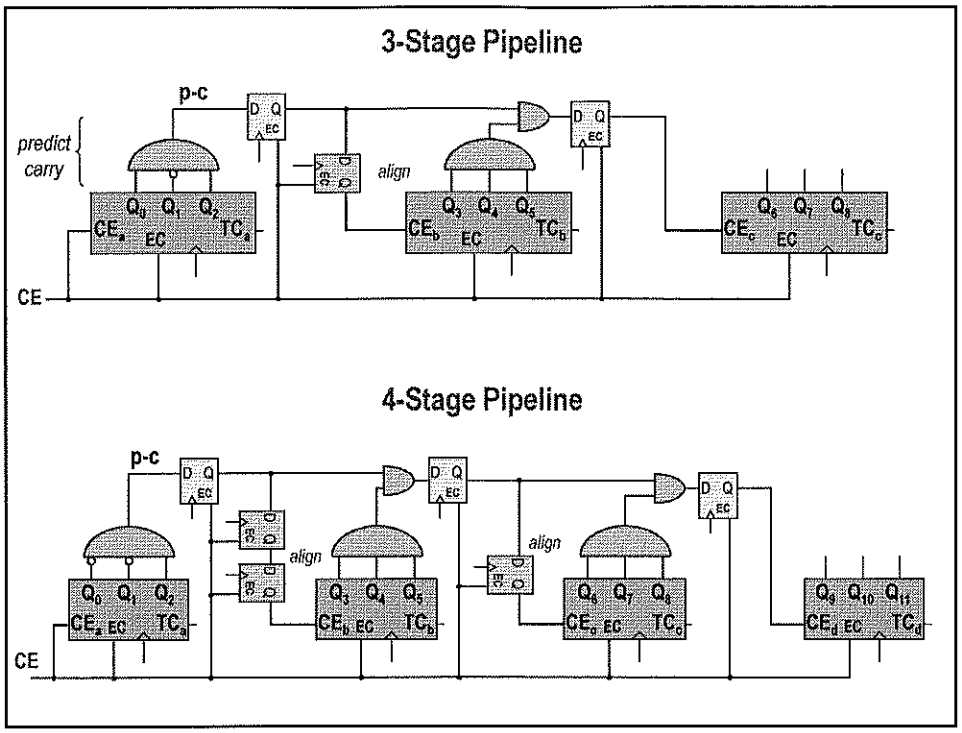
$Q_0Q_1Q_2$	$Q_3Q_4Q_5$	
111	000	
000	000	
100	100	*wrong state transition (carry too late)*
111	100	
000	100	
100	010	


2-Stage Pipeline
predict carry
1
p-c
D Q
clk
$Q_0Q_1Q_2$
6
7
0
1
$t_{carry-a}$
TC_a
p-c
CE_b
$Q_3Q_4Q_5$
7
0
$t_{carry-b}$
TC_b


3-Stage Pipeline
p-c
D Q
predict carry
1
align
$t_{carry-a}$
$t_{carry-b}$
$t_{carry-c}$
clk
$Q_0Q_1Q_2$
5
6
7
0
$t_{carry-a}$
TC_a
p-c
$CE_{b,c}$
$Q_3Q_4Q_5$
7
0
$t_{carry-b,c}$
$TC_{b,c}$

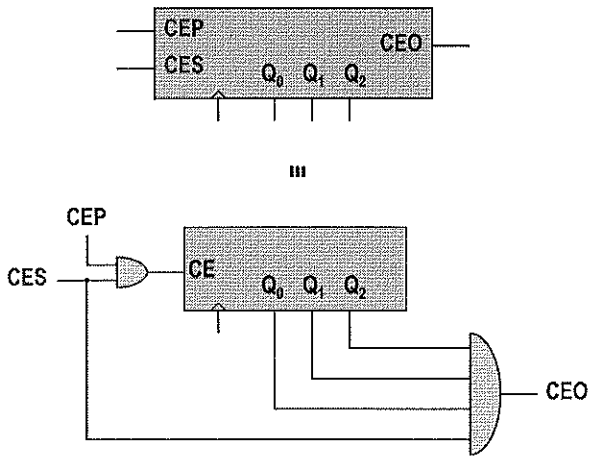

3-Stage Pipeline
p-c
predict carry
1
align
CE_a TC_a
CE_b TC_b
CE_c TC_c
$t_{carry-a}$
$t_{carry-b}$
$t_{carry-c}$
4-Stage Pipeline
p-c
predict carry
1
align
align
CE_a TC_a
CE_b TC_b
CE_c TC_c
CE_d TC_d
$t_{carry-a}$
$t_{carry-b}$
$t_{carry-c}$
$t_{carry-d}$


1.2 Binary Counter with Count Enable
p-c
e-c
EC
predict carry
parallel control
CE
CE_a TC_a
CE_b TC_b
clk
$Q_0Q_1Q_2$
6
7
0
p-c
CE
e-c
$Q_3Q_4Q_5$
7
0



2. Prescaler Technique (Multi-Cycle)

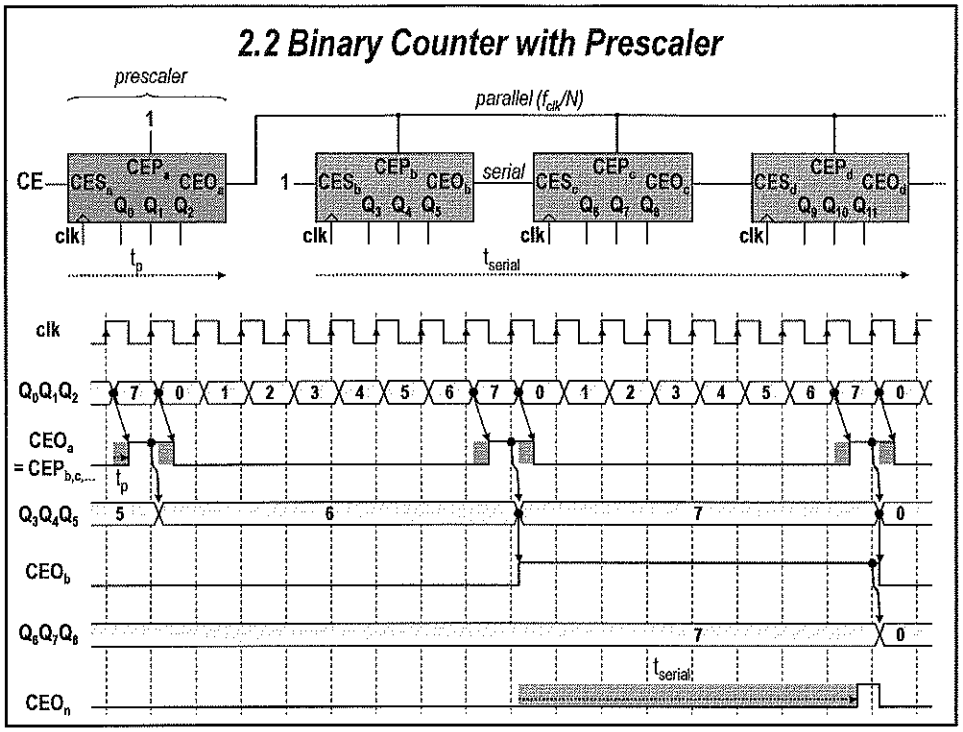
2.1 Counter Module (Serial and Parallel Count Enable)



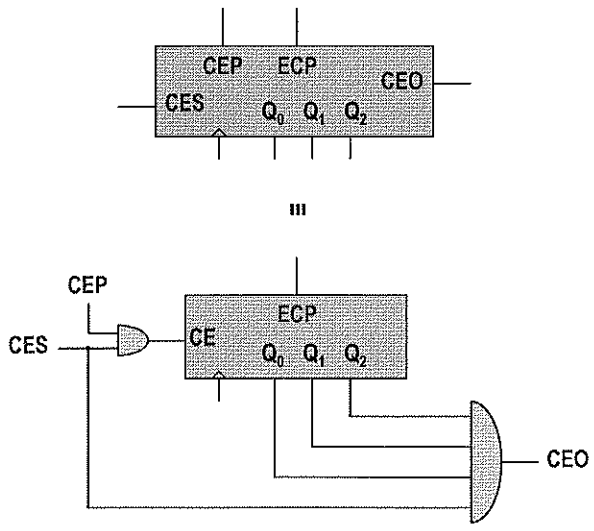
CEP = Count Enable Parallel
CES = Count Enable Serial

CEO = Count Enable Out

2.2 Binary Counter with Prescaler



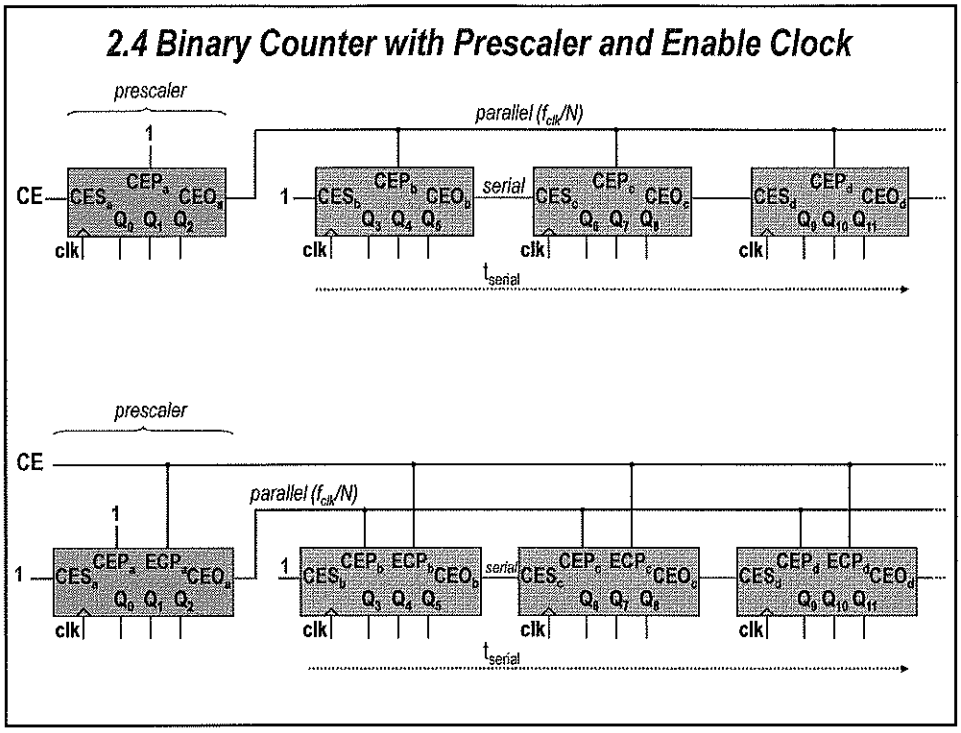
2.3 Counter Module (Count Enable & Parallel Enable Clock)



ECP = Enable Clock Parallel
CEP = Count Enable Parallel
CES = Count Enable Serial

CEO = Count Enable Out

2.4 Binary Counter with Prescaler and Enable Clock



HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

Digitale Synthese
FSM: Input/Output

EmSD
Embedded System Design

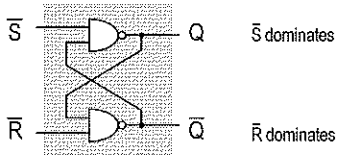
ir. J. Meel
jan 2009

1. FSM Input Conditioning

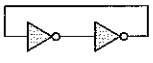
1.1 Debouncing

SR-latch (cross-coupled NAND gates)

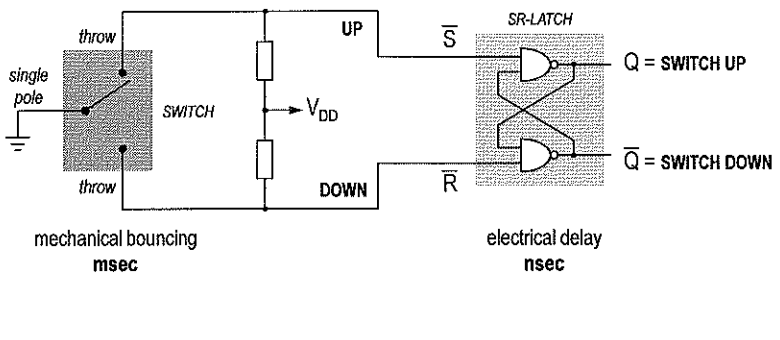
asynchronous circuit

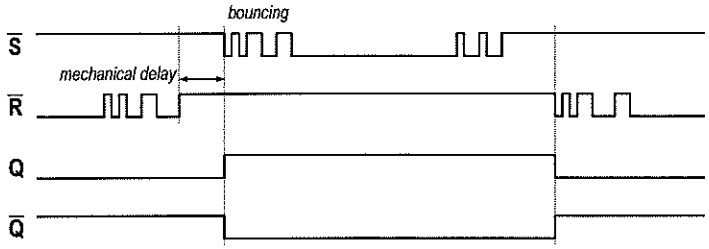


$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	*state*
0	0	1	1	*ambiguous*
0	1	1	0	*SET*
1	0	0	1	*RESET*
1	1	Q	$\overline{Q}$	*memory*

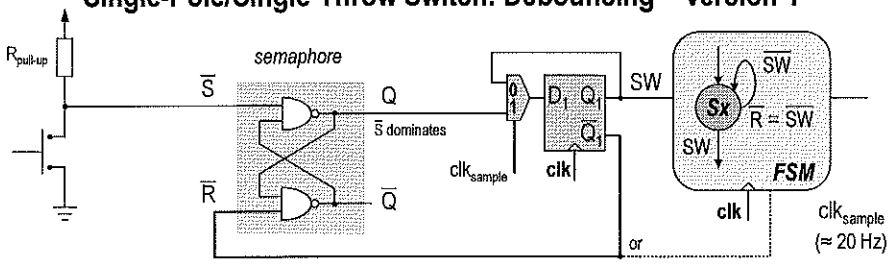


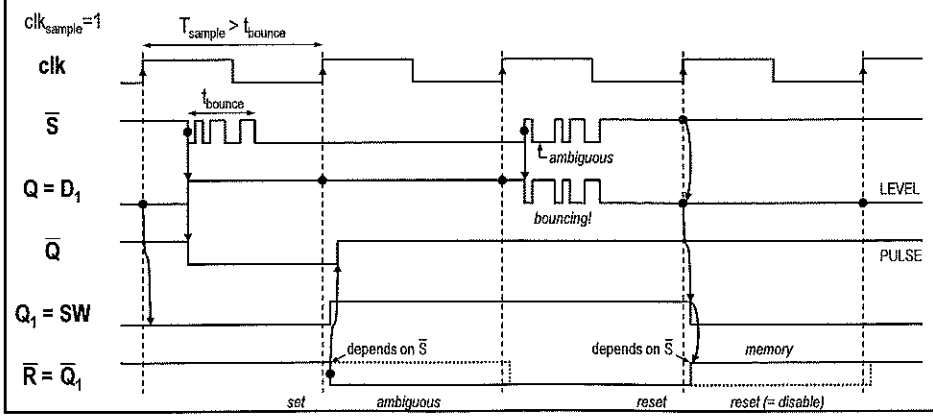
Single-Pole/Double-Throw Switch Debouncing



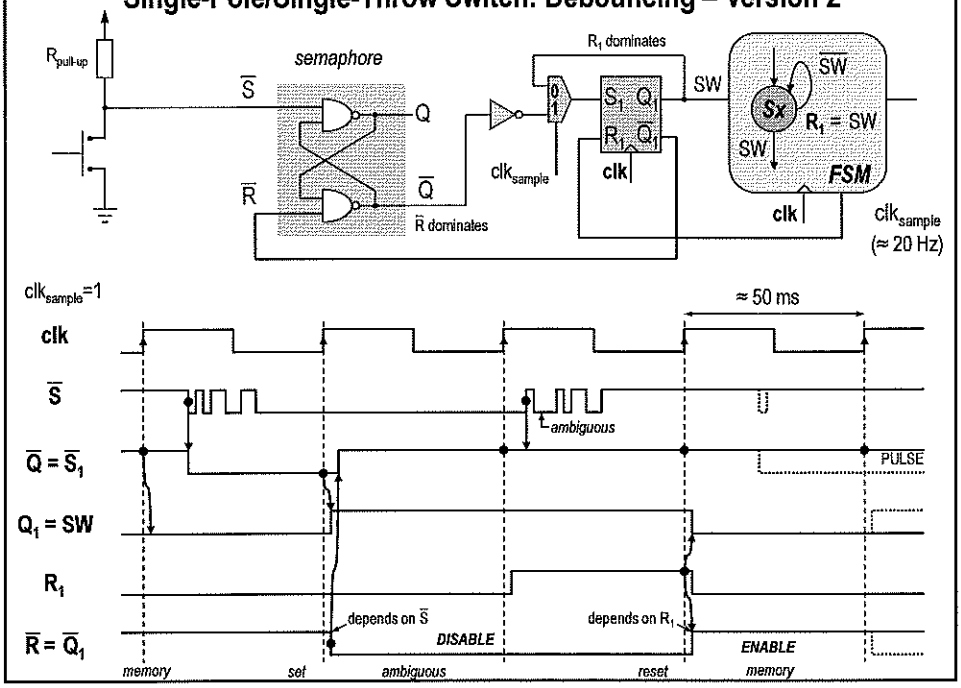


Single-Pole/Single-Throw Switch: Debouncing – Version 1

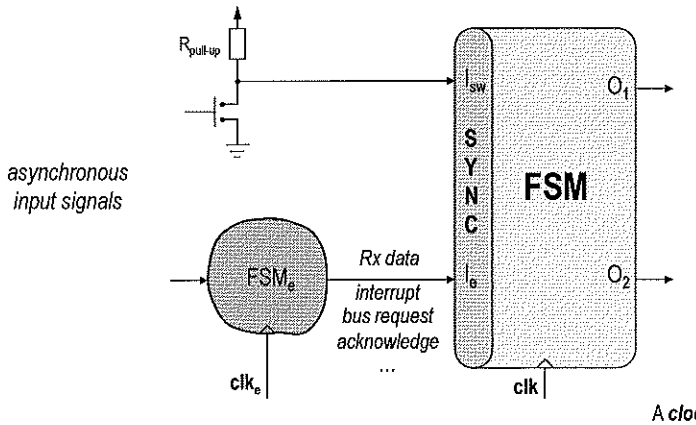




Single-Pole/Single-Throw Switch: Debouncing – Version 2



1.2 Synchronization of Asynchronous Input Signals



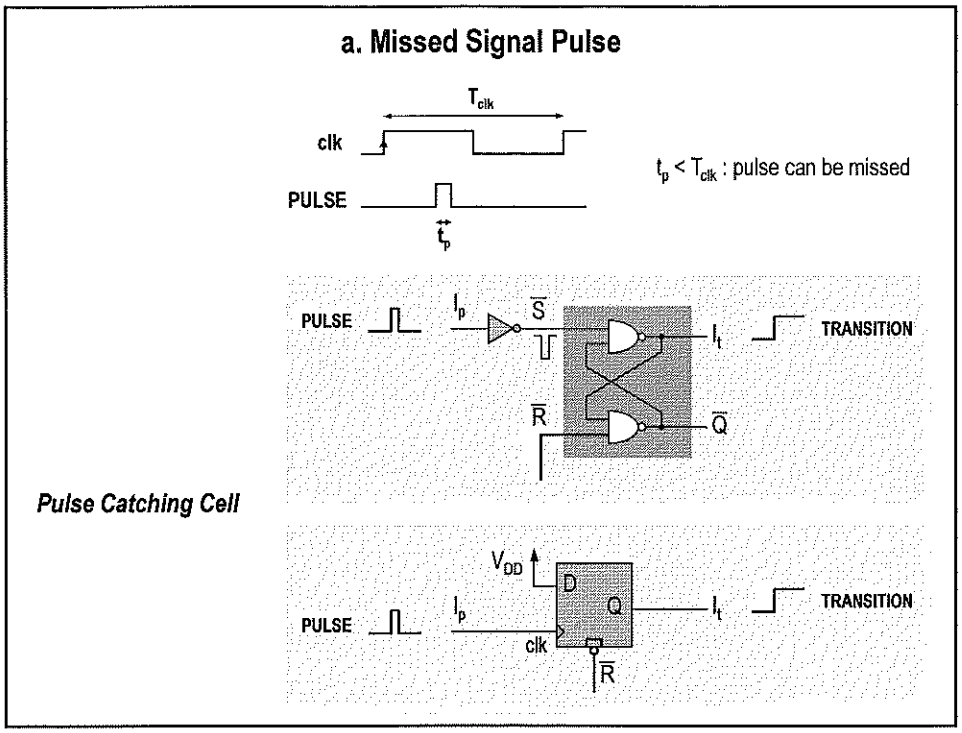
A ***clock domain*** is a group of logic elements and related signals that are synchronized to one clock.

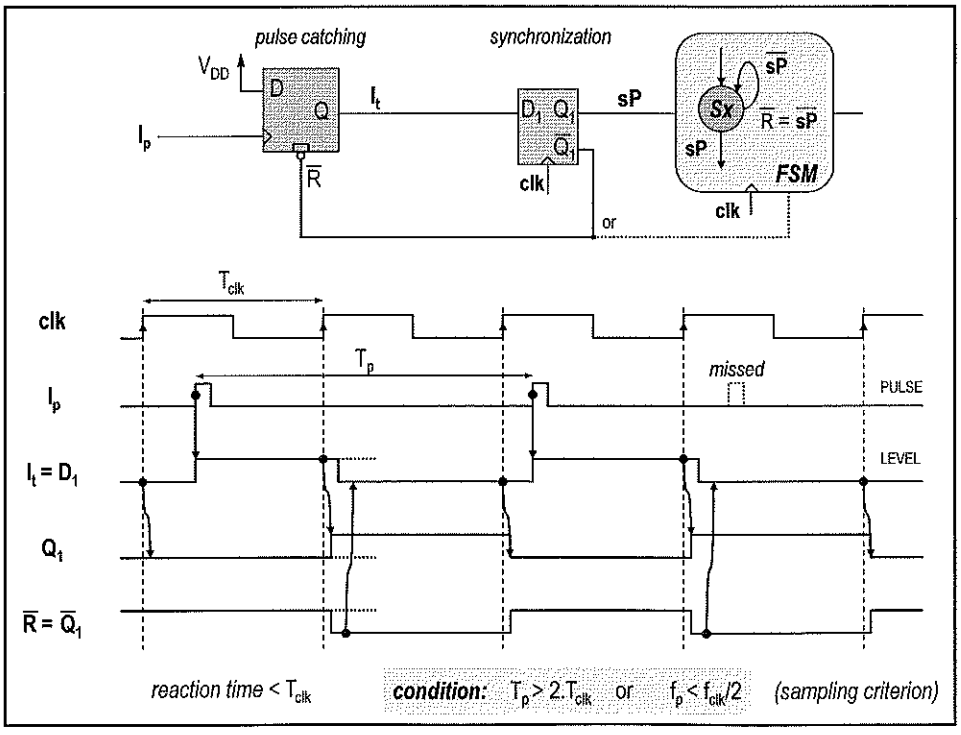
Synchronization errors:

a. Missed signal pulse

b. Wrong state transition (n FF's)

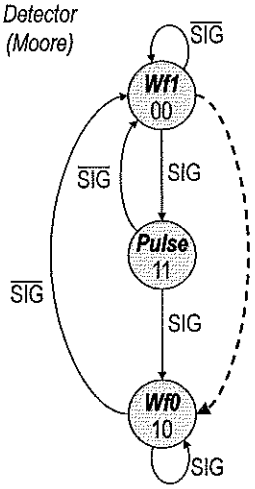
c. Undefined state transition (1 FF - metastability)


a. Missed Signal Pulse
T_{clk}
clk
PULSE
t_p
$t_p < T_{clk}$: pulse can be missed
PULSE
I_p
$\overline{S}$
I_t
TRANSITION
$\overline{R}$
$\overline{Q}$
Pulse Catching Cell
PULSE
V_{DD}
D
Q
I_p
clk
$\overline{R}$
I_t
TRANSITION

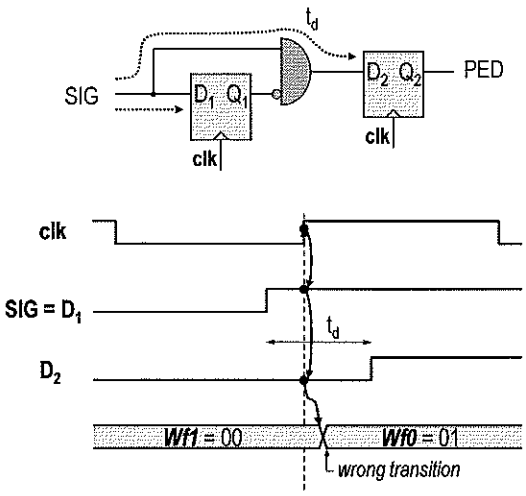

pulse catching
synchronization
V_{DD}
D
Q
I_t
I_p
$\overline{R}$
D_1 Q_1
Q_1
clk
sP
$\overline{sP}$
Sx
$\overline{R} = \overline{sP}$
sP
FSM
clk
or
T_{clk}
clk
T_p
missed
I_p
PULSE
$I_t = D_1$
LEVEL
Q_1
$\overline{R} = \overline{Q_1}$
reaction time < T_{clk}
condition: $T_p > 2.T_{clk}$ or $f_p < f_{clk}/2$ (sampling criterion)

b. Wrong State Transition (n FF's)

Positive Edge Detector (Moore)



<state> = [Q_1 Q_2]

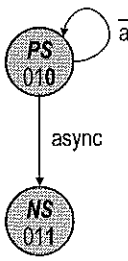


cause: More than 1 FF has to change state to realize a state transition controlled by an asynchronous input.

FSM rules to prevent wrong state transitions

cause: synchronous inputs controlling state transitions

1. Only *1 asynchronous input* can control a conditional state transition.
2. Only *1 state variable* (=flip-flop) can change during a conditional state transition.
   Present state and next state are logically adjacent (unit-distance rule).
3. Only *1 next state.*
   Only 1 flip-flop can change, independent of the level of the input.
4. The *active level* of the asynchronous signal must last at least 2 *clock periods*.



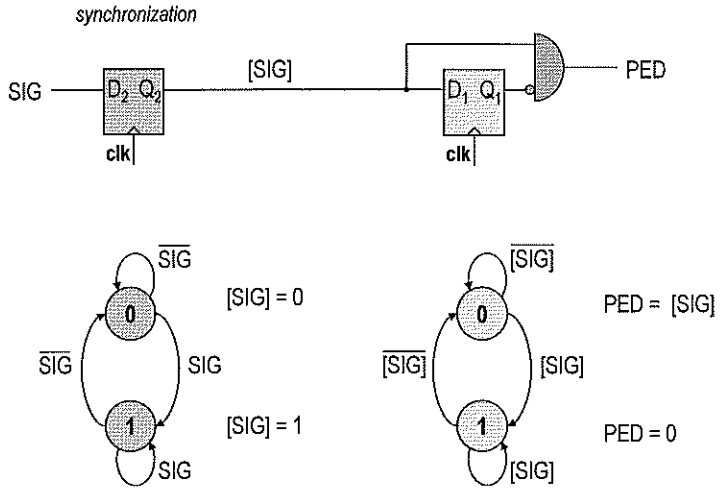
Go/NoGo statement

→ *1 asynchronous input,* $T_{async} > 2.T_{clk}$

→ *1 next state*

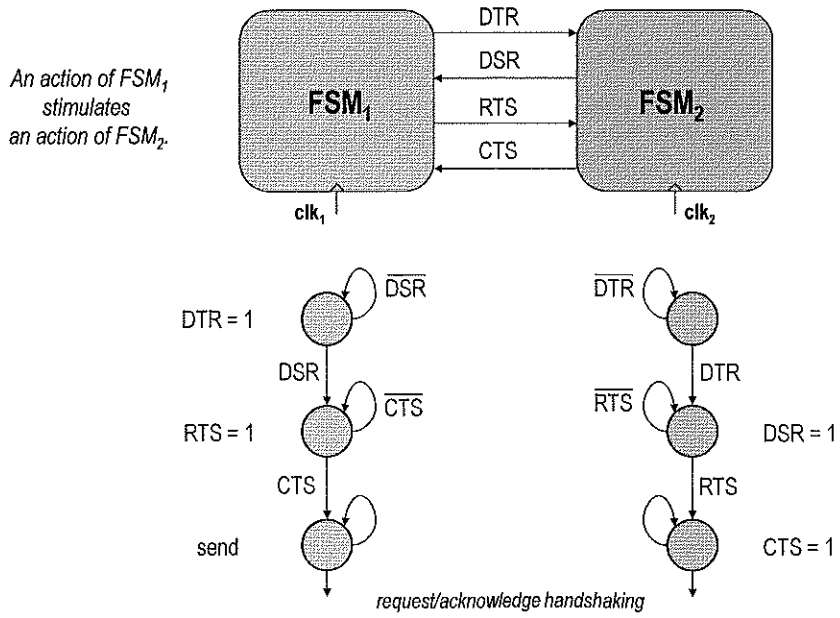
1 state variable changes

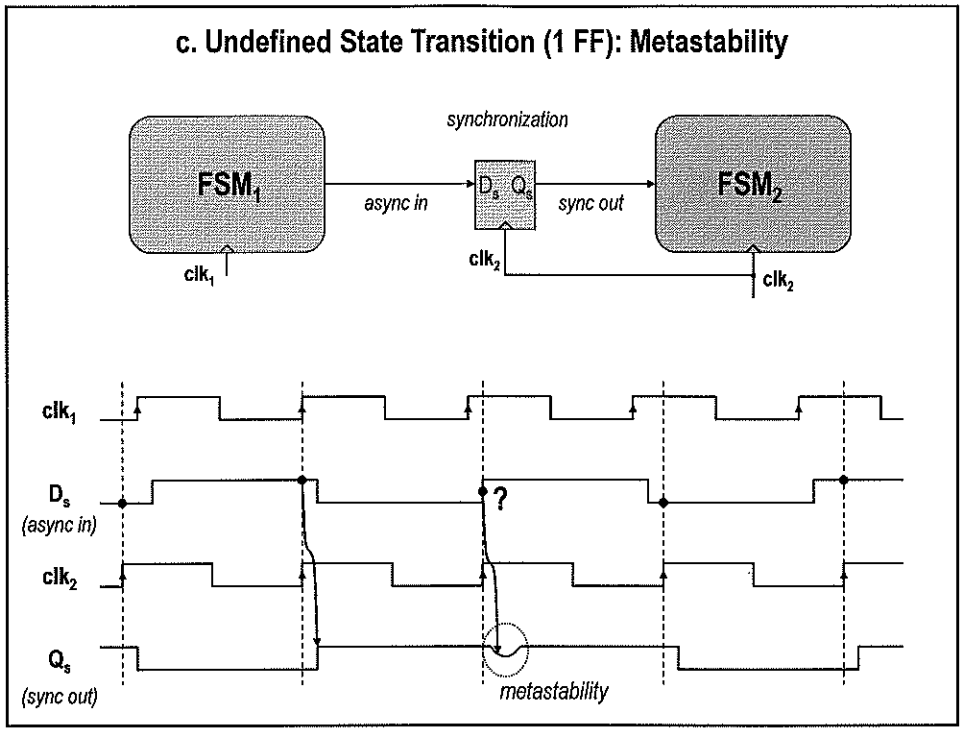
Synchronization flip-flop to prevent wrong state transitions

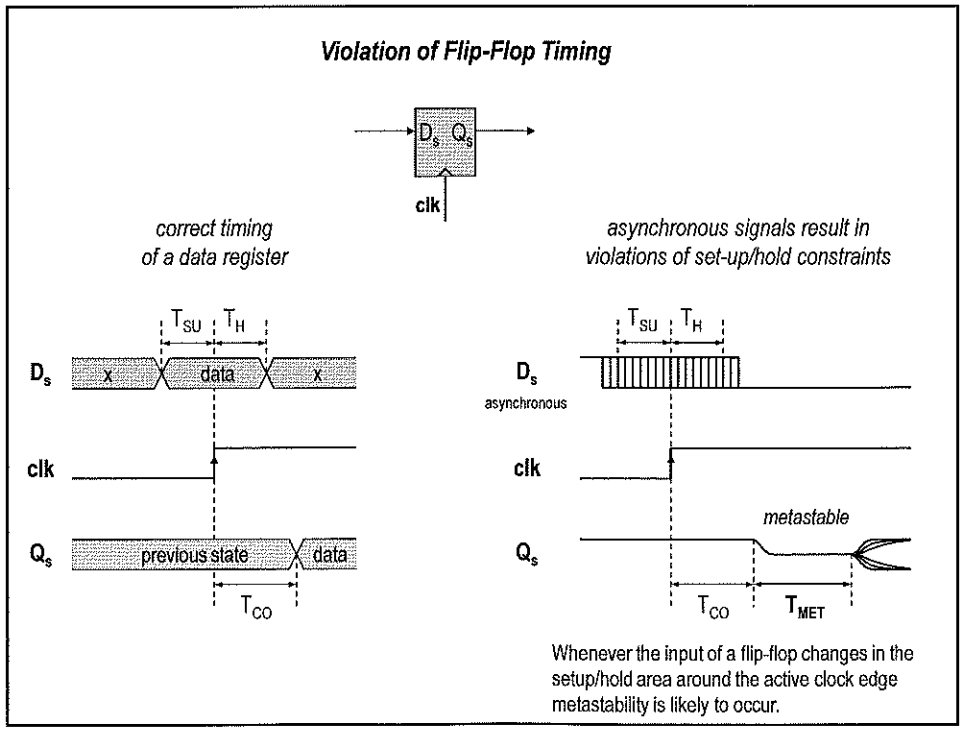


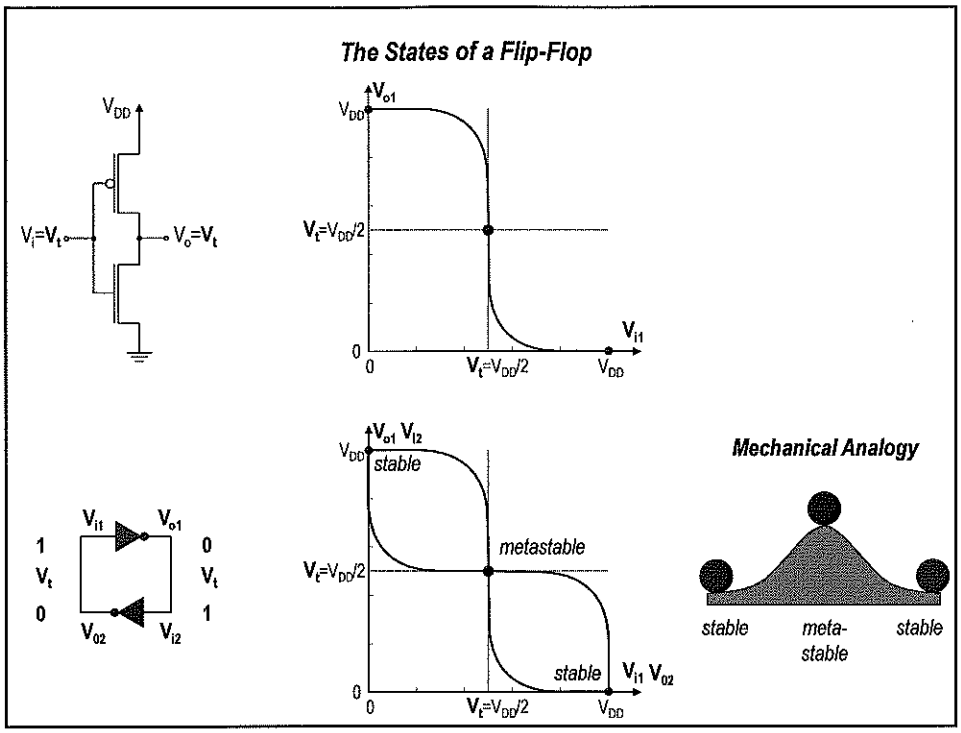
*The influence of the asynchronous signal SIG on the FSM is **isolated** by the synchronization FF.*

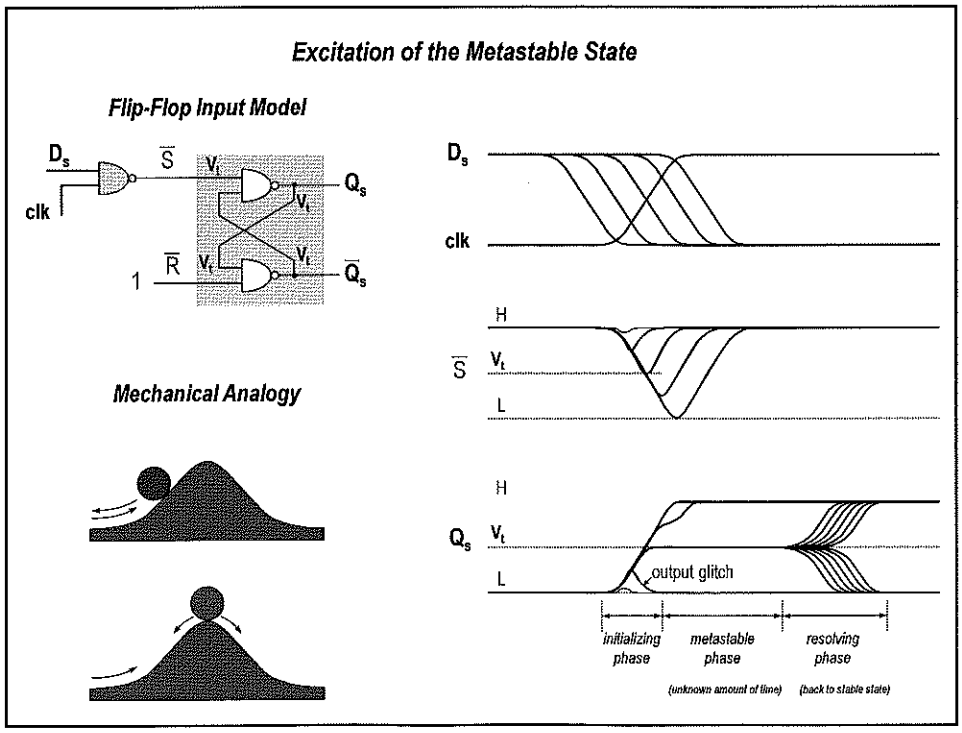
Go/NoGo example: Handshake interface based on asynchronous signals



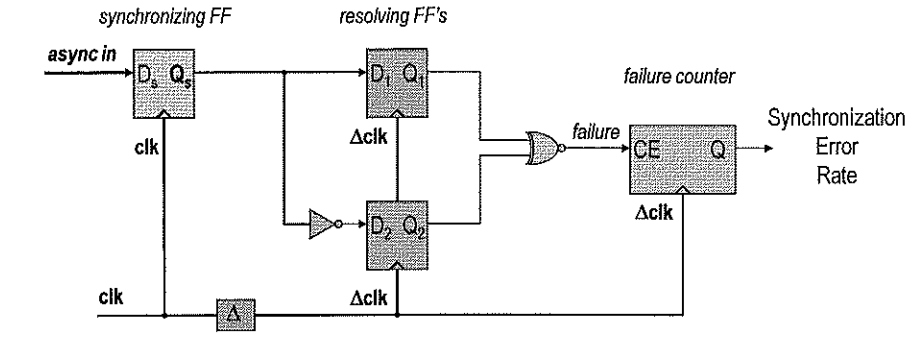

c. Undefined State Transition (1 FF): Metastability
synchronization
FSM1
async in
Ds Qs
sync out
FSM2
clk1
clk2
clk2
clk1
Ds
(async in)
clk2
Qs
(sync out)
?
metastability

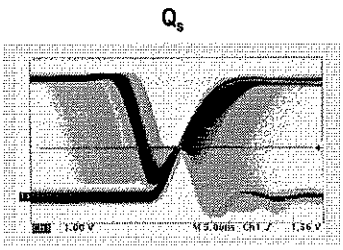

Violation of Flip-Flop Timing
Ds Qs
clk
correct timing
of a data register
asynchronous signals result in
violations of set-up/hold constraints
TSU TH
Ds
x
data
x
clk
Qs
previous state
data
TCO
TSU TH
Ds
asynchronous
clk
metastable
Qs
TCO
TMET
Whenever the input of a flip-flop changes in the
setup/hold area around the active clock edge
metastability is likely to occur.

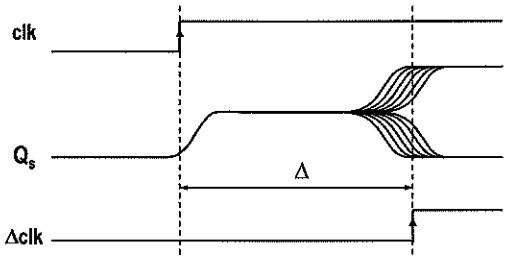

The States of a Flip-Flop
V_{DD}
$V_i=V_t$
$V_o=V_t$
V_{o1}
$V_t=V_{DD}/2$
V_{i1}
0
V_{i1} V_{o1}
1
V_t
0
V_{o2} V_{i2}
stable
metastable
V_{i1} V_{o2}
Mechanical Analogy
stable
meta-
stable
stable


Excitation of the Metastable State
Flip-Flop Input Model
D_s
clk
$\overline{S}$
$\overline{R}$
1
Q_s
$\overline{Q}_s$
Mechanical Analogy
H
V_t
L
output glitch
initializing phase
metastable phase
resolving phase
(unknown amount of time)
(back to stable state)

Metastability Test Circuit







Metastability Measurement Data

$$MTBF = \frac{e^{\frac{\Delta}{\tau}}}{W.f_{clk}.f_{data}}$$

$$\ln(MTBF) = \frac{\Delta}{\tau} - \ln(W.f_{clk}.f_{data})$$

MTBF = Mean Time Between Failures

design parameter

Δ = resolution interval

signal parameters

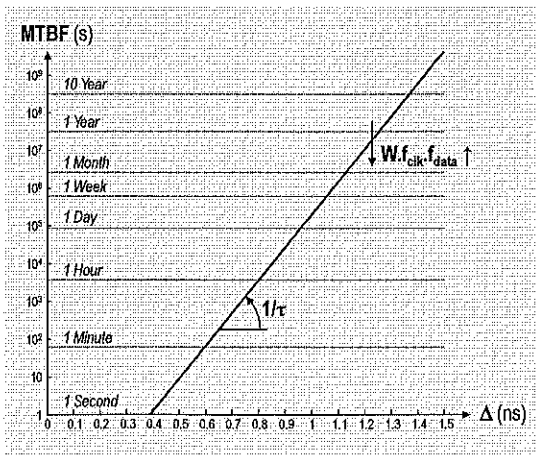
f_{clk} = clock rate

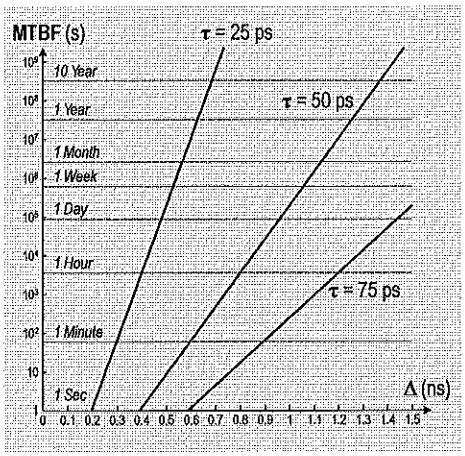
f_{data} = data rate

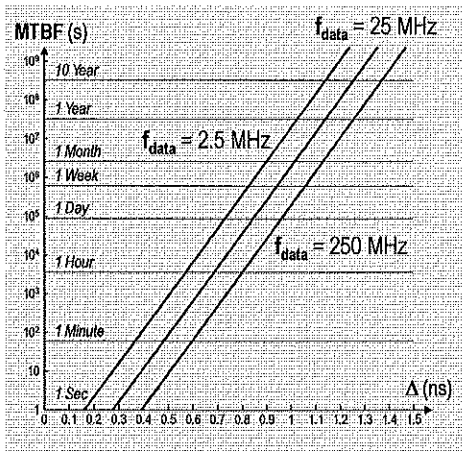
device parameters

$W = T_{SU} + T_H$
= metastability interval

τ = settling time
~ 1 / bandwidth

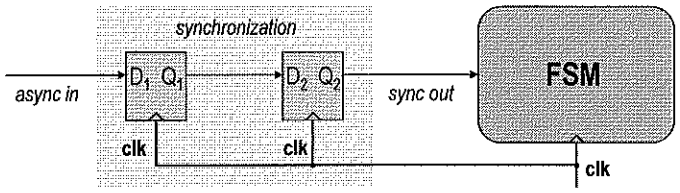




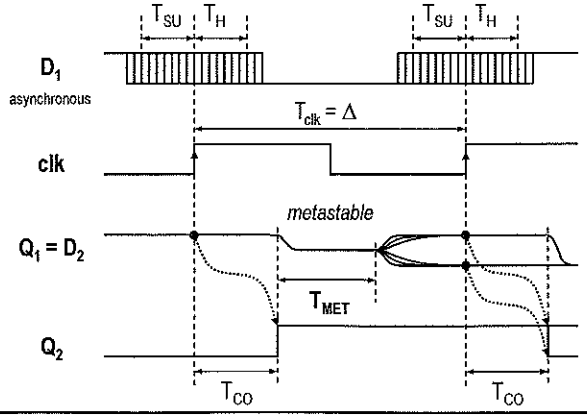


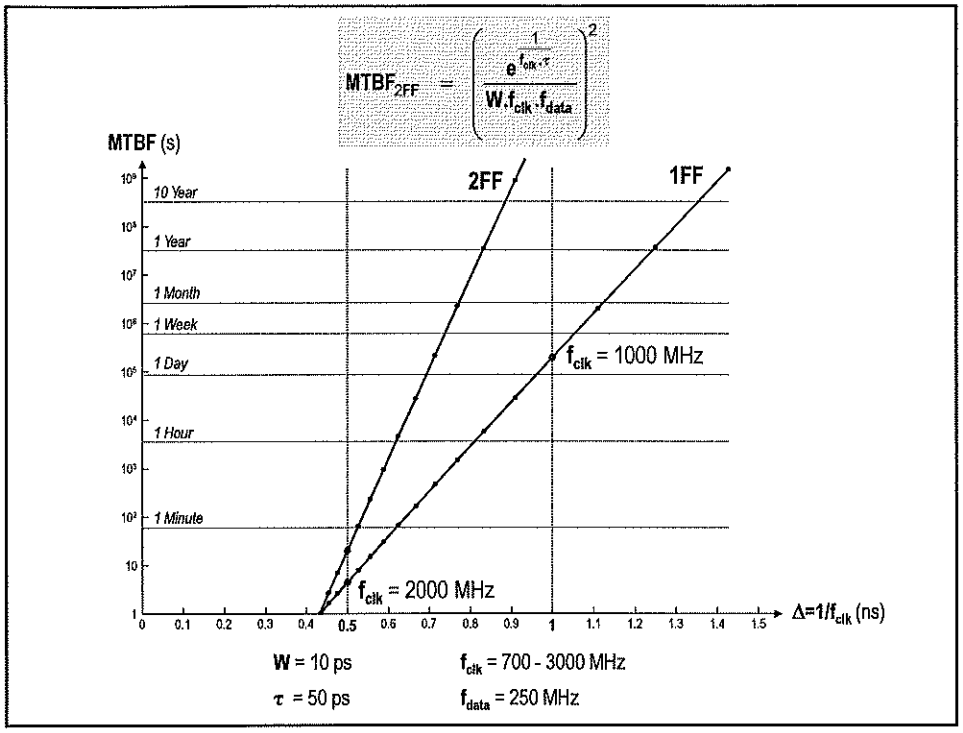
W = 10 ps	f_{clk} = 1000 MHz	**W** = 10 ps	f_{clk} = 1000 MHz
τ = 25 - 75 ps	f_{data} = 250 MHz	τ = 50 ps	f_{data} = 2.5 - 250 MHz

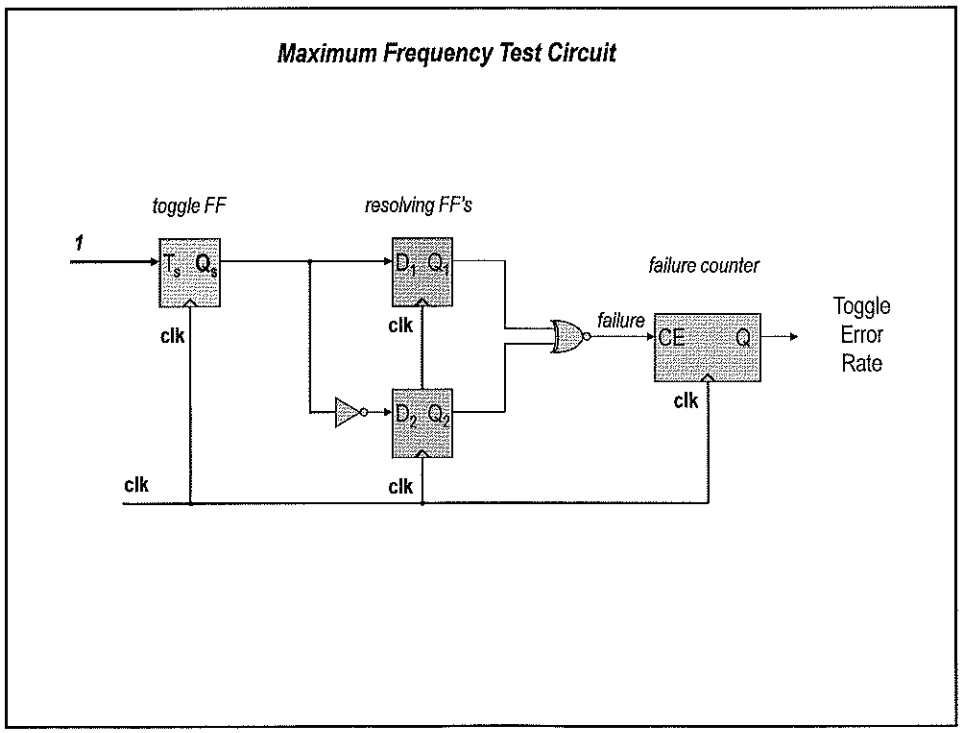
Synchronizer Circuit (Two-Stage Synchronization)



$$MTBF_{2FF} = \left(\frac{e^{\frac{\Delta}{\tau}}}{W.f_{clk}.f_{data}} \right)^2$$



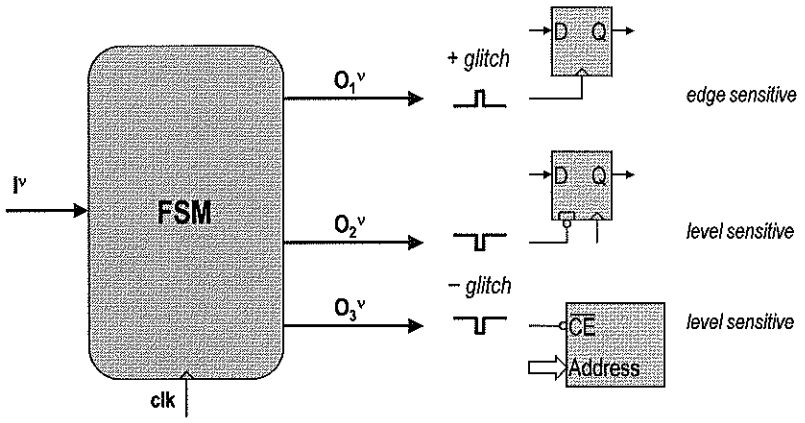

$MTBF_{2FF} = \left(\frac{1}{W.f_{clk}.f_{data}} \right)^2$
MTBF (s)
10 Year
1 Year
1 Month
1 Week
1 Day
1 Hour
1 Minute
2FF
1FF
f_{clk} = 1000 MHz
f_{clk} = 2000 MHz
$\Delta=1/f_{clk}$ (ns)
W = 10 ps
τ = 50 ps
f_{clk} = 700 - 3000 MHz
f_{data} = 250 MHz


Maximum Frequency Test Circuit
toggle FF
resolving FF's
failure counter
1
clk
failure
CE
Q
Toggle
Error
Rate

2. FSM Output Conditioning

2.1 Output Glitches

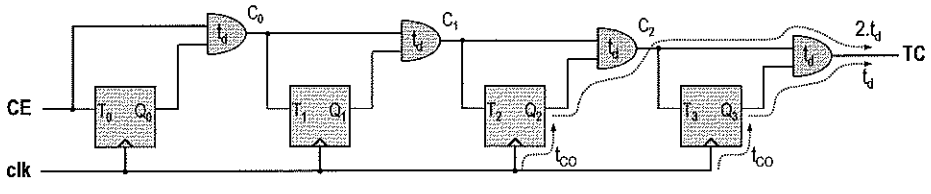
glitch (spike) = uncontrolled and unwanted transient on an output signal

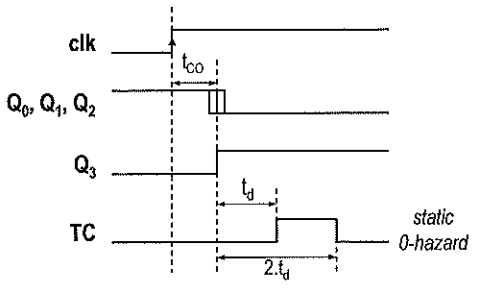


A glitch on an FSM output can result in unwanted transitions in the controlled system.

2.2 Combinatorial Outputs

a. Multi-bit Flashing (Race Condition)



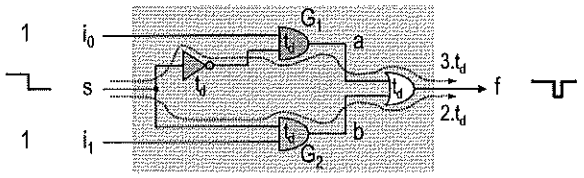


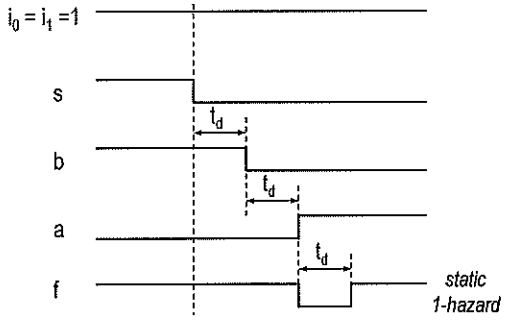
Q_0	Q_1	Q_2	Q_3
1	1	1	0
↓	↓	↓	↓
0	0	0	1

multi-bit flashing

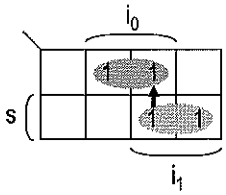
Race condition:
The change in two or more state variables for any state transition can result in an output glitch due to the difference in delay from the state variables to the output.

b. Single-bit Flashing



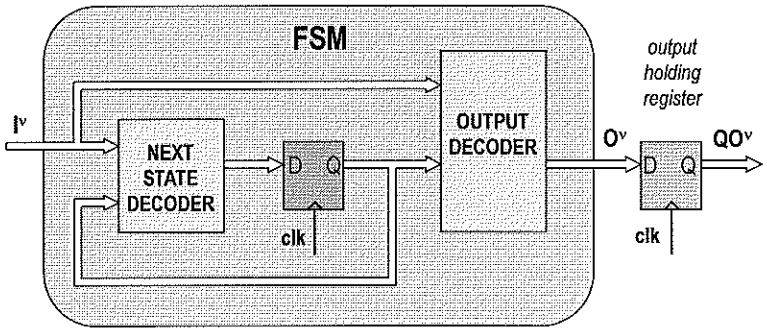


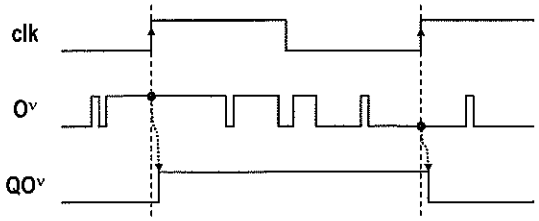
static 1-hazard



The change of a single state variable can result in an output glitch when the change propagates to the output on two different paths with different delays.

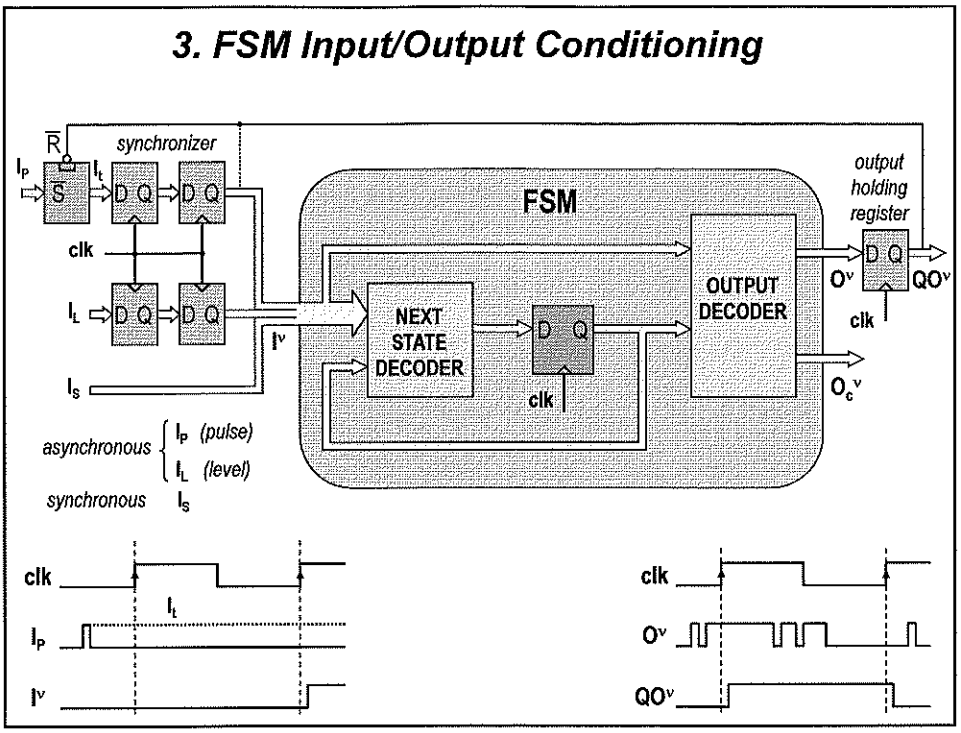
2.2 Registered Outputs: Glitch Free





glitch-free output

3. FSM Input/Output Conditioning



HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

Digitale Synthese
FPGA: XC Virtex-II

XILINX®

EmSD
Embedded System Design

ir. J. Meel

may 2006

1. XILINX Virtex-II Familiy of FPGAs

VIRTEX-II Features

- 0.15µ 8-layer copper CMOS Process
- Power Supply: 1.5 V Core, 3.3V I/O
- capacity to 10M systems gates
- ***Flexible Logic Resources***
  - · CLB: 8 LUTs, 8 FFs
  - · dedicated SOP-logic (Sum of Products)
  - · wide multiplexer logic
- ***Arithmetic Functions***
  - · fast look-ahead carry logic chains
  - · distributed multiplier
  - · dedicated 18x18 Multiplier blocks
- ***Internal Memory***
  - · distributed RAM (LUT)
  - ·18Kbit BlockRAMs

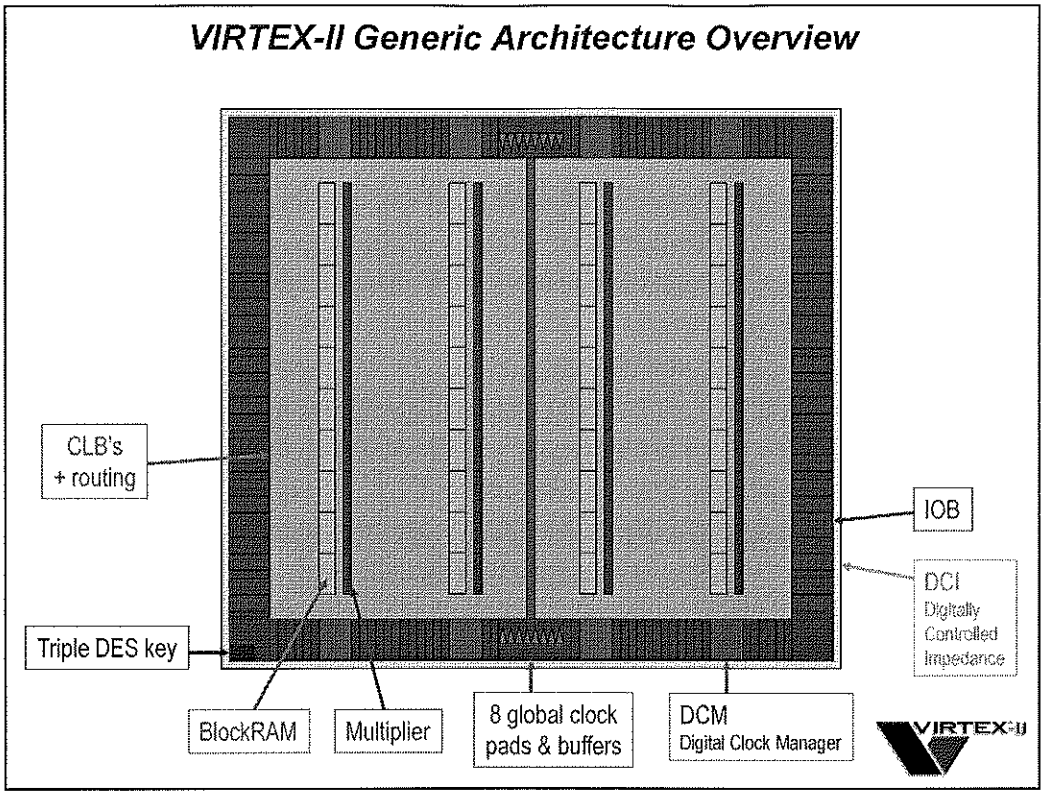
The Virtex-II family is a platform FPGA developed for high performance from low-density to high-density designs that are based on IP cores and customized modules. The family delivers complete solutions for telecommunication, wireless, networking, video, and DSP applications, including PCI, LVDS, and DDR interfaces.

The leading-edge 0.15 µm / 0.12 µm CMOS 8-layer metal process and the Virtex-II architecture are optimized for high speed with low power consumption. Combining a wide variety of flexible features and a large range of densities up to 10 million system gates, the Virtex-II family enhances programmable logic design capabilities and is a powerful alternative to mask-programmed gates arrays.

The Virtex-II family comprises 11 members, ranging from 40K to 8M system gates.

VIRTEX-II Features

- ***High-Performance Interfaces to External Memory***
  - DRAM interfaces (SDR / DDR SDRAM)
  - SRAM interfaces (SDR / DDR / QDR SRAM)
  - CAM interfaces
- ***Digital Clock Managers (DCM)***
  - 200MHz+ System Clock
  - Precise clock de-skew
  - Flexible frequency synthesis
  - High-resolution phase shifting
- ***Select I/O Technology***
  - 840 Mb/s Low-Voltage Differential Signaling I/O (LVDS)
  - Digitally Controlled Impedance (DCI)
  - PCI-X compatible (133 MHz and 66 MHz) at 3.3V
  - 19 single-ended and 6 differential standards
- ***Triple DES encrypted bitstream***
- ***Flip-Chip and Wire-Bond BGA Packages***



Virtex-II devices are user-programmable gate arrays with various configurable elements. The Virtex-II architecture is optimized for high-density and high-performance logic designs.

The programmable device is comprised of input/output blocks (IOBs) and internal configurable logic blocks (CLBs).

Programmable I/O blocks provide the interface between package pins and the internal configurable logic. Most popular and leading-edge I/O standards are supported by the programmable IOBs.

The internal configurable logic includes four major elements organized in a regular array.

• Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. BUFTs (3-state buffers) associated with each CLB element drive dedicated segmentable horizontal routing resources.

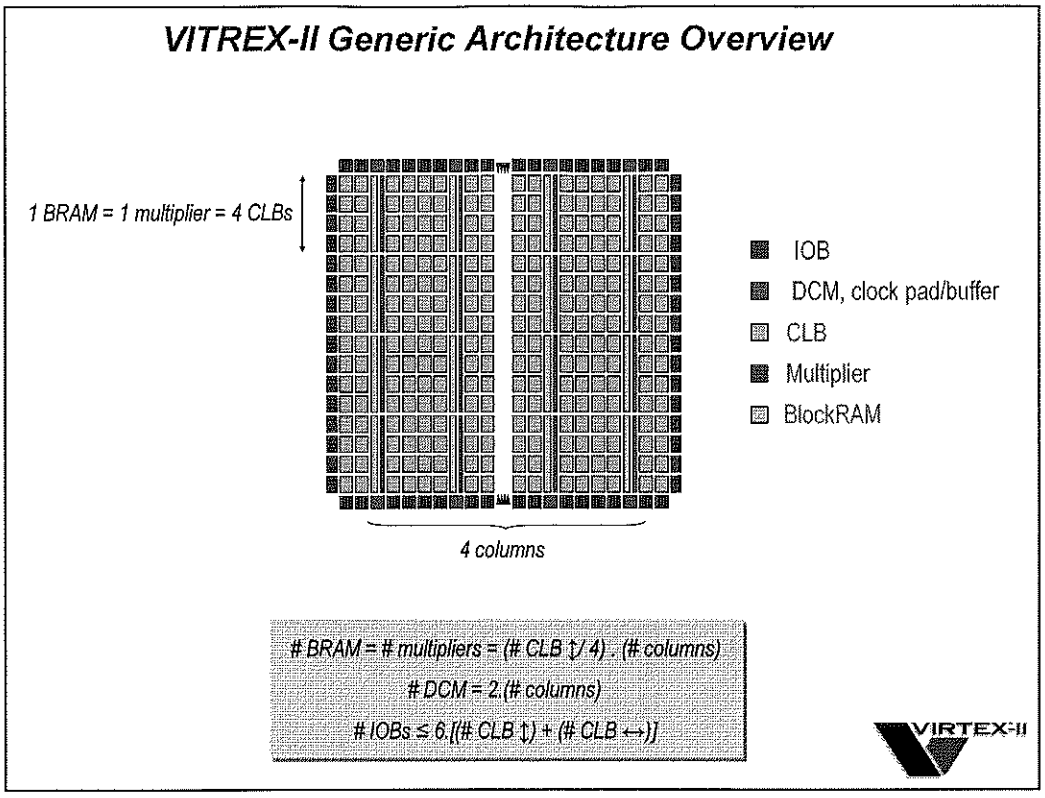
• Block SelectRAM memory modules provide large 18 Kbit storage elements of dual-port RAM.

• Multiplier blocks are 18-bit x 18-bit dedicated multipliers.

• DCM (Digital Clock Manager) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, coarse- and fine-grained clock phase shifting.

A new generation of programmable routing resources called Active Interconnect Technology interconnects all of these elements. The general routing matrix (GRM) is an array of routing switches. Each programmable element is tied to a switch matrix, allowing multiple connections to the general routing matrix. The overall programmable interconnection is hierarchical and designed to support high-speed designs.

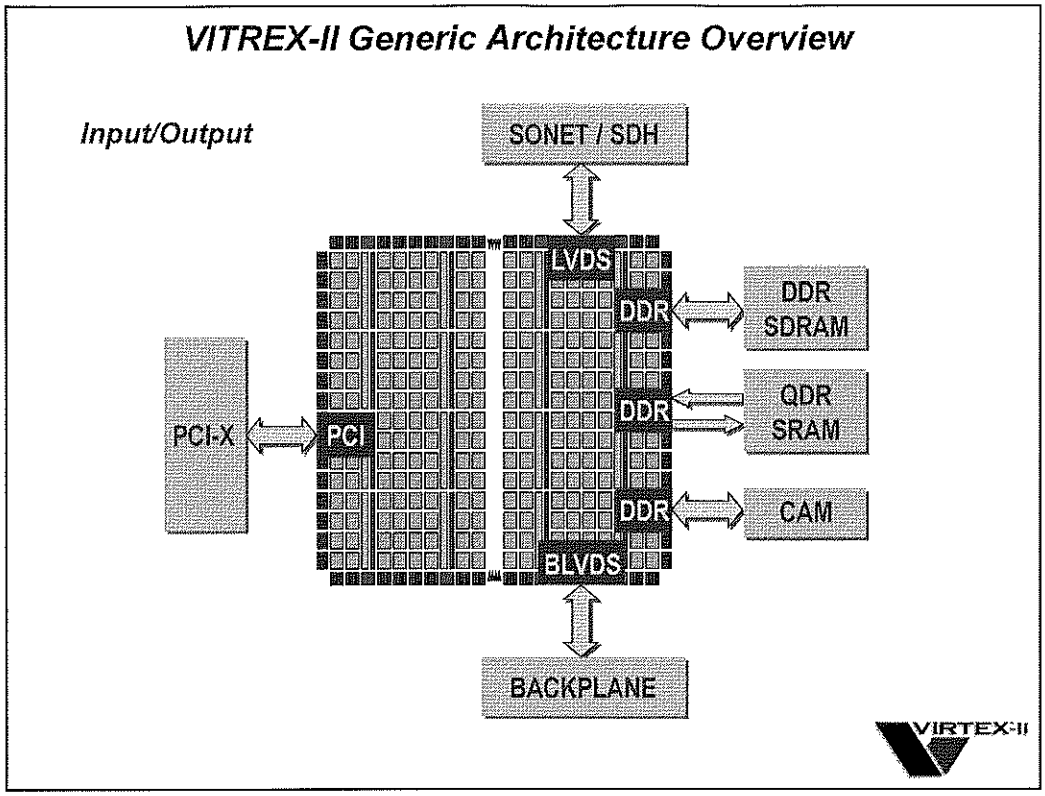
All programmable elements, including the routing resources, are controlled by values stored in static memory cells. These values are loaded in the memory cells during configuration and can be reloaded to change the functions of the programmable elements.



Virtex-II 18Kbit Block SelectRAM memory blocks are located in either two, four or six columns. The number of blocks per column depends of the device array size and is equivalent to the number of CLBs in a column divided by four.

Multiplier organization is identical to the 18 Kbit Block SelectRAM organization, because each multiplier is associated with an 18 Kbit block SelectRAM resource.

Virtex-II DCMs are placed on the top and bottom of each block RAM and multiplier column.



Input/Output Blocks (IOBs)

IOBs are programmable and can be categorized as follows:

• Input block with an optional single-data-rate or double-data-rate (DDR) register

• Output block with an optional single-data-rate or DDR register, and an optional 3-state buffer, to be driven directly or through a single or DDR register

• Bidirectional block (any combination of input and output configurations)

These registers are either edge-triggered D-type flip-flops or level-sensitive latches.

IOBs support the single-ended I/O standards (LVTTL, LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V), PCI-X compatible (133 MHz and 66 MHz) at 3.3V, PCI compliant (66 MHz and 33 MHz) at 3.3V, HSTL (Class I, II, III, and IV), SSTL (3.3V and 2.5V, Class land II) ...)

The digitally controlled impedance (DCI) I/O feature automatically provides on-chip termination for each I/O element.

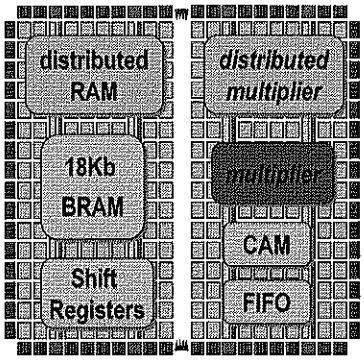
The IOB elements also support differential signaling I/O standards (LVDS, BLVDS (Bus LVDS), ULVDS, LDT, LVPECL). Two adjacent pads are used for each differential pair. Two or four IOB blocks connect to one switch matrix to access the routing resources.

High-Performance Interfaces to External Memory:

- DRAM interfaces:
  - · SDR /DDR SDRAM
  - · Reduced Latency DRAM
- SRAM interfaces
  - · SDR /DDR SRAM
  - · QDR™ SRAM
- CAM interfaces

VITREX-II Generic Architecture Overview

Arithmetic
Memory



SelectRAM™ Memory Hierarchy

- Up to 1.5 Mb of *distributed* SelectRAM resources (can be organised as shiftregisters)
- Up to 93,184 cascadable 16-bit shift registers (distributed)
- 3 Mb of dual-port RAM in 18 Kbit *Block* SelectRAM resources

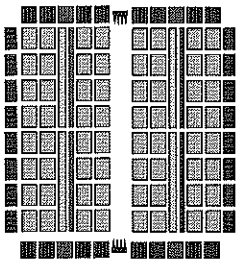
Arithmetic Functions

- Fast look-ahead carry logic chains
- The CLB elements have dedicated logic to implement efficient multipliers in logic
- Dedicated 18-bit x 18-bit multiplier blocks

Virtex-II Family

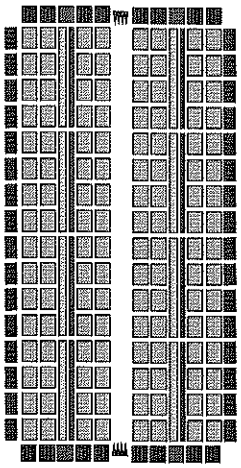
2 Columns BlockRAMs & Multipliers

XC2V40



CLB array: *8 x 8*
 # BRAM: *4*
 # mult: *4*
 # DCM: *4*
 # IOB: *88*

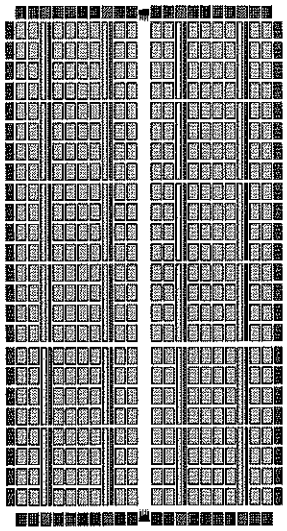
XC2V80



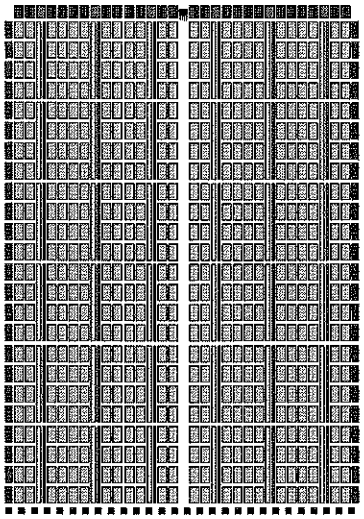
CLB array: *16 x 8*
 # BRAM: *8*
 # mult: *8*
 # DCM: *4*
 # IOB: *120*

4 and 6 Columns BlockRAMs & Multipliers

XC2V250



XC2V3000 ...



CLB array: 24 x 16
 # BRAM&x: 24
 # DCM: 8
 # IOB: 200

VIRTEX-II

Virtex-II Family Members

Device XC2V	40	80	250	500	1000	1500	2000	3000	4000	6000	8000
system gates	40K	80K	250K	500K	1M	1.5M	2M	3M	4M	6M	8M
CLB array	**8 x 8**	**16 x 8**	**24 x 16**	**32 x 24**	**40 x 32**	**48 x 40**	**56 x 48**	**64 x 56**	**80 x 72**	**96 x 88**	**112 x 104**
# slices	256	512	1536	3072	5120	4680	10752	14336	23040	33792	46592
# CLB FFs	512	1024	3072	6144	10240	15360	21504	28672	46080	67584	93184
CLB RAM (kbits)	8	16	48	96	160	240	336	448	720	1056	1456
# 18Kbits BRAM	***4***	***8***	***24***	***32***	***40***	***48***	***56***	***96***	***120***	***144***	***168***
# multiplier	4	8	24	32	40	48	56	96	120	144	168
max IOB (single)	**88**	**120**	**200**	**264**	**432**	**528**	**624**	**720**	**912**	**1.104**	**1.296**
DCM	4	4	8	8	8	8	8	12	12	12	12
configuration memory (bit)	0.4M	0.6M	1.7M	2.8M	4.1M	5.7M	7.5M	10.5M	15.7M	21.9M	29.1M

BRAM & Multipliers: *2 Columns* | *4 Columns* | *6 Columns*

1 CLB = 4 slices = 8 FFs = 128 RAM bits

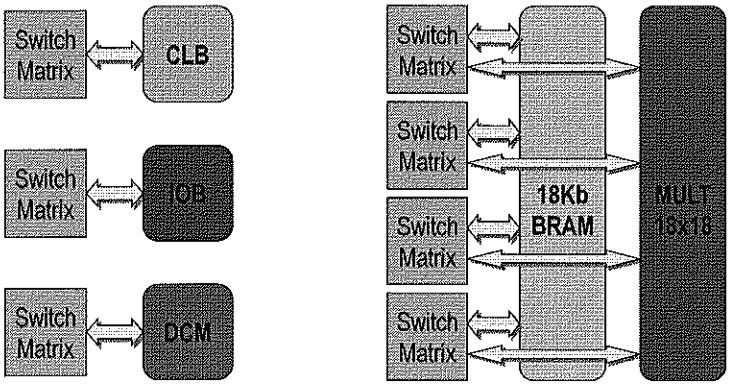
 # BRAM = # multipliers = (# CLB ↕/ 4) (# columns)

 # DCM = 2.(# columns)

 # IOBs ≤ 6.[(# CLB ↕) + (# CLB ↔)]

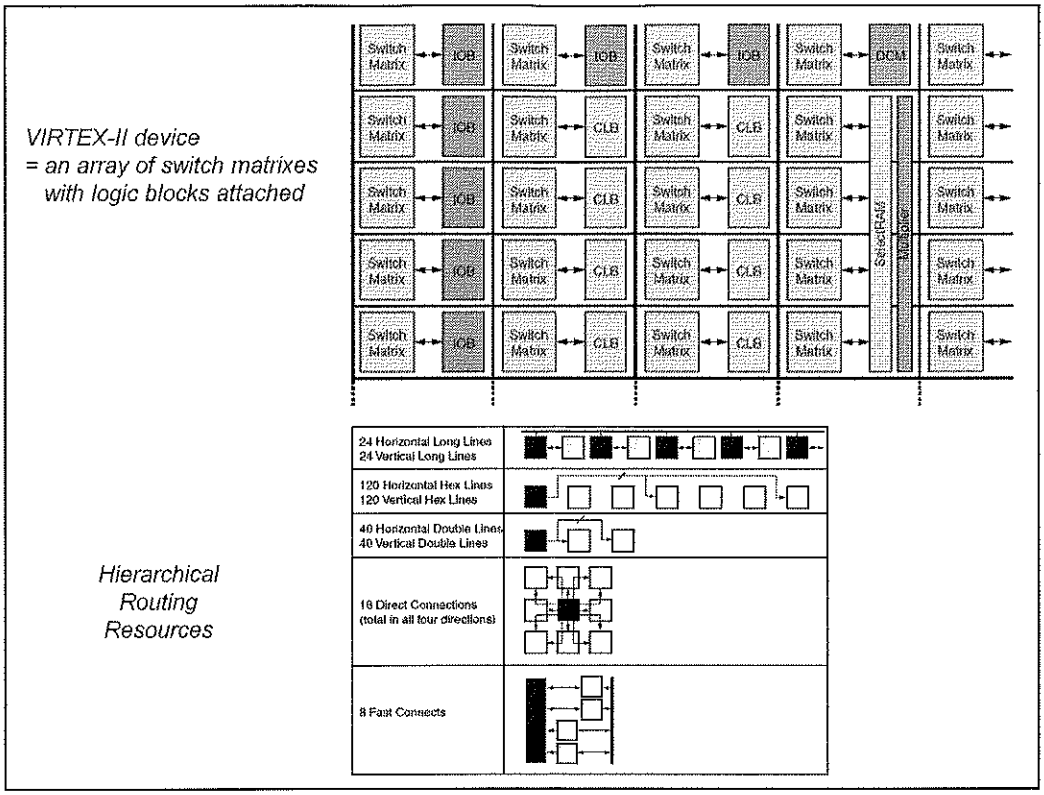
Active Interconnect Technology

- Interconnect an array of switch matrices
- CLBs, IOBs, blockRAM, multipliers, and DCMs are all connected to an identical switch matrix for access to global routing resources
- Simplify design and place & route (regular array)



Local and global Virtex-II routing resources are optimized for speed and timing predictability, as well as to facilitate IP cores implementation. Virtex-II Active Interconnect Technology is a fully buffered programmable routing matrix. All routing resources are segmented to offer the advantages of a hierarchical solution.

The IOB, CLB, Block SelectRAM, multiplier, and DCM elements all use the same interconnect scheme (connection to an identical switch matrix) for access to the global routing matrix. Timing models are shared, greatly improving the predictability of the performance of high-speed designs.



Each Virtex-II device can be represented as an array of switch matrixes with logic blocks attached.

Place-and-route software takes advantage of this regular array to deliver optimum system performance and fast compile times. The segmented routing resources are essential to guarantee IP cores portability and to efficiently handle an incremental design flow that is based on modular implementations. Total design time is reduced due to fewer and shorter design iterations.

Hierarchical Routing Resources

Most Virtex-II signals are routed using the global routing resources, which are located in horizontal and vertical routing channels between each switch matrix.

Virtex-II has fully buffered programmable interconnections, with a number of resources counted between any two adjacent switch matrix rows or columns. Fanout has minimal impact on the performance of a net.

• The *long lines* are bidirectional wires that distribute signals across the device. Vertical and horizontal long lines span the full height and width of the device.

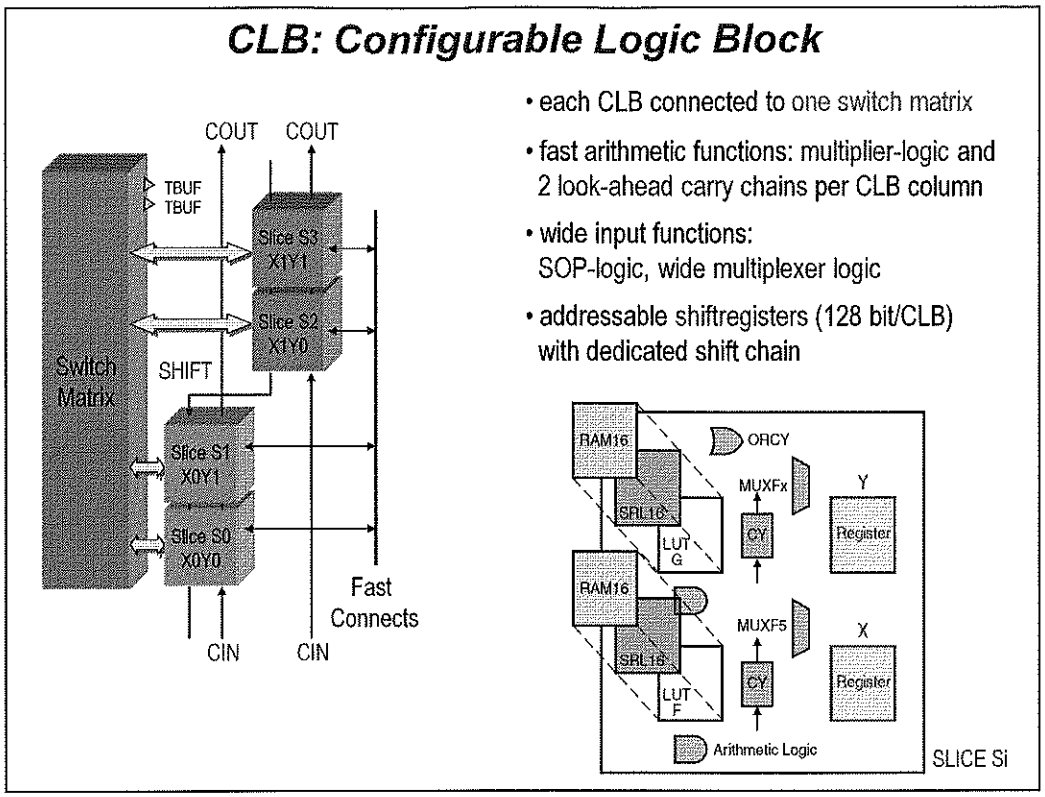
• The *hex lines* route signals to every third or sixth block away in all four directions. Organized in a staggered pattern, hex lines can only be driven from one end. Hex-line signals can be accessed either at the endpoints or at the midpoint (three blocks from the source).

• The *double lines* route signals to every first or second block away in all four directions. Organized in a staggered pattern, double lines can be driven only at their endpoints. Double-line signals can be accessed

either at the endpoints or at the midpoint (one block from the source).

• The *direct connect lines* route signals to neighbouring blocks:vertically, horizontally, and diagonally.

• The *fast connect lines* are the internal CLB local interconnections from LUT outputs to LUT inputs.



The Virtex-II configurable logic blocks (CLB) are organized in an array and are used to build combinatorial and synchronous logic designs.

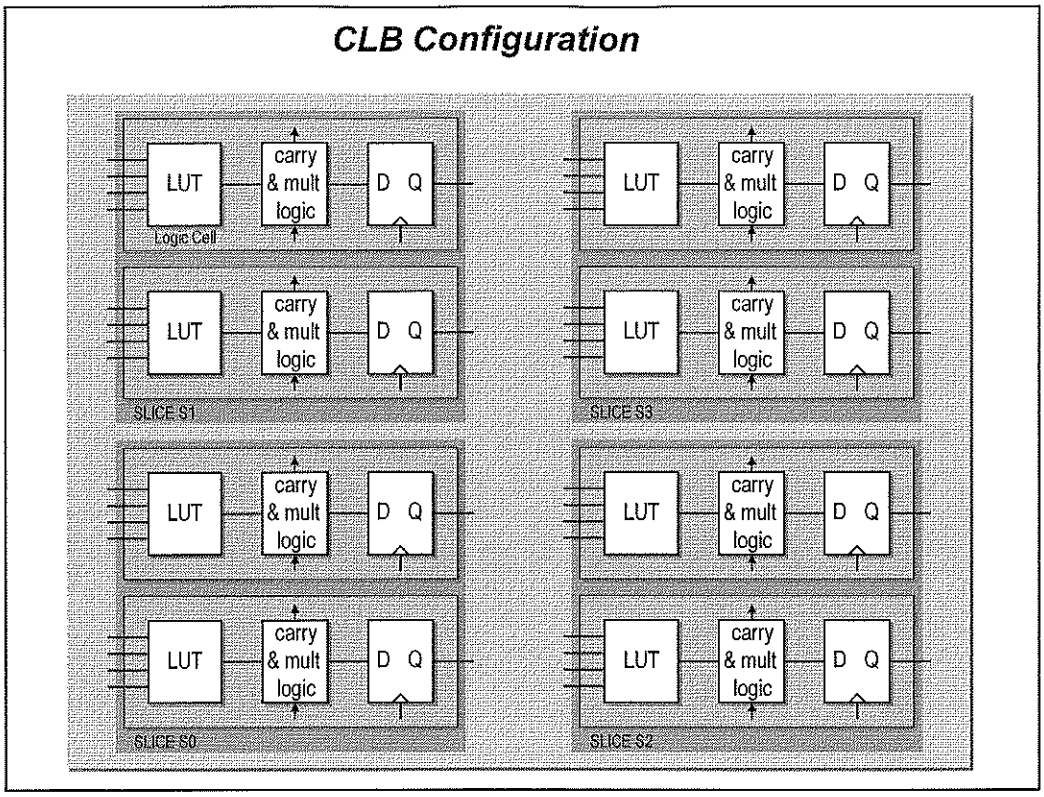
CLB

Each CLB element is tied to a switch matrix to access the general routing matrix. A CLB element comprises 4 similar slices, with fast local feedback within the CLB. The four slices are split in two columns of two slices with two independent carry logic chains and one common shift chain.

SLICE

Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements (either edge-triggered D-type flip-flops or level-sensitive latches). Each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element.

The output from the function generator in each slice drives both the slice output and the D input of the storage element.



Configurable Logic Blocks (CLBs)

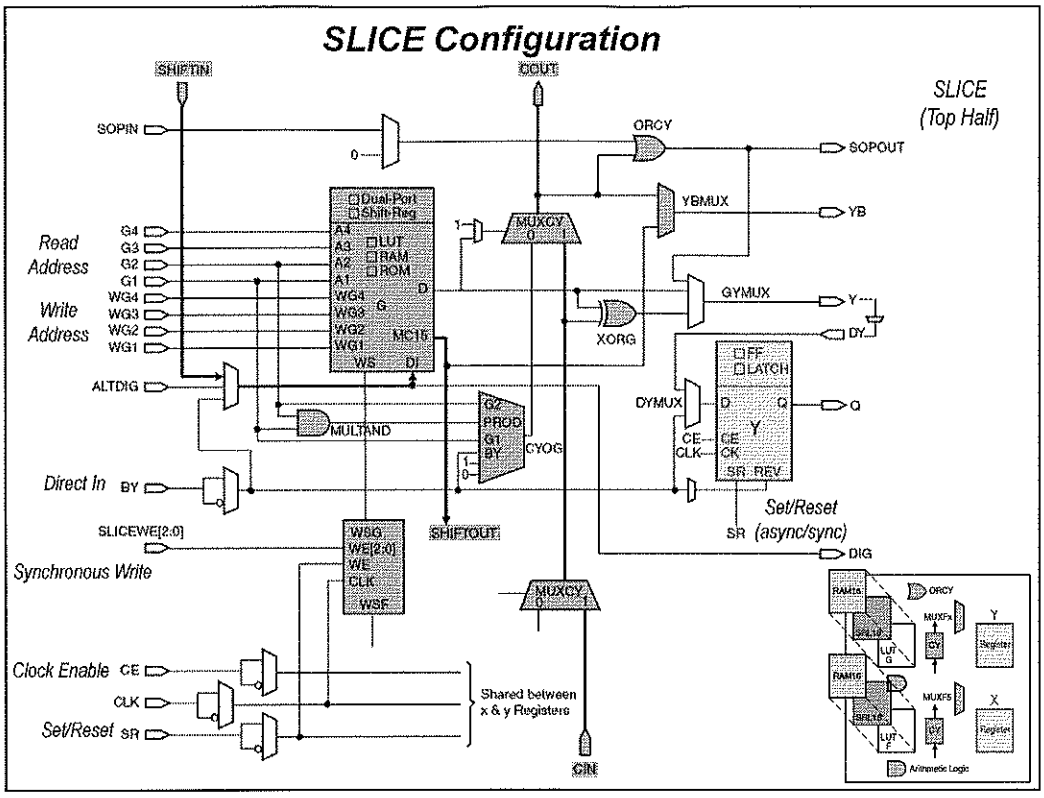
CLB resources include four slices and two 3-state buffers. Each slice is equivalent and contains:

- Two function generators (LUT F & G)
- Two storage elements
- Arithmetic logic gates
- Large multiplexers
- Wide function capability
- Fast carry look-ahead chain
- Horizontal cascade chain (OR gate)

The function generators F & G are configurable as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM memory.

The two storage elements are either edge-triggered D-type flip-flops or level-sensitive latches.

Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.



The output from the function generator in each slice drives both the slice output and the D input of the storage element.

Look-Up Table

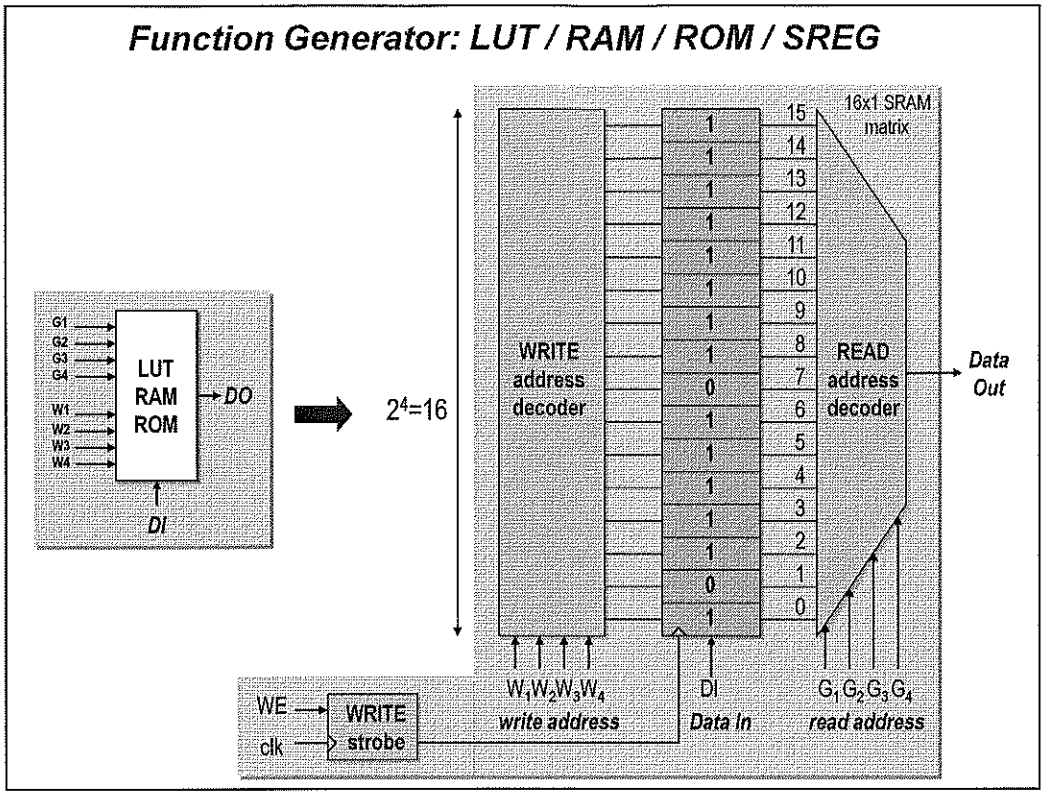
Virtex-II function generators are implemented as 4-input look-up tables (LUTs). Four independent inputs are provided to each of the two function generators in a slice (F and G). These function generators are each capable of implementing any arbitrarily defined boolean function of four inputs. The propagation delay is therefore independent of the function implemented. Signals from the function generators can exit the slice (X or Y output), can input the XOR dedicated gate (for arithmetic logic), or input the carry-logic multiplexer (see fast look-ahead carry logic), or feed the D input of the storage element, or go to the MUXF5 (not shown).

Register/Latch

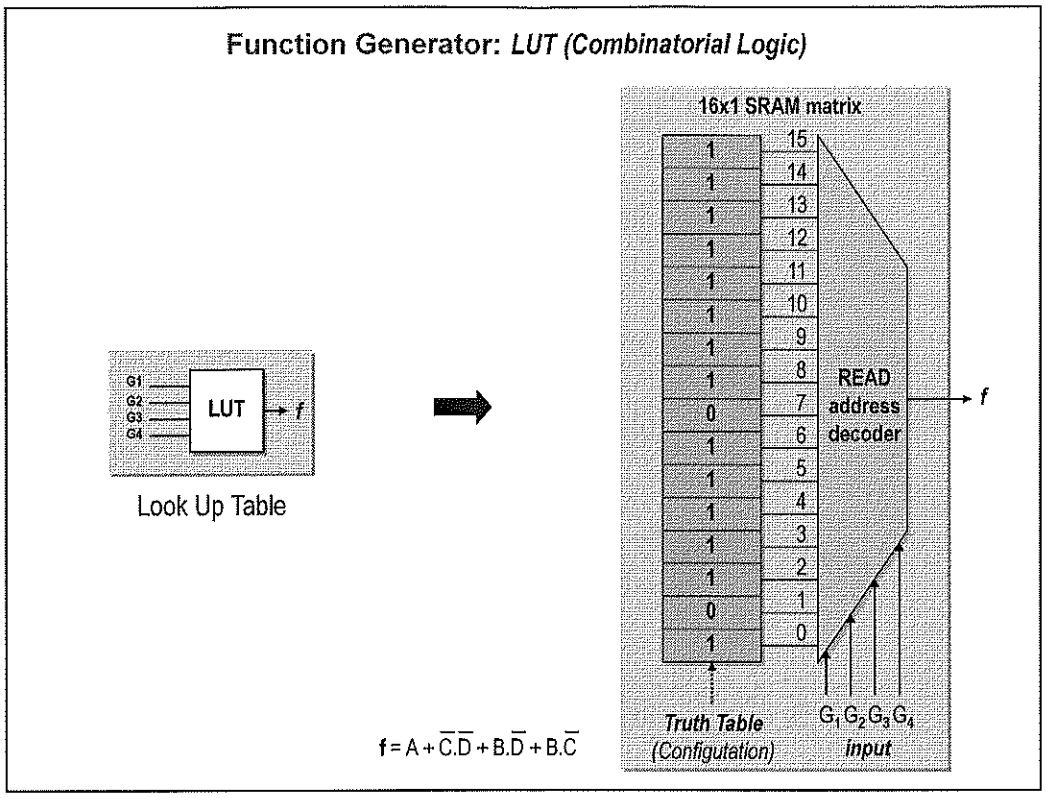
The storage elements in a Virtex-II slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D input can be directly driven by the X or Y output via the DX or DY input, or by the slice inputs bypassing the function generators via the BX or BY input (Direct In). The clock enable signal (CE) is active High by default. If left unconnected, the clock enable for that storage element defaults to the active state.

In addition to clock (CK) and clock enable (CE) signals, each slice has set and reset signals (SR and BY slice inputs). SR forces the storage element into the state specified by the attribute SRHIGH or SRLOW. SRHIGH forces a logic "1" when SR is asserted. SRLOW forces a logic "0". When SR is used, a second input (BY) forces the storage element into the opposite state. The reset condition is predominant over the set condition.

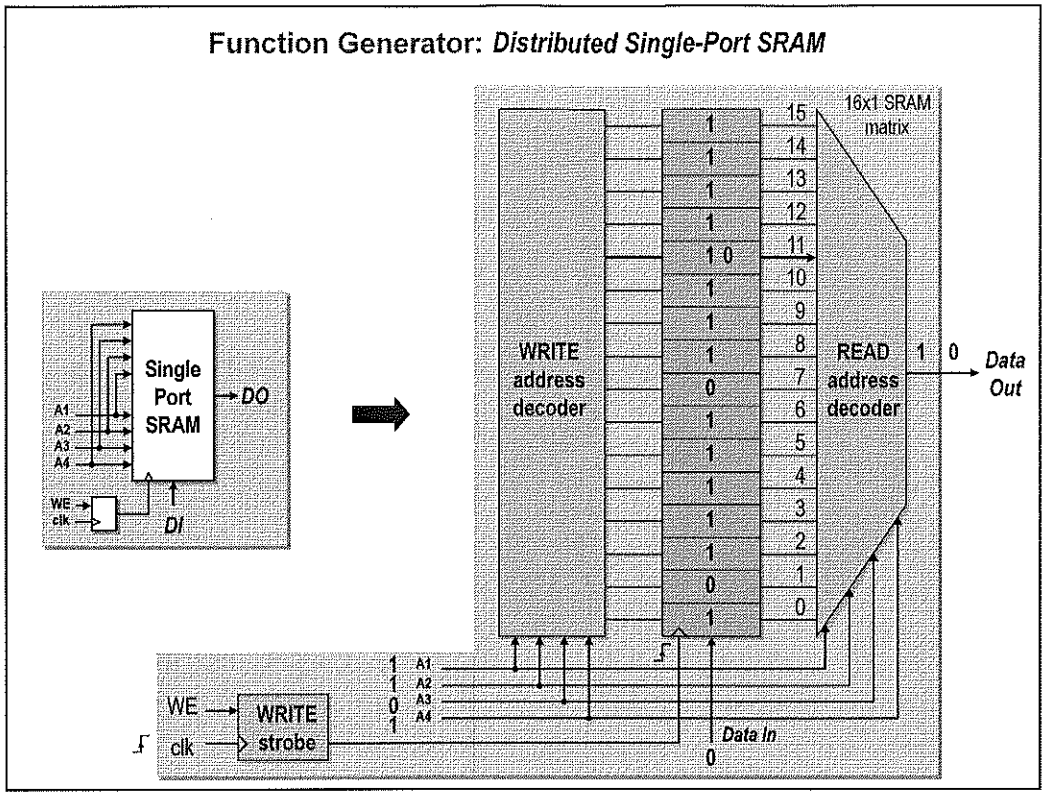
The initial state after configuration or global initial state is defined by a separate INIT0 and INIT1 attribute. For each slice, set and reset can be set to be synchronous or asynchronous. The control signals clock (CLK), clock enable (CE) and set/reset (SR) are common to both storage elements in one slice. All of the control signals have independent polarity.



Distributed SelectRAM memory modules are synchronous (write) resources. The combinatorial read access time is extremely fast, while the synchronous write simplifies high-speed designs. A synchronous read can be implemented with a storage element in the same slice. The distributed SelectRAM memory and the storage element share the same clock input.A Write Enable (WE) input is active High, and is driven by the SR input.

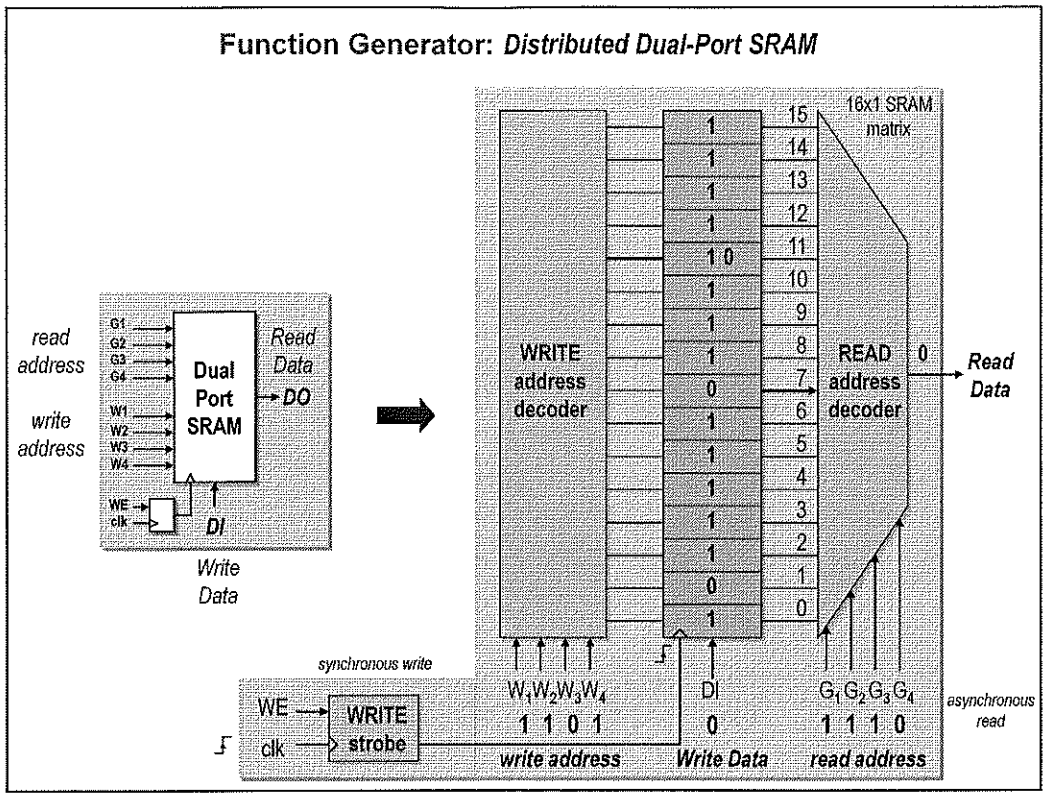


Virtex-II function generators are implemented as 4-input look-up tables (LUTs). Four independent inputs are provided to each of the two function generators in a slice (F and G). These function generators are each capable of implementing any arbitrarily defined boolean function of four inputs. The propagation delay is therefore independent of the function implemented.



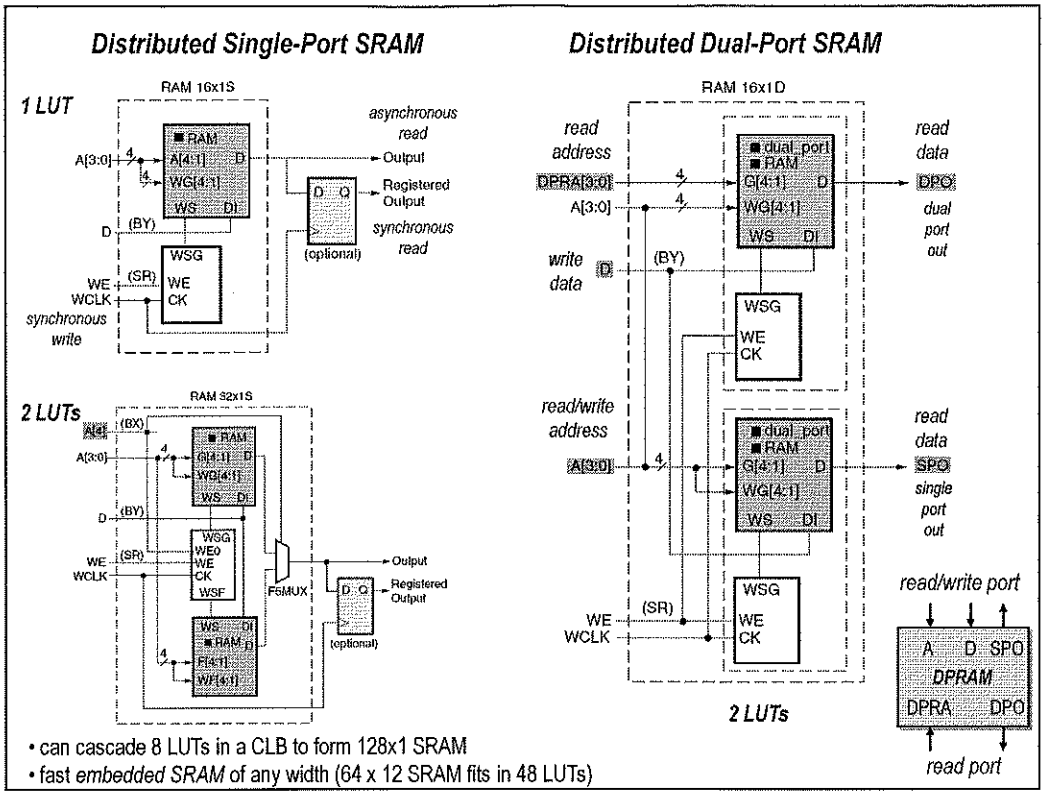
Distributed SelectRAM memory modules are synchronous (write) resources. The combinatorial read access time is extremely fast, while the synchronous write simplifies high-speed designs. A synchronous read can be implemented with a storage element in the same slice. The distributed SelectRAM memory and the storage element share the same clock input.A Write Enable (WE) input is active High, and is driven by the SR input.

For single-port configurations, distributed SelectRAM memory has one address port for synchronous writes and asynchronous reads. Read and write addresses share the same address bus.



For dual-port configurations, distributed SelectRAM memory has one port for synchronous writes and asynchronous reads and another port for asynchronous reads. The function generator (LUT) has separated read address inputs (A1, A2, A3, A4) and write address inputs (WG1/WF1, WG2/WF2, WG3/WF3, WG4/WF4).

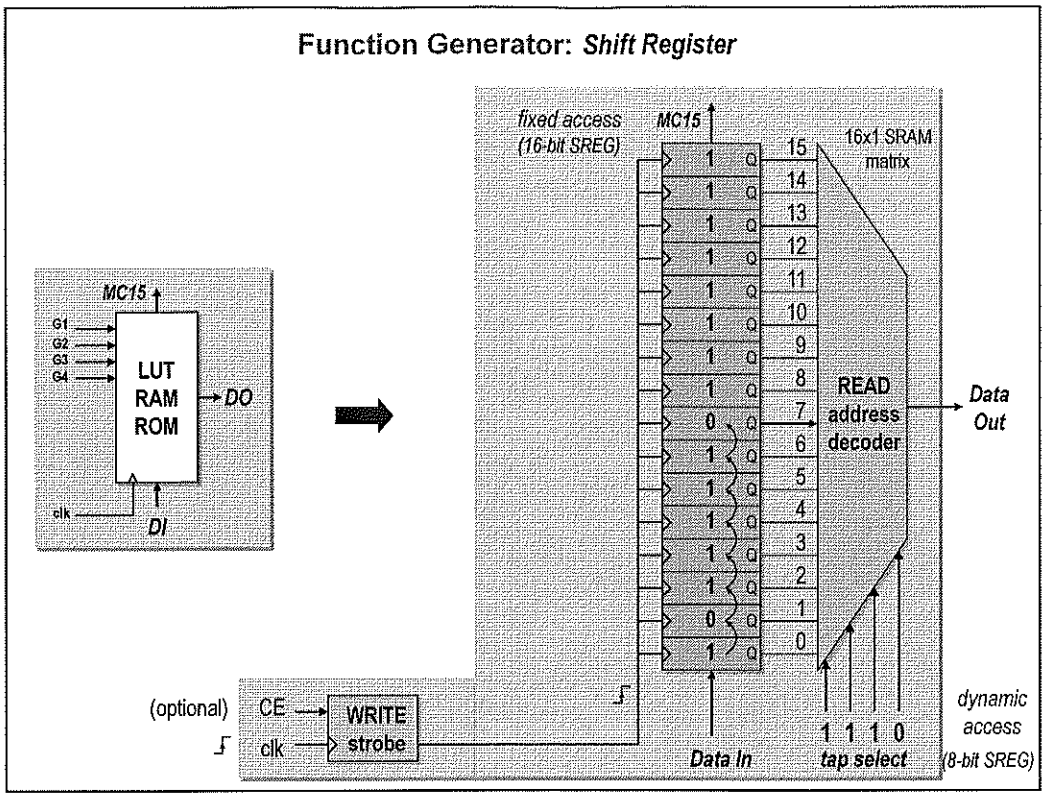
One function generator (R/W port) is connected with shared read and write addresses. The second function generator has the A inputs (read) connected to the second read-only port address and the W inputs (write) shared with the first read/write port address.



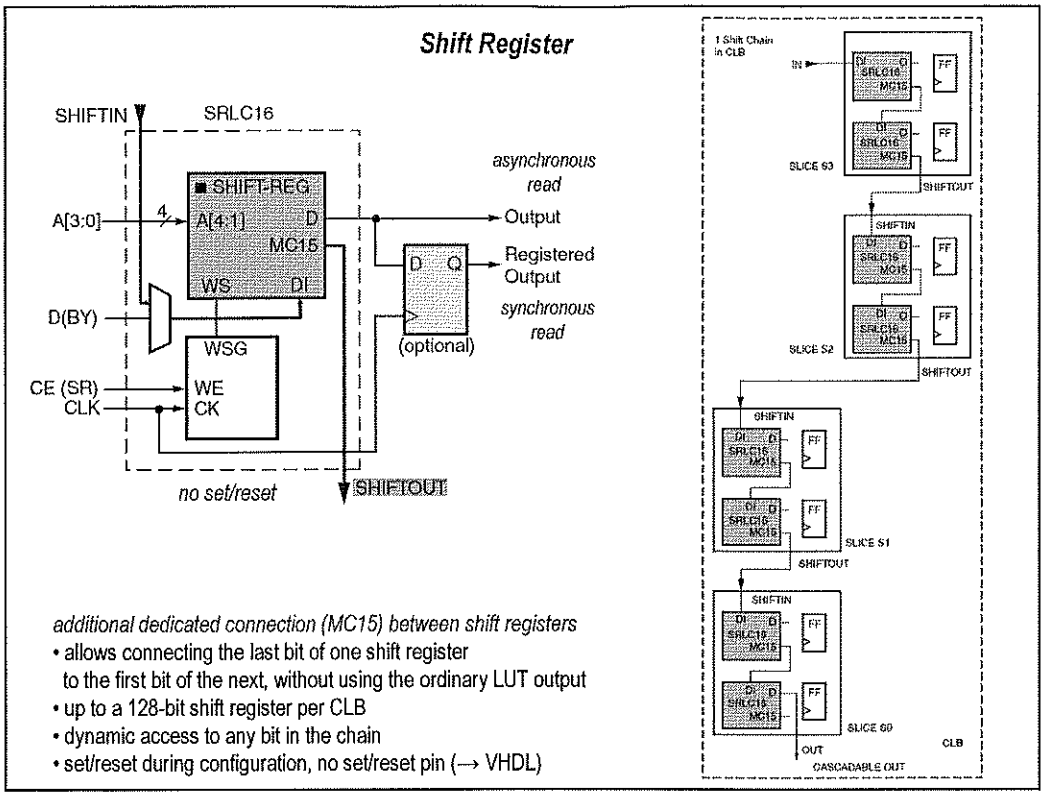
Each function generator (LUT) can implement a 16 x 1-bit synchronous RAM resource called a distributed SelectRAM element. The SelectRAM elements are configurable within a CLB to implement the following:

- Single-Port 16 x 8 bit RAM
- Single-Port 32 x 4 bit RAM
- Single-Port 64 x 2 bit RAM
- Single-Port 128 x 1 bit RAM
- Dual-Port 16 x 4 bit RAM
- Dual-Port 32 x 2 bit RAM
- Dual-Port 64 x 1 bit RAM

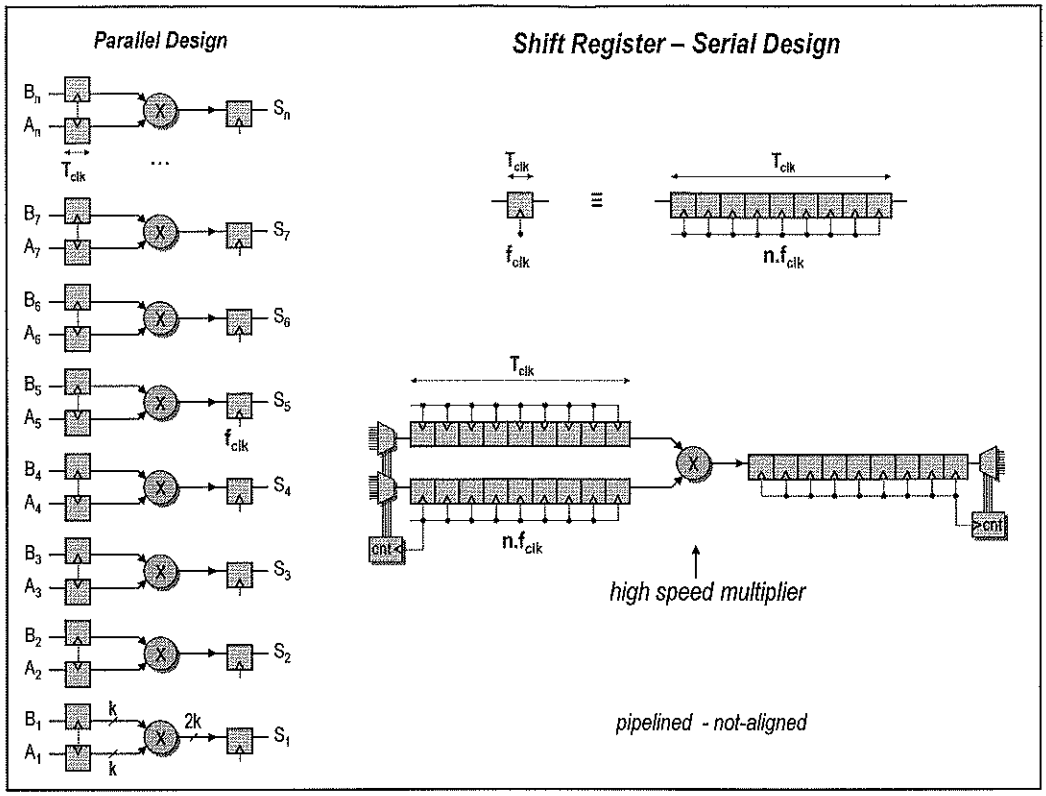
Distributed SelectRAM memory modules are synchronous (write) resources. The combinatorial read access time is extremely fast, while the synchronous write simplifies high-speed designs. A synchronous read can be implemented with a storage element in the same slice. The distributed SelectRAM memory and the storage element share the same clock input.A Write Enable (WE) input is active High, and is driven by the SR input.



Each function generator can also be configured as a 16-bit shift register. The write operation is synchronous with a clock input (CLK) and an optional clock enable CE. A dynamic read access is performed through the 4-bit address bus, A[3:0]. The configurable 16-bit shift register cannot be set or reset. The read is asynchronous, however the storage element or flip-flop is available to implement a synchronous read.Th e storage element should always be used with a constant address. For example, when building an 8-bit shift register and configuring the addresses to point to the 7th bit, the 8th bit can be the flip-flop. The overall system performance is improved by using the superior clock-to-out of the flip-flops.



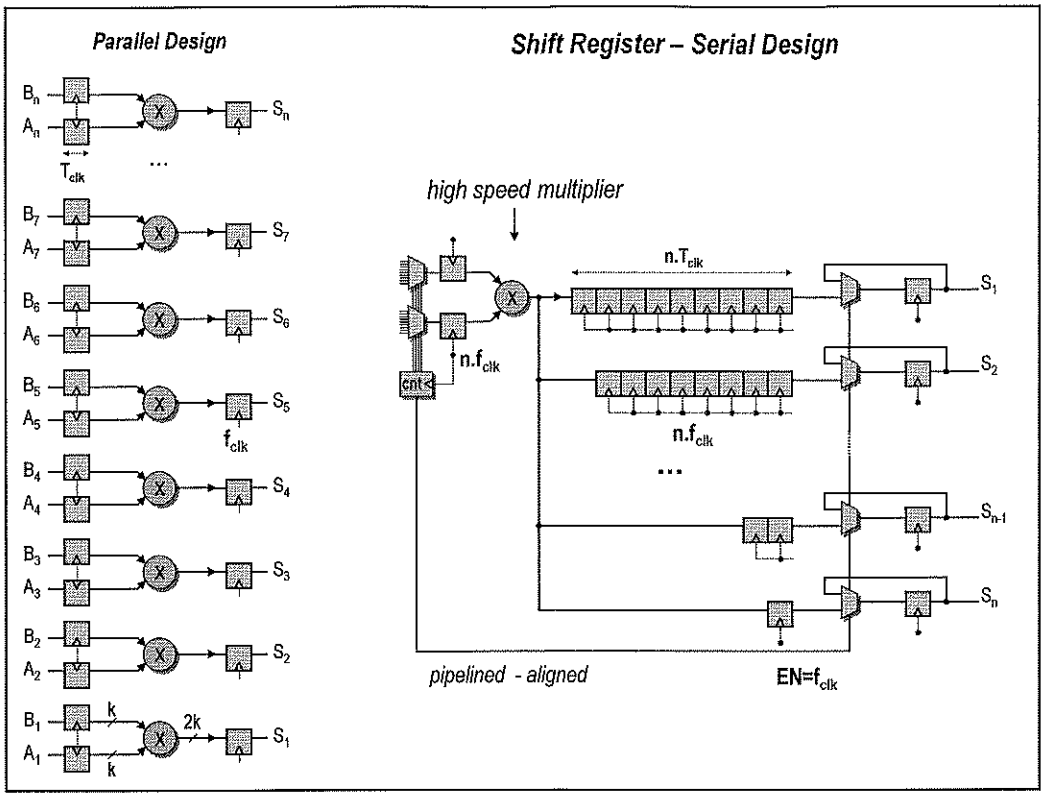
An additional dedicated connection between shift registers allows connecting the last bit of one shift register to the first bit of the next, without using the ordinary LUT output. Longer shift registers can be built with dynamic access to any bit in the chain. The shift register chaining and the MUXF5, MUXF6, and MUXF7 multiplexers allow up to a 128-bit shift register with addressable access to be implemented in one CLB.



With a time multiplexing scheme the shift register can be used to share common hardware with n data streams.

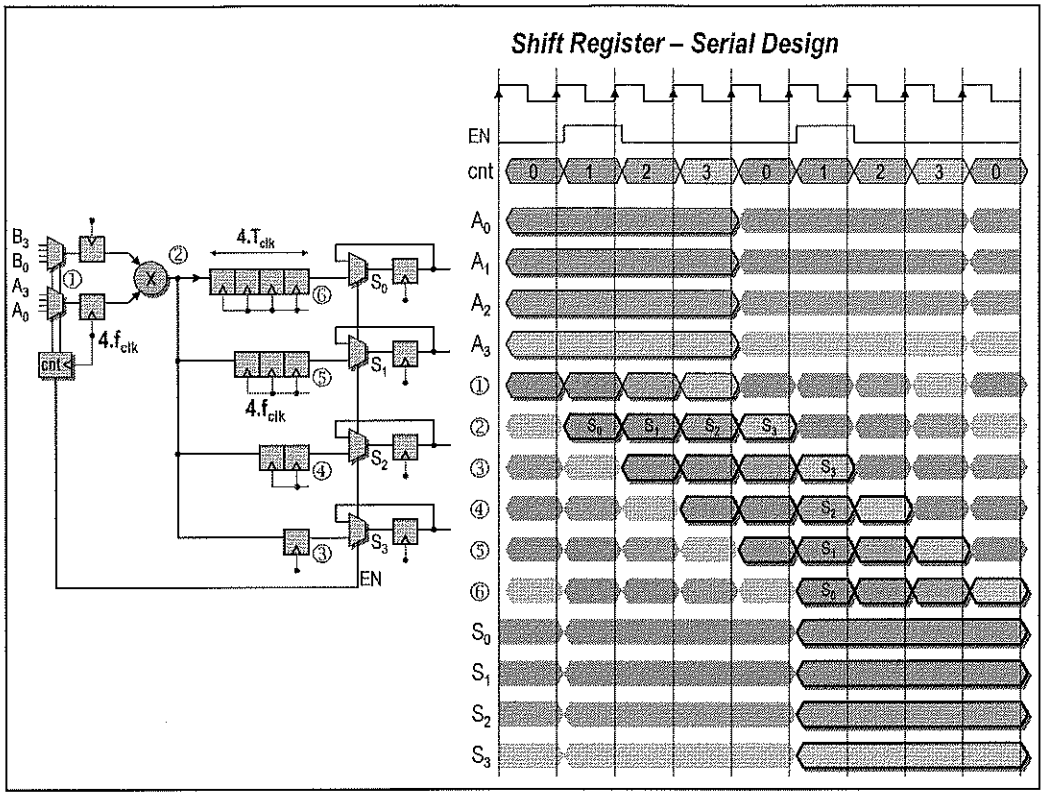
The clock speed must be increased with n. An original delay of T_{clk} must be implemented by a shift register of n flip-flop's.

The shared hardware must run at a n times higher clock speed.



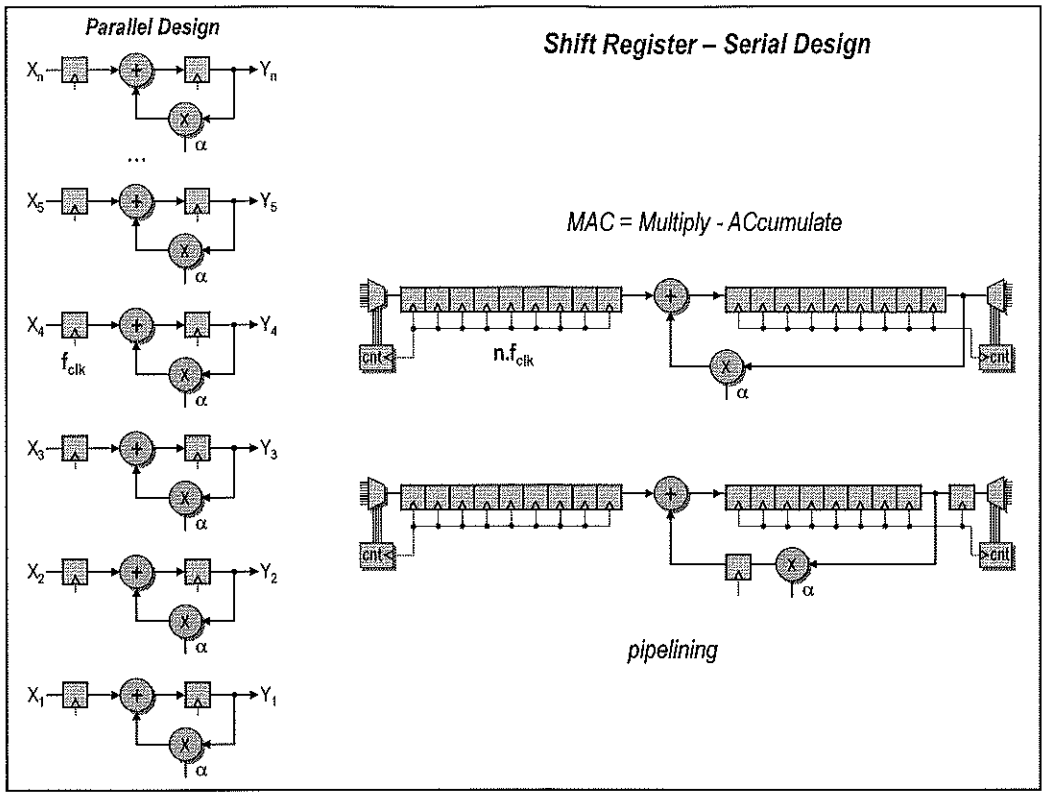
In a time multiplexing scheme the original parallel and aligned data is reduced by a factor n and placed in a sequential data stream. This is the parallel-to-serial conversion.

After processing by the shared hardware a serial-to-parallel conversion is executed. To align the resulting parallel data a delay must be introduced which depends on the position of the serial data in the sequential data stream. The data processed first will be delayed for the longest time. The different delays can be implemented by independent shift registers. A single shift register with n sequential branches could also be used, but this has to be implemented with flip-flop's because the shift registers implemented in the LUTs have only a single output.

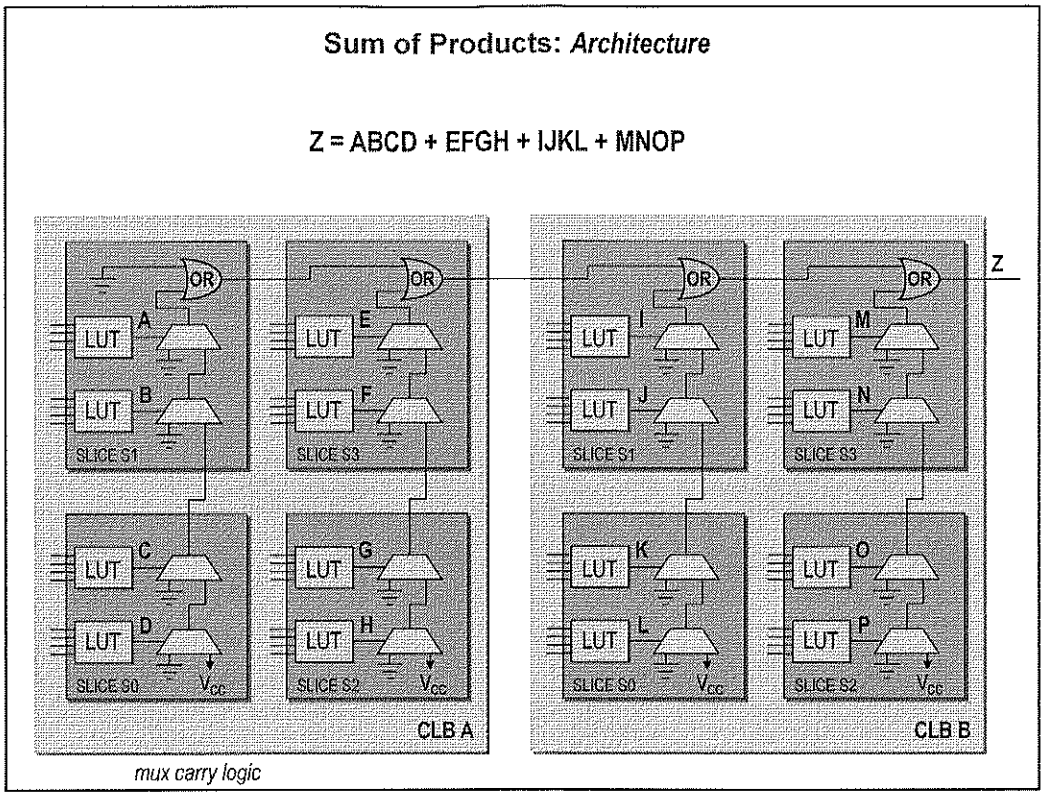


In a time multiplexing scheme the original parallel and aligned data is reduced by a factor n and placed in a sequential data stream. This is the parallel-to-serial conversion.

After processing by the shared hardware a serial-to-parallel conversion is executed. To align the resulting parallel data a delay must be introduced which depends on the position of the serial data in the sequential data stream. The data processed first will experience the longest delay.



Pipelining can be used to increase the system clock speed. Different combinatorial operations are separated by registers to reduce the critical path in the system.

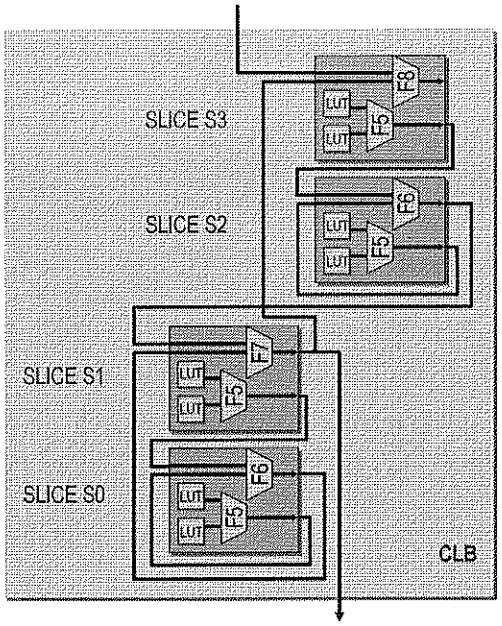


Sum of Products

Each Virtex-II slice has a dedicated OR gate named ORCY, ORing together outputs from the slices carryout and the ORCY from an adjacent slice. The ORCY gate with the dedicated Sum of Products (SOP) chain are designed for implementing large, flexible SOP chains. One input of each ORCY is connected through the fast SOP chain to the output of the previous ORCY in the same slice row. The second input is connected to the output of the top MUXCY in the same slice.

LUTs and MUXCYs can implement large AND gates or other combinatorial logic functions.

Wide Multiplexer Logic: *Architecture*



MUXF8 combines the 2 MUXF7 outputs (2 CLBs)

MUXF5 combines 2 LUT outputs (1 SLICE)

MUXF6 combines SLICES S2 and S3

MUXF7 combines the 2 MUXF6 outputs

MUXF6 combines SLICES S0 and S1

2:1 mux (1 LUT)
4:1 mux (1 SLICE)
8:1 mux (2 SLICEs)
16:1 mux (1 CLB = 4 SLICEs)
32:1 mux (2 CLB = 8 SLICEs)

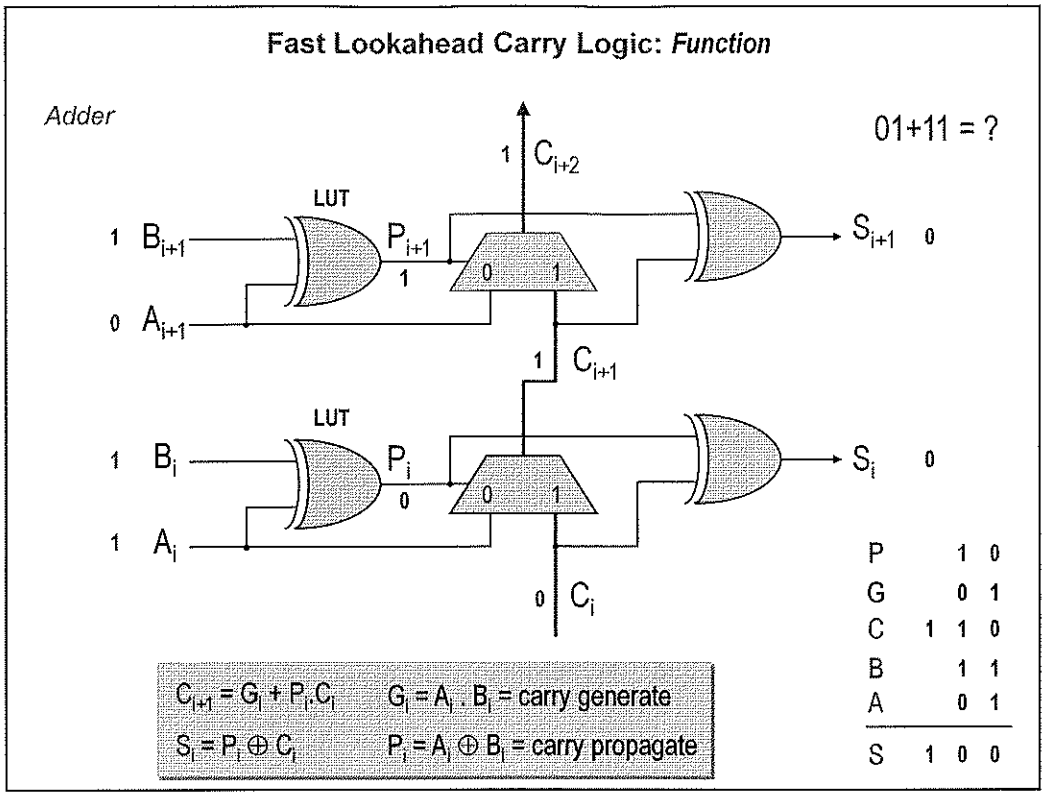
In addition to the basic LUTs, the Virtex-II slice contains logic (MUXF5 and MUXFX multiplexers) that combines function generators to provide any function of five, six, seven, or eight inputs. The MUXFX are either MUXF6,MUXF 7 or MUXF8 according to the slice considered in the CLB. Selected functions up to nine inputs (MUXF5 multiplexer) can be implemented in one slice. The MUXFX can also be a MUXF6, MUXF7, or MUXF8 multiplexers to map any functions of six, seven, or eight inputs and selected wide logic functions.

Virtex-II function generators and associated multiplexers can implement the following:

- 4:1 multiplexer in one slice
- 8:1 multiplexer in two slices
- 16:1 multiplexer in one CLB element (4 slices)
- 32:1 multiplexer in two CLB elements (8 slices)

Each Virtex-II slice has one MUXF5 multiplexer and one MUXFX multiplexer. The MUXFX multiplexer implements the MUXF6,MUXF 7,or MUXF8.

Each CLB element has two MUXF6 multiplexers, one MUXF7 multiplexer and one MUXF8 multiplexer. Any LUT can implement a 2:1 multiplexer.

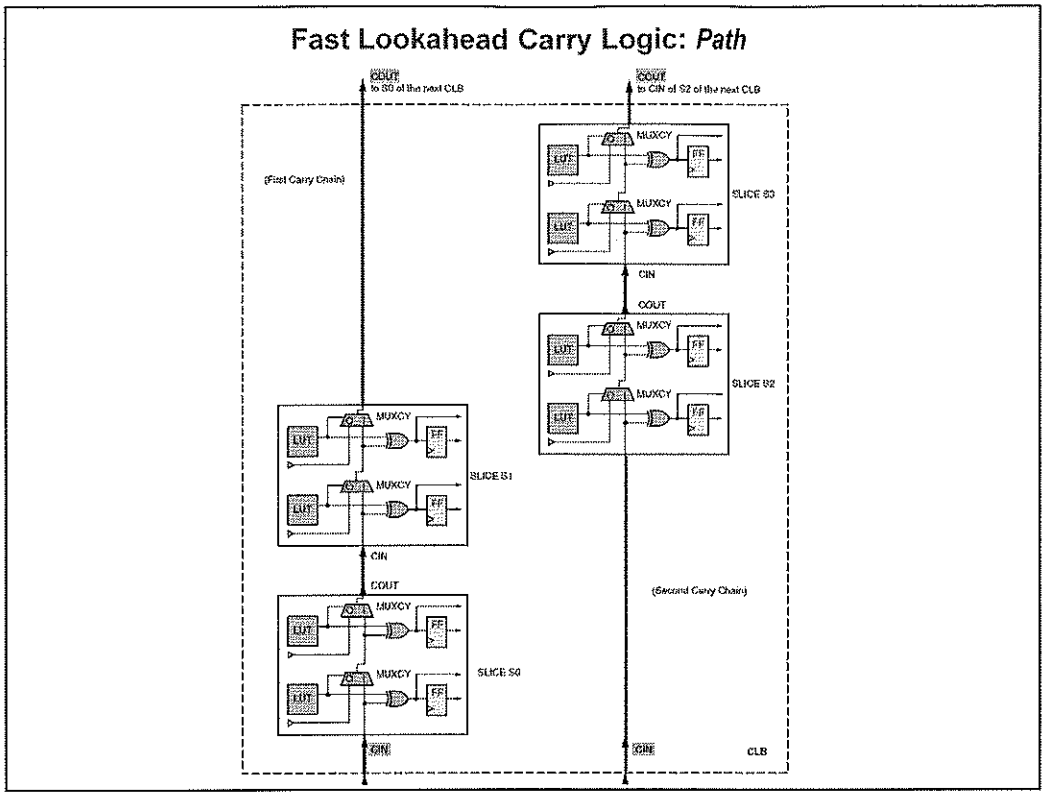


Fast Lookahead Carry Logic

Dedicated carry logic provides fast arithmetic addition and subtraction. The Virtex-II CLB has two separate carry chains.The height of the carry chains is two bits per slice.The carry chain in the Virtex-II device is running upward. The dedicated carry path and carry multiplexer (MUXCY) can also be used to cascade function generators for implementing wide logic functions.

Arithmetic Logic

The arithmetic logic includes an XOR gate that allows a 2-bit full adder to be implemented within a slice.A dedicated AND (MULT_AND) gate improves the efficiency of multiplier implementation.



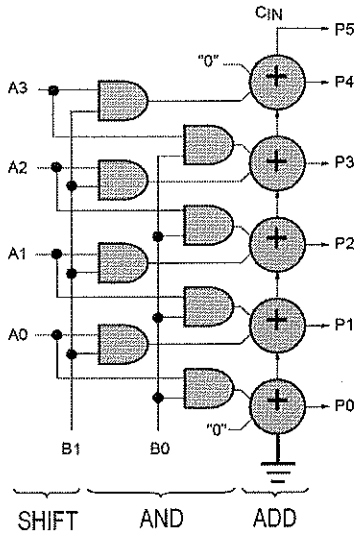
Dedicated carry logic provides fast arithmetic carry capability for high speed arithmetic functions.

There are two separate carry chains in the CLB, one per slice. The height of the carry chains is 4 bits per CLB.The logic consists of a 2-input MUX and an XOR gate. The XOR gate allows a 1-bit full adder to be implemented for 1 bit.

In addition, a dedicated AND gate improves the efficiency of the multiplier implementation. The dedicated carry path is used to cascade LUT functions for implementing wide logic functions. This reduces logic delays due to the decreased number of logic levels even for very high fan-in functions.

Dedicated Multiplier Arithmetic: *Function*

Nx2 multiplier



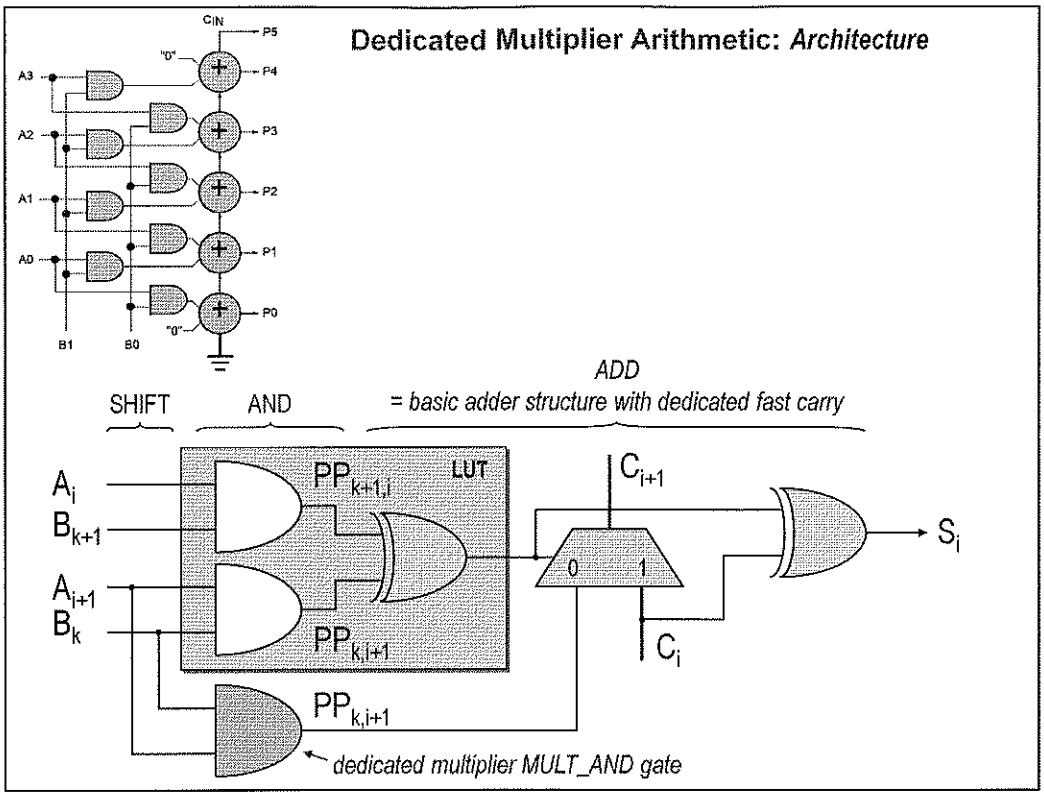
01101 * 101 = ?

	01101	**A**
*	101	**B**
	01101	(A<<0) and B(0) = PP_0
	000000	(A<<1) and B(1) = PP_1
+	0110100	(A<<2) and B(2) = PP_2
	1000001	**P**

SHIFT, AND, ADD

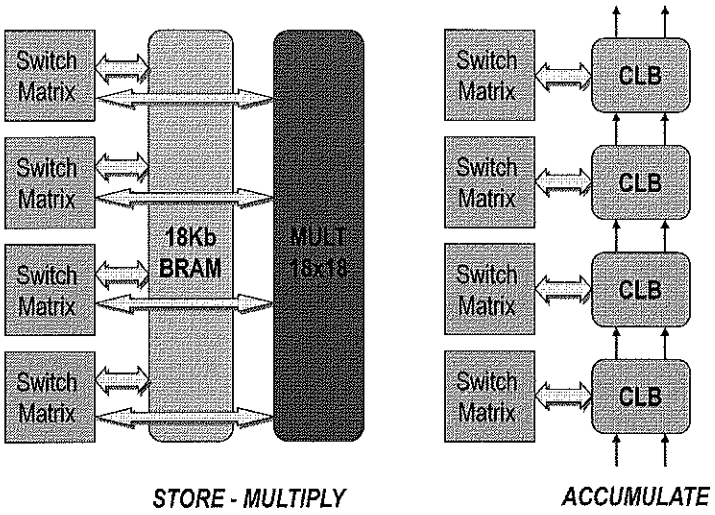
A*B = (A<<0 and B(0)) + ((A<<1) and B(1)) + ((A<<2) and B(2))

The multiplication of two (unsigned) binary multiplicands is essentially a series of shift and add operations. The figure illustrates how an N x 2 Full Multiplier is implemented using the same logic resources as a simple adder. The key to implementing multipliers efficiently in a Virtex device is leveraging the extra AND gate (next to each LUT) and the carry-chain logic.



A dedicated AND (MULT_AND) gate improves the efficiency of multiplier implementation.

SelectRAM en Multiplier Blocks



DSP MAC function

Sharing of the interconnect is optimized for an 18 bit-wide BlockRAM feeding the multiplier block.

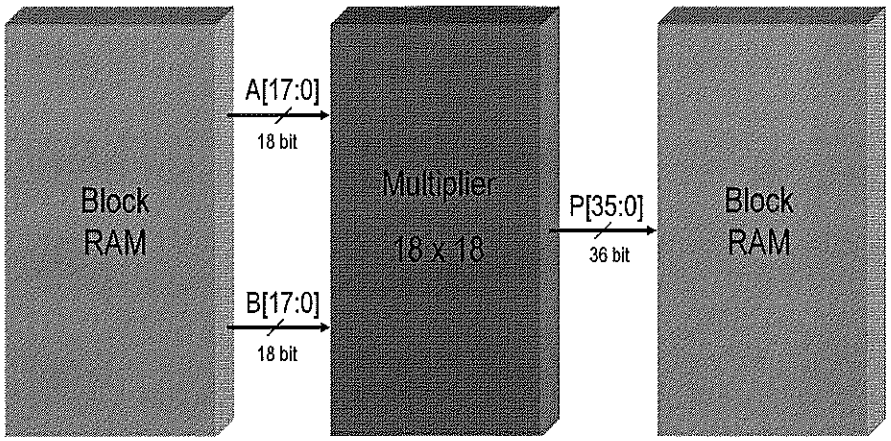
Each SelectRAM memory and multiplier block is tied to four switch matrices.

The interconnect is designed to allow SelectRAM memory and multiplier blocks to be used at the same time, but some interconnect is shared between the SelectRAM and the multiplier. Thus, SelectRAM memory can be used only up to 18 bits wide when the multiplier is used, because the multiplier shares inputs with the upper data bits of the SelectRAM memory.

This sharing of the interconnect is optimized for an 18-bit-wide block SelectRAM resource feeding the multiplier.

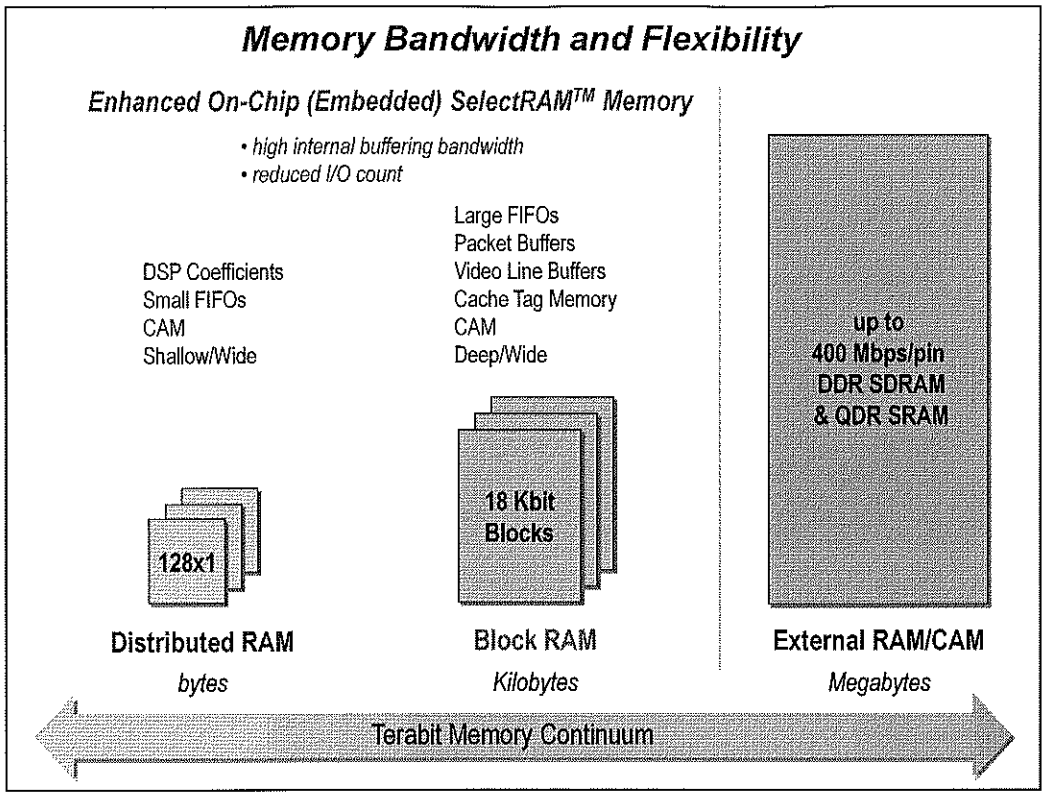
The use of SelectRAM memory and the multiplier with an accumulator in LUTs allows for implementation of a digital signal processor (DSP) multiplier-accumulator (MAC) function, which is commonly used in finite and infinite impulse response (FIR and IIR) digital filters.

18-bit x 18-bit Multipliers



- 2's complement signed operation
- optimized for high-speed operations (I/O ports can be pipelined in CLBs: 230 MHz)
- lower power consumption compared to a distributed multiplier
- fast interface between BlockRAM and Multiplier is ideal for MAC architecture
- BlockRAM and Multiplier are organized in columns

A Virtex-II multiplier block is an 18-bit by 18-bit 2's complement signed multiplier. Virtex-II devices incorporate many embedded multiplier blocks. These multipliers can be associated with an 18 Kbit block SelectRAM resource or can be used independently. They are optimized for high-speed operations and have a lower power consumption compared to an 18-bit x 18-bit multiplier in slices.



A terabit memory continuum is available:

SelectRAM™ Memory Hierarchy

Virtex-IIdevices incorporate large amounts of 18 Kbit block SelectRAM. These complement the distributed SelectRAM resources that provide shallow RAM structures implemented in CLBs.

- 3 Mb of dual-port RAM in 18 Kbit *block* SelectRAM resources
- up to 1.5 Mb of *distributed* SelectRAM resources

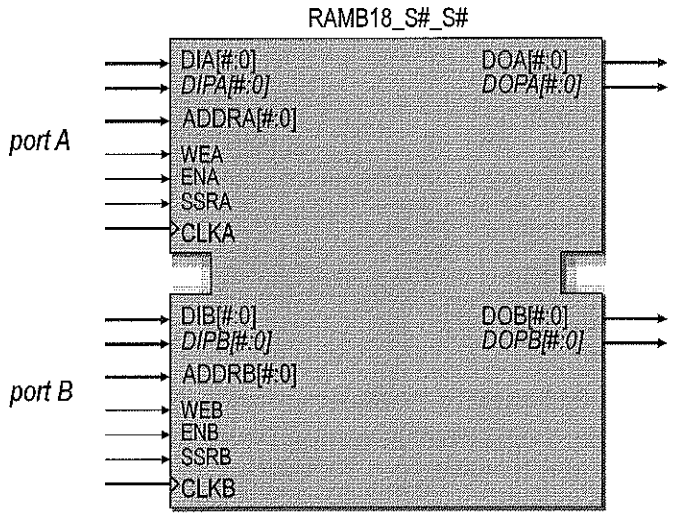
High-Performance Interfaces to External Memory (IOB)

- DRAM interfaces
  - · SDR /DDR SDRAM
  - · Network FCRAM
  - · Reduced Latency DRAM
- SRAM interfaces
  - · SDR /DDR SRAM
  - · QDR™ SRAM
- CAM interfaces

Block S(elect)RAM

18Kbit true dual-port RAM

- two independently clocked/controlled fully synchronous ports
- each port has register-like access (data, address and control)
- separate data [D] and parity [DP] inputs/outputs



Control Signal	Function
CLK	read and write clock
EN	synchronous enable (read, write, set/reset)
WE	write enable
SSR	set DO register to SRVAL (attribute)

Virtex-IIdevices incorporate large amounts of 18 Kbit block SelectRAM. These complement the distributed SelectRAM resources that provide shallow RAM structures implemented in CLBs. Each Virtex-II block SelectRAM is an 18 Kbit true dual-port RAM with two independently clocked and independently controlled synchronous ports that access a common storage area. Both ports are functionally identical.CLK, EN, WE, and SSR polarities are defined through configuration.

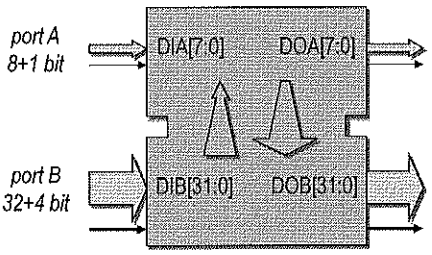
Each port has the following types of inputs: Clock and Clock Enable, Write Enable, Set/Reset, and Address, as well as separate Data/parity data inputs (for write) and Data/parity data outputs (for read).

Operation is synchronous; the block SelectRAM behaves like a register. Control, address and data inputs must (and need only) be valid during the set-up time window prior to a rising (or falling, a configuration option) clock edge. Data outputs change as a result of the same clock edge.

True Dual-Port Block RAM: *Configurations*

configurations available on each port

Configuration	Capacity	Width	Depth	Address	Data	Parity	
16K x 1	16Kb	1	16384	14	1	0	no parity bits reduced capacity
8K x 2	16Kb	2	8192	13	2	0	
4K x 4	16Kb	4	4096	12	4	0	
2K x 9	18Kb	9	2048	11	8	1	parity bit / 8 bits full capacity
1K x 18	18Kb	18	1024	10	16	2	
512 x 36	18Kb	36	512	9	32	4	



- up to 3 Mb on-chip block RAM (cascadable blocks)
- parity bits are stored and behave as data bits
- parity bits must be generated/checked externally in user logic (CLBs)
- independent port A and B data width configuration
- built-in bus-width conversion including parity bits

The Virtex-II block SelectRAM supports various configurations, including single- and dual-port RAM and various data/address aspect ratios.

Single-Port Configuration

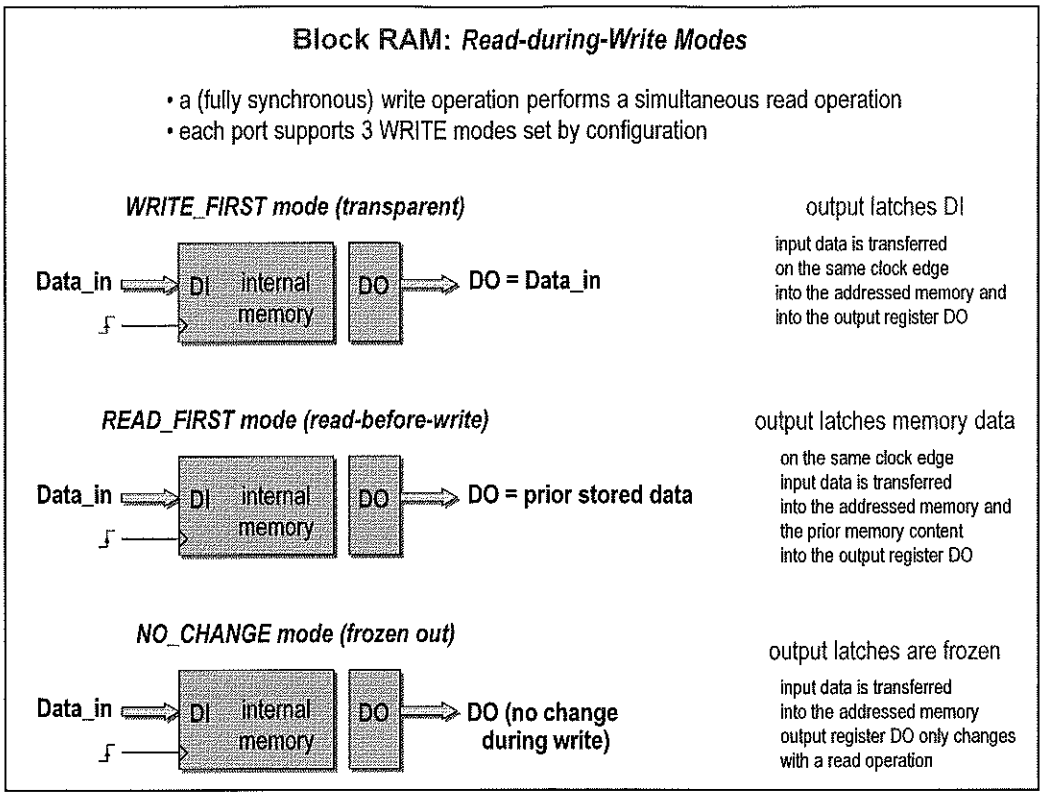
As a single-port RAM, the block SelectRAM has access to the 18 Kbit memory locations in any of the 2K x 9-bit, 1K x 18-bit, or 512 x 36-bit configurations and to 16 Kbit memory locations in any of the 16K x 1-bit, 8K x 2-bit, or 4K x 4-bit configurations. The advantage of the 9-bit, 18-bit and 36-bit widths is the ability to store a parity bit for each eight bits. Parity bits must be generated or checked externally in user logic. In such cases, the width is viewed as 8 + 1, 16 + 2, or 32 + 4. These extra parity bits are stored and behave exactly as the other bits, including the timing parameters. Video applications can use the 9-bit ratio of Virtex-II block SelectRAM memory to advantage.

Each block SelectRAM cell is a fully synchronous memory. Input data bus and output data bus widths are identical.

Dual-Port Configuration

As a dual-port RAM, each port of block SelectRAM has access to a common 18 Kbit memory resource. These are fully synchronous ports with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion.

If both ports are configured in either 2K x 9-bit, 1K x 18-bit, or 512 x 36-bit configurations, the 18 Kbit block is accessible from port A or B. If both ports are configured in either 16K x 1-bit, 8K x 2-bit. or 4K x 4-bit configurations, the 16 K-bit block is accessible from Port A or Port B. All other configurations result in one port having access to an 18 Kbit memory block and the other port having access to a 16 K-bit subset of the memory block equal to 16 Kbits.



Read/Write Operations

The Virtex-II block SelectRAM read operation is fully synchronous. An address is presented, and the read operation is enabled by control signals WEA and WEB in addition to ENA or ENB. Then, depending on clock polarity, a rising or falling clock edge causes the stored data to be loaded into output registers.

The write operation is also fully synchronous. Data and address are presented, and the write operation is enabled by control signals WEA or WEB in addition to ENA or ENB. Then, again depending on the clock input mode, a rising or falling clock edge causes the data to be loaded into the memory cell addressed.

A write operation performs a simultaneous read operation. Three different options are available, selected by configuration:

1. "WRITE_FIRST"

The "WRITE_FIRST" option is a transparent mode.Th e same clock edge that writes the data input (DI) into the memory also transfers DI into the output registers DO.

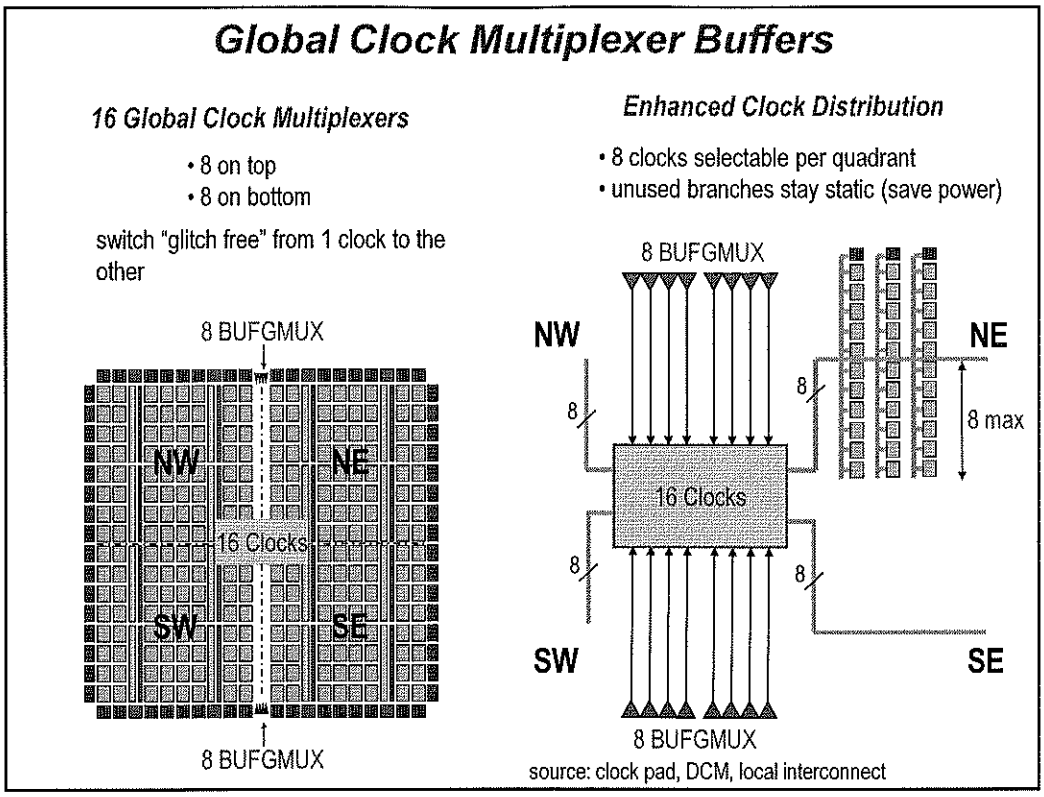
2. "READ_FIRST"

The "READ_FIRST" option is a read-before-write mode.

The same clock edge that writes data input (DI) into the memory also transfers the prior content of the memory cell addressed into the data output registers DO.

3. "NO_CHANGE"

The "NO_CHANGE" option maintains the content of the output registers, regardless of the write operation. The clock edge during the write mode has no effect on the content of the data output register DO. When the port is configured as "NO_CHANGE", only a read operation loads a new value in the output register DO.



The DCM and global clock multiplexer buffers provide a complete solution for designing high-speed clocking schemes.

Virtex-II devices have 16 clock input pins that can also be used as regular user I/Os. Eight clock pads are on the top edge of the device, in the middle of the array, and eight are on the bottom edge. The global clock multiplexer buffer represents the input to dedicated low-skew clock tree distribution in Virtex-II devices. Like the clock pads, eight global clock multiplexer buffers are on the top edge of the device and eight are on the bottom edge.

Each global clock buffer can either be driven by the clock pad to distribute a clock directly to the device, or driven by the Digital Clock Manager (DCM).

Each global clock buffer can also be driven by local interconnects. The DCM has clock output(s) that can be connected to global clock buffer inputs. Global clock buffers are used to distribute the clock to some or all synchronous logic elements (such as registers in CLBs and IOBs, and SelectRAM blocks. Eight global clocks can be used in each quadrant of the Virtex-II device.

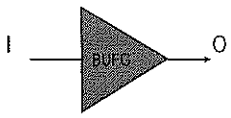
Clock distribution in Virtex-II devices:

In each quadrant, up to eight clocks are organized in clock rows. A clock row supports up to 16 CLB rows (eight up and eight down). For the largest devices a new clock row is added, as necessary.

Global Clocks: BUFGMUX

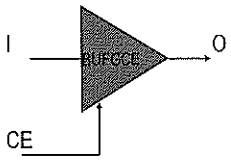
BUFG: Clock Buffer

- low skew clock distribution



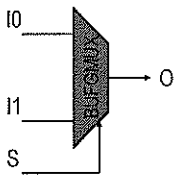
BUFGCE: Clock Buffer with Clock Enable

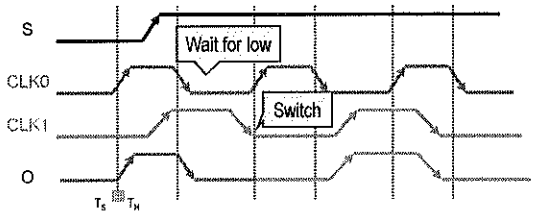
- CE = 0: block the clock (O = 0)
- CE = 1 (before active edge of I): clock pulse passes



BUFGMUX: Clock Multiplexer "Glitch-Free"

- switch from one clock to another glitch-free
- unrelated (asynchronous) clocks I0 and I1





No pulse width shorter than 1/2 of the period

Global clocks are driven by dedicated clock buffers (BUFG), which can also be used to gate the clock (BUFGCE) or to multiplex between two independent clock inputs (BUFGMUX).

BUFGCE

If the CE input is active (High) prior to the incoming rising clock edge, the following clock pulse passes through the clock buffer. Any level change of CE during the incoming clock High time has no effect.

If the CE input is inactive (Low) prior to the incoming rising clock edge, the following clock pulse does not pass through the clock buffer, and the output stays Low. Any level change of CE during the incoming clock High time has no effect.

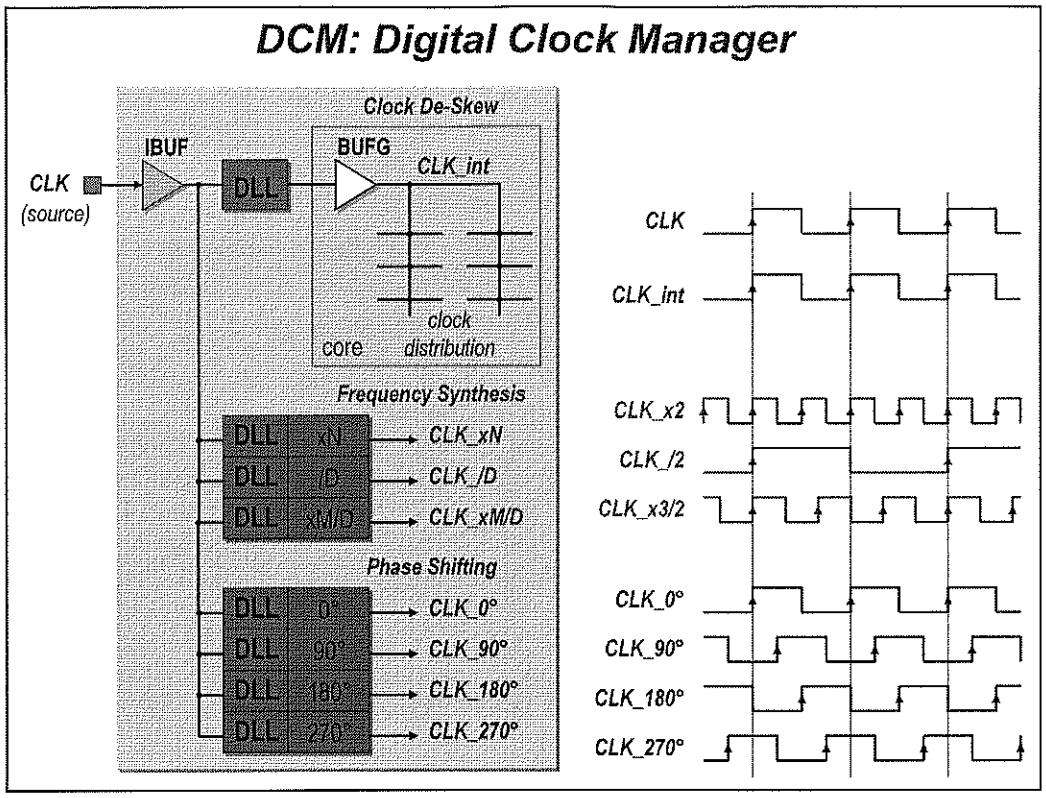
CE must not change during a short setup window just prior to the rising clock edge on the BUFGCE input I. Violating this setup time requirement can result in an undefined runt pulse output.

BUFGMUX

BUFGMUX can switch between two unrelated, even asynchronous clocks. Basically, a Low on S selects the I0 input, a High on S selects the I1 input. Switching from one clock to the other is done in such a way that the output High and Low time is never shorter than the shortest High or Low time of either input clock. As long as the presently selected clock is High, any level change of S has no effect.

If the presently selected clock is Low while S changes, or if it goes Low after S has changed, the output is kept Low until the other ("to-be-selected") clock has made a transition from High to Low. At that instant, the new clock starts driving the output.

The two clock inputs can be asynchronous with regard to each other, and the S input can change at any time, except for a short setup time prior to the rising edge of the presently selected clock; that is, prior to the rising edge of the BUFGMUX output O. Violating this setup time requirement can result in an undefined runt pulse output.



The Virtex-II DCM offers a wide range of powerful clock management features.

• **Clock De-skew**:T he DCM generates new system clocks (either internally or externally to the FPGA), which are phase-aligned to the input clock, thus eliminating clock distribution delays.

• **Frequency Synthesis**: The DCM generates a wide range of output clock frequencies, performing very flexible clock multiplication and division. Very flexible frequency synthesis provides a clock output frequency equal to any M/D ratio of the input clock frequency, where M and D are two integers. To generate de-skewed internal/ external clocks, each DCM can be used to eliminate clock distribution delay.

Frequency synthesis can be used in multiple clock domain applications:

• fast internal clock, slow external clock

• time domain multiplexing (use one circuit twice/clock cycle)

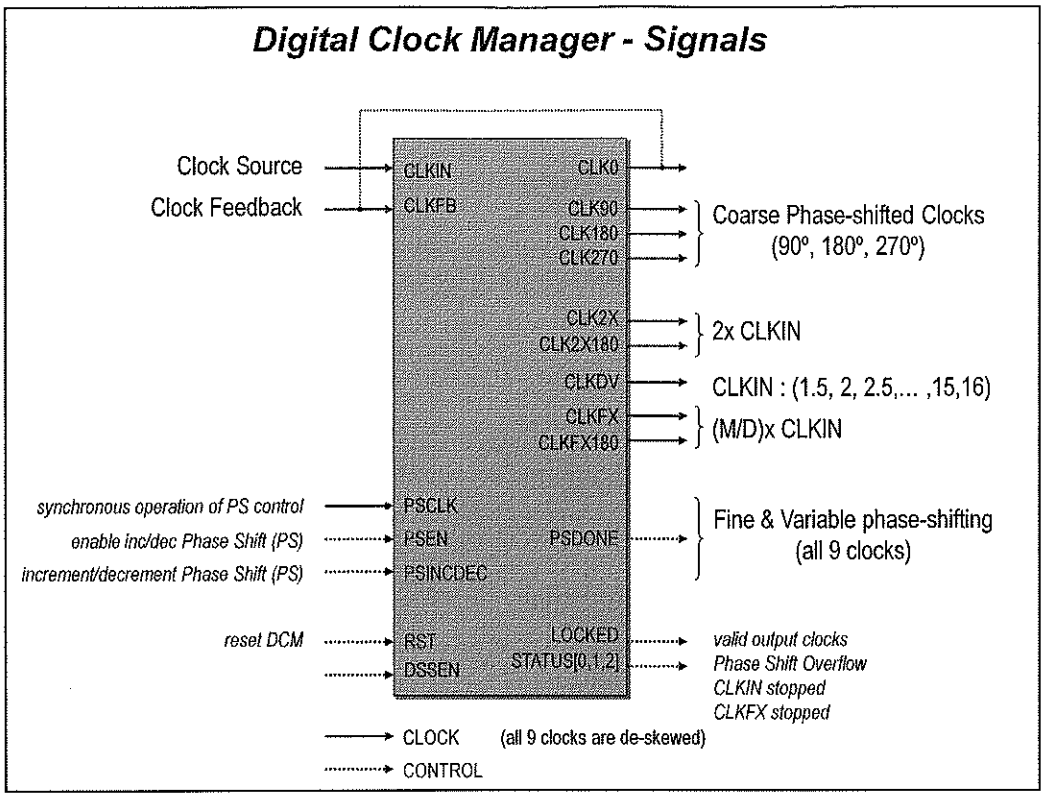
• **Phase Shifting**: The DCM provides coarse phase shifting (90-, 180-, and 270-degree phase-shifted versions of the output clocks). This can be used in clock multiplexed applications:

  • 0° - 180°: DDR (Double Data Rate)

  • 0° - 90° - 180° - 270° :QDR (Quadruple Data Rate)

Fine-grained phase shifting with dynamic phase shift control offers high-resolution phase adjustments in increments of 1/256 of the clock period.

The DCM utilizes fully digital delay lines allowing robust high-precision control of clock phase and frequency. It also utilizes fully digital feedback systems, operating dynamically to compensate for temperature and voltage variations during operation.



Up to 4 of the 9 DCM clock outputs can drive inputs to global clock buffers or global clock multiplexer buffers simultaneously. All DCM clock outputs can simultaneously drive general routing resources, including routes to output buffers.

- The CLK2X and CLK2X180 outputs double the clock frequency.
- The CLKDV output creates divided output clocks with division options of 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, and 16.
- The CLKFX and CLKFX180 outputs can be used to produce clocks at the following frequency:

FREQCLKFX = (M/D) * FREQCLKIN where M and D are two integers.

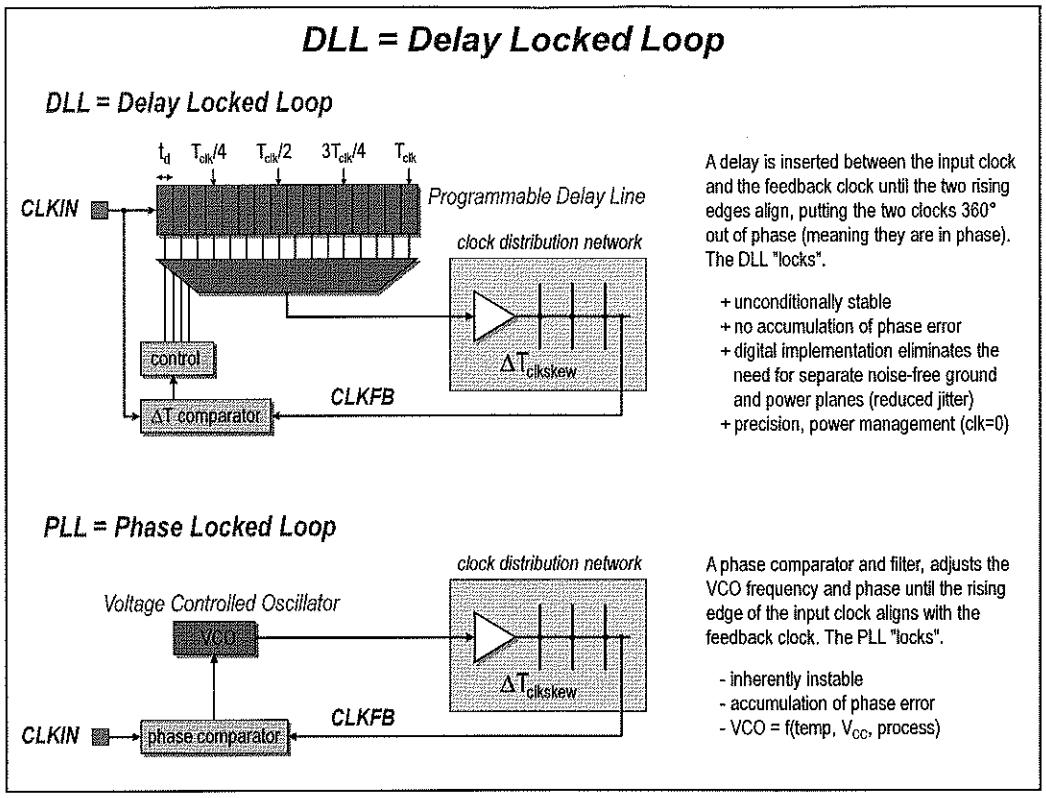
Very flexible frequency synthesis provides a clock output frequency equal to any M/D ratio of the input clock frequency, where M and D are two integers. To generate de-skewed internal/ external clocks, each DCM can be used to eliminate clock distribution delay.

- CLK2X180 is phase shifted 180 degrees relative to CLK2X.
- CLKFX180 is phase shifted 180 degrees relative to CLKFX.

The DCM has the following general control signals:

- RST input pin: resets the entire DCM
- LOCKED output pin: asserted High when all enabled DCM circuits have locked.
- STATUS output pins (active High): STATUS[0] = Phase Shift Overflow
  STATUS[1] = CLKIN Stopped
  STATUS[2]= CLKFX Stopped

The DCM can be configured to delay the completion of the Virtex-II configuration process until after the DCM has achieved lock. This guarantees that the chip does not begin operating until after the system clocks generated by the DCM have stabilized.



Clock De-Skew

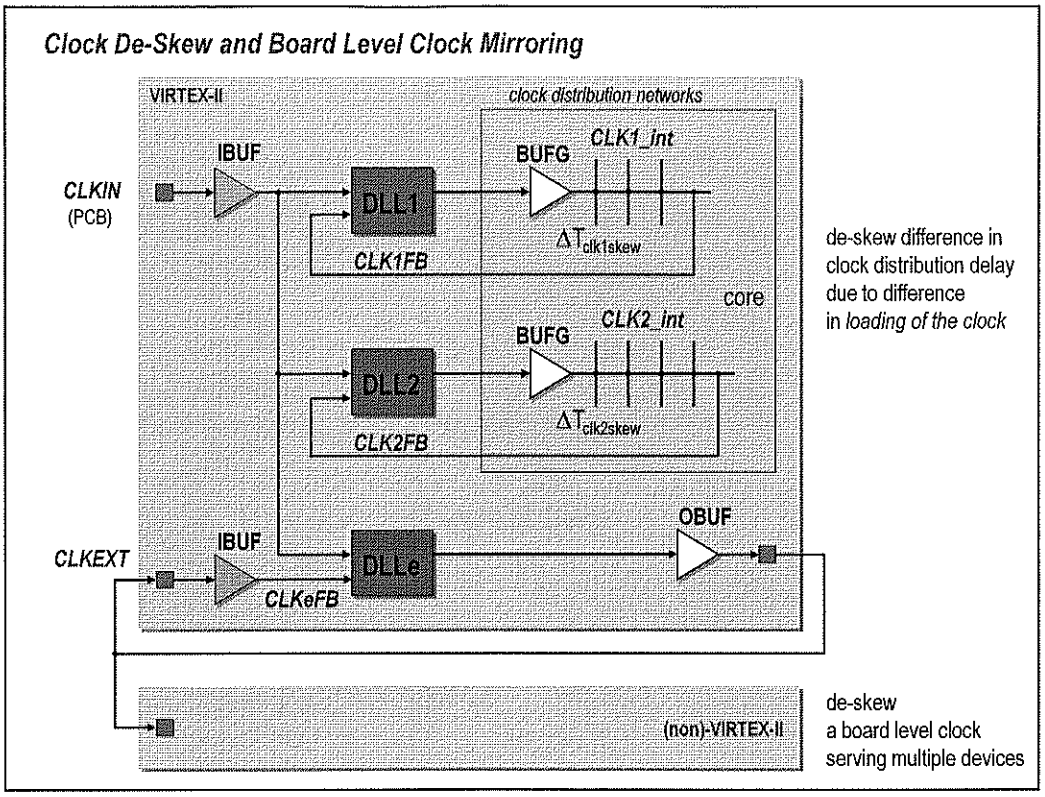
The DCM de-skews the output clocks relative to the input clock by automatically adjusting a digital delay line. Additional delay is introduced so that clock edges arrive at internal registers and block RAMs simultaneously with the clock edges arriving at the input clock pad. Alternatively, external clocks, which are also de-skewed relative to the input clock, can be generated for board-level routing. All DCM output clocks are phase-aligned to CLK0 and, therefore, are also phase-aligned to the input clock.

To achieve clock de-skew, the CLKFB input must be connected, and its source must be either CLK0 or CLK2X. Note that CLKFB must always be connected, unless only the CLKFX or CLKFX180 outputs are used and de-skew is not required.

The DCM contains a digitally-controlled feedback circuit (delay locked loop) that can completely eliminate clock distribution delays. Clock de-skew works as follows:

The incoming clock drives a long chain of delay elements (individual small buffers). A wide multiplexer selects any one of these buffers as an output. A controller drives the select inputs of this multiplexer. The phase detector in this controller compares the incoming clock signal (CLKIN) against a feedback input (CLKFB), which must be another version of the same clock signal, usually from the far end of the internal clock distribution network (but it can also be from an output pin).

The phase detector steers the controller to adjust the tap selection, and thus the through-delay in the DCM, in such a way that the two inputs to the phase comparator coincide.(This is a typical servo loop.) The tap controller adds exactly the right amount of delay to the clock distribution network to give it a total delay of one full clock period. For a repetitive clock signal, this effectively eliminates the clock distribution delay completely.

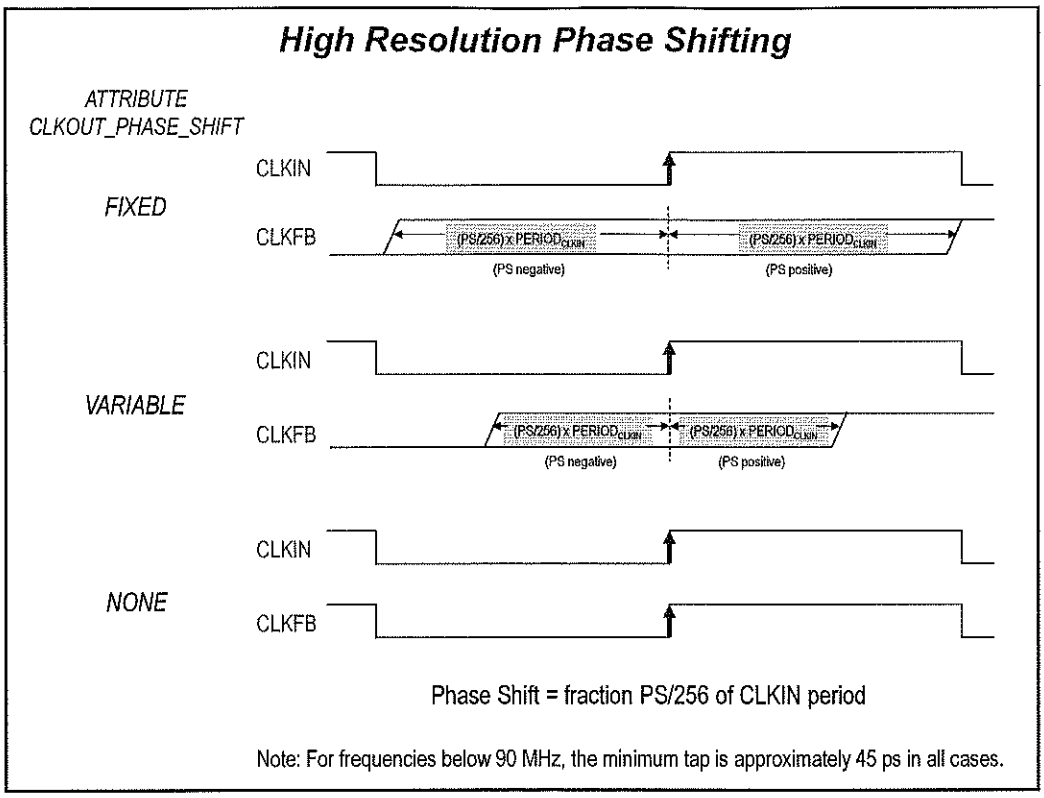


The DLL can also act as a clock mirror. By driving the DLL output off-chip and then back in again, the DLL can be used to de-skew a board level clock between multiple devices.

The clock mirroring scheme should be used for systems (such as PCI) that specify a loading/fan out limit on the system clock. This scheme can also be used when the clock source in a single board system is not capable of driving all the loads on the board.

While designing the board level route, ensure that the return net delay to the source equals the delay to the other chips involved.

Do not use the DLL output clock signals until after activation of the LOCKED signal. Prior to the activation of the LOCKED signal, the DLL output clocks are not valid and can exhibit glitches, spikes, or other spurious movement.



Phase Shifting

The DCM provides additional control over clock skew through either coarse or fine-grained phase shifting.

The CLK0, CLK90, CLK180, and CLK270 outputs are each phase shifted by 1/4 of the input clock period relative to each other, providing coarse phase control. Note that CLK90 and CLK270 are not available in high-frequency mode.

Fine-phase adjustment affects all nine DCM output clocks. When activated, the phase shift between the rising edges of CLKIN and CLKFB is a fixed or variable fraction of the input clock period.

In variable mode, the PHASE_SHIFT value can also be dynamically incremented or decremented as determined by PSINCDEC synchronously to PSCLK, when the PSEN input is active. The PHASE_SHIFT attribute is the numerator in the following equation:

Phase Shift (ns) = (PHASE_SHIFT/256) * PERIODCLKIN

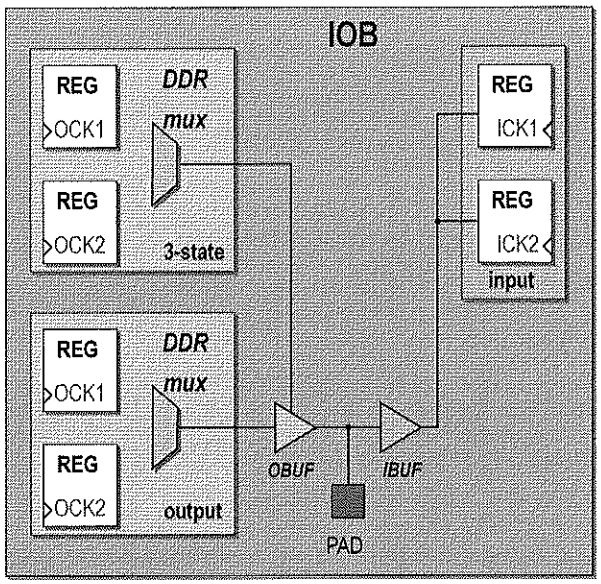
The full range of this attribute is always -255 to +255, but its practical range varies with CLKIN frequency, as constrained by the total delay achievable by the phase shift delay line (FINE_SHIFT_RANGE). Total delay is a function of the number of delay taps used in the circuit, the process, voltage, and temperature.

Absolute range (fixed mode) = ± FINE_SHIFT_RANGE

Absolute range (variable mode) = ± FINE_SHIFT_RANGE/2

The reason for the difference between fixed and variable modes is as follows. For variable mode to allow symmetric, dynamic sweeps from -255/256 to +255/256, the DCM sets the "zero phase skew" point as the middle of the delay line, thus dividing the total delay line range in half. In fixed mode, since the PHASE_SHIFT value never changes after configuration, the entire delay line is available for insertion into either the CLKIN or CLKFB path (to create either positive or negative skew).

IOB: Input/Output Block



- input, output and 3-state have DDR (Double Data Rate) registers
- *clock enable* for each group of DDR registers (ICE, OCE, TCE)
- separate *clocks* for I & O (ICK, OCK)
- shared *set and reset* (SR, REV) independent sync/async configuration independent set/reset configuration

IOBs are programmable and can be categorized as follows:

- Input block with an optional single-data-rate or double-data-rate (DDR) register
- Output block with an optional single-data-rate or DDR register, and an optional 3-state buffer, to be driven directly or through a single or DDR register
- Bidirectional block (any combination of input and output configurations)

These registers are either edge-triggered D-type flip-flops or level-sensitive latches.

IOBs support the following single-ended I/O standards:LVTT L, LVCMOS (3.3V, 2.5V, 1.8V, and 1.5V), PCI-X compatible (133 MHz and 66 MHz) at 3.3V, PCI compliant (66 MHz and 33 MHz) at 3.3V, ...

The digitally controlled impedance (DCI) I/O feature automatically provides on-chip termination for each I/O element.

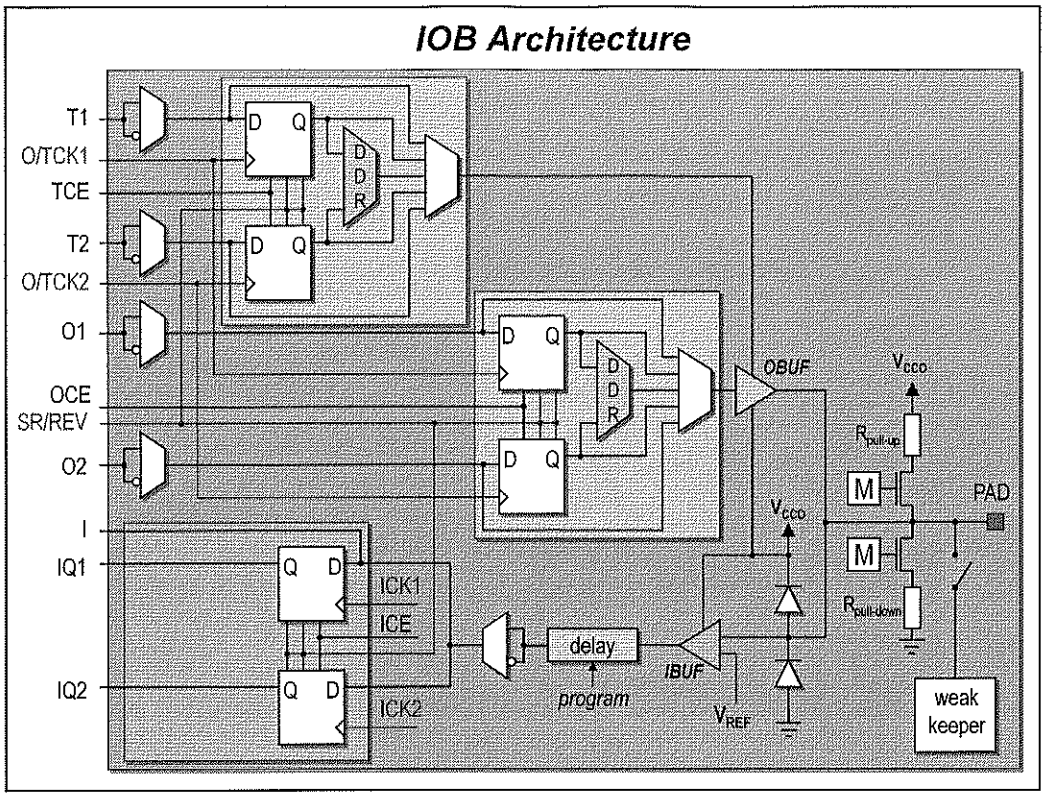
The IOB elements also support the following differential signaling I/O standards: LVDS, BLVDS (Bus LVDS), ULVDS, LDT, LVPECL.

Virtex-II I/O blocks (IOBs) are provided in groups of two or four on the perimeter of each device. Each IOB can be used as input and/or output for single-ended I/Os. Two adjacent IOBs can be used as a differential pair. A differential pair is always connected to the same switch matrix to access the routing resources

Logic Resources

IOB blocks include six storage elements. Each storage element can be configured either as an edge-triggered D-type flip-flop or as a level-sensitive latch.

On the input, output, and 3-state path, one or two DDR registers can be used.



Each group of two registers has a clock enable signal (ICE for the input registers, OCE for the output registers, and TCE for the 3-state registers). The clock enable signals are active High by default. If left unconnected, the clock enable for that storage element defaults to the active state.

Each IOB block has common synchronous or asynchronous set and reset (SR and REV signals).

SR forces the storage element into the state specified by the SRHIGH or SRLOW attribute. SRHIGH forces a logic "1". SRLOW forces a logic "0". When SR is used, a second input (REV) forces the storage element into the opposite state. The reset condition predominates over the set condition. The initial state after configuration or global initialization state is defined by a separate INIT0 and INIT1 attribute. By default, the SRLOW attribute forces INIT0, and the SRHIGH attribute forces INIT1. For each storage element, the SRHIGH, SRLOW, INIT0, and INIT1 attributes are independent. Synchronous or asynchronous set / reset is consistent in an IOB block. All the control signals have independent polarity. Any inverter placed on a control input is automatically absorbed.

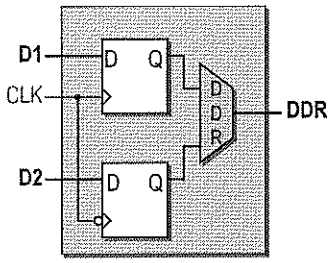
Each device pad has optional pull-up and pull-down in all SelectI/O-Ultra configurations. Values of the optional pull-up and pull-down resistors are in the range 10 - 60 KΩ,. Each device pad has an optional weak-keeper in LVTTL, LVCMOS, and PCI SelectI/O-Ultra configurations. The clamp diode is always present, even when power is not.

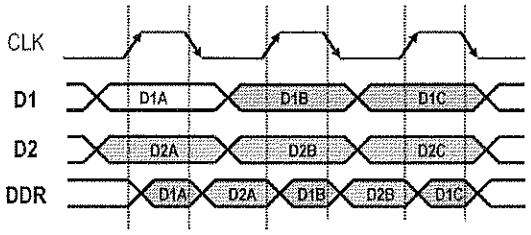
The optional weak-keeper circuit is connected to each user I/O pad. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low. If the pin is connected to a multiple-source signal, the weak-keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way eliminates bus chatter; pull-up or pull-down override the weak-keeper circuit.

All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients.

Double Data Rate Registers

Two bits of data are carried on each DDR output during one period of the clock.

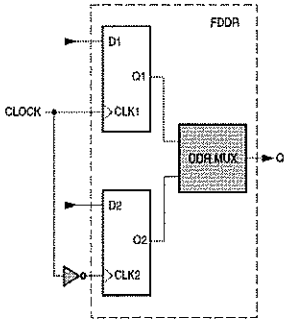


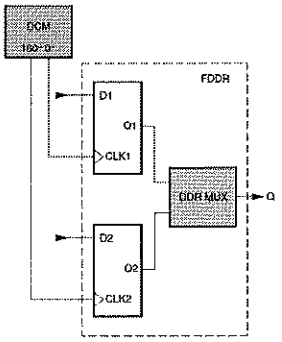


The bandwidth of a data port is increased by changing the signals on both edges of a clock.

DDR registers can be clocked by:
- CLK and not (CLK)
- CLK0 and CLK180 DLL outputs

A slight shift in duty cycle can dramatically decrease performance.



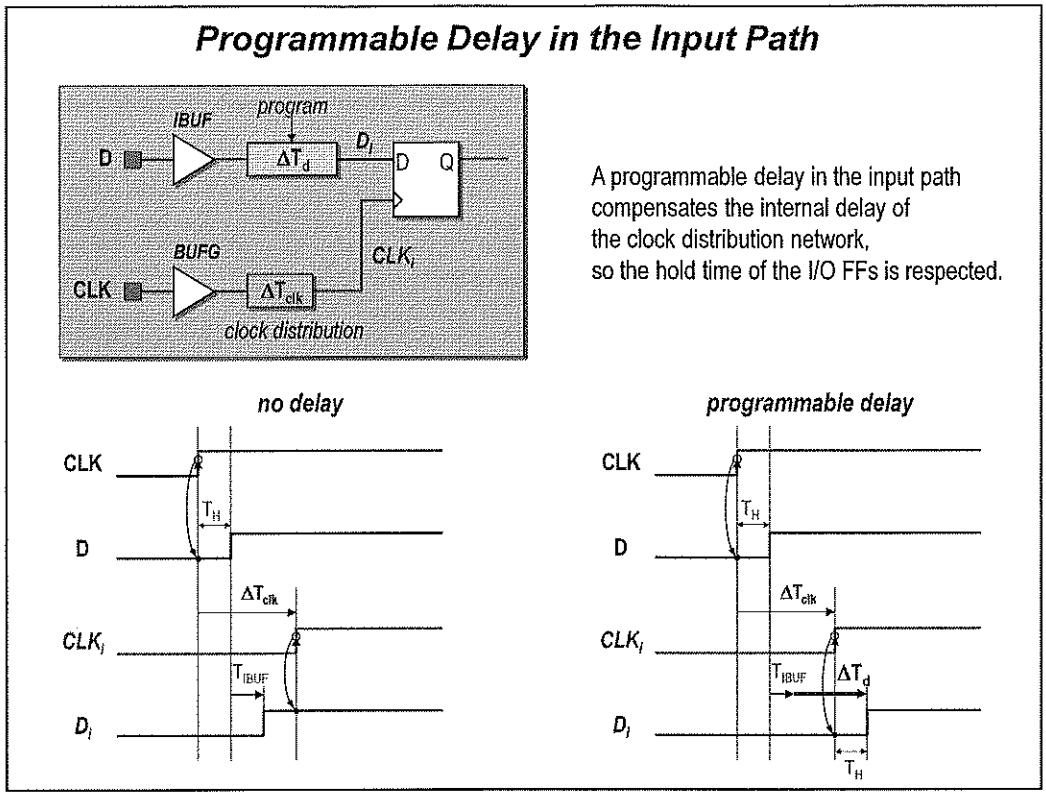


50/50 duty cycle clock

Double data rate is directly accomplished by the two registers on each path, clocked by the rising edges (or falling edges) from two different clock nets.Th e two clock signals are generated by the DCM and must be 180 degrees out of phase. A single clock triggers one register on a Low to High transition and a second register on a High to Low transition.

There are two input, output, and 3-state data signals, each being alternately clocked out.

The DDR mechanism can be used to mirror a copy of the clock on the output (like Manchester coding). This is useful for propagating a clock along the data that has an identical delay. It is also useful for multiple clock generation, where there is a unique clock driver for every clock load. Virtex-II devices can produce many copies of a clock with very little skew.



A programmable delay in the input path compensates the internal delay of the clock distribution network, so the hold time of the I/O FFs is respected.

This optional delay element at the D-input of the storage element eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the Virtex-II device, and when used, assures that the pad-to-pad hold time is zero.

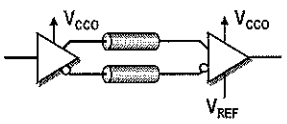
I/O Standards

LVPECL = Low Voltage Positive Emitter-Coupled Logic
LDT = Lightning Data Transport (O=current driver)
LVDS = Low Voltage Differential Signaling (O=current driver)
BLVDS = Bus LVDS (bidirectional)

LVTTL = Low Voltage TTL (O=push-pull)
LVCMOS = Low Voltage CMOS
PCI = Peripheral Component Interface
GTL = Gunning Transistor Logic (O=open drain)
GTLP = GTL Plus (Pentium Pro Processor)
HSTL = High Speed Transceiver Logic (O=push-pull)
SSTL2 = Stub Series Terminated Logic for 2.5 V (O=push-pull)
SSTL3 = Stub Series Terminated Logic for 3.3 V (O=push-pull)

differential I/O

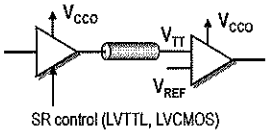
I/O Standard	V_{REF}	V_{OD} (V)
LVPECL_33	3.3	0.490 - 1.220
LDT_25	2.5	0.430 - 0.670
LVDS_33	3.3	0.250 - 0.400
LVDS_25	2.5	0.250 - 0.400
LVDSEXT_33	3.3	0.330 - 0.700
LVDSEXT_25	2.5	0.330 - 0.700
BLVDS_25	2.5	0.250 - 0.450
ULVDS_25	2.5	0.430 - 0.670



single-ended I/O

I/O Standard	V_{CCO}	V_{REF}	V_{TT}
LVTTL	3.3	N/A	N/A
LVCMOS33	3.3	N/A	N/A
LVCMOS25	2.5	N/A	N/A
LVCMOS18	1.8	N/A	N/A
LVCMOS15	1.5	N/A	N/A
PCI33_3	3.3	N/A	N/A
PCI66_3	3.3	N/A	N/A
PCI-X	3.3	N/A	N/A
GTL	*	0.8	1.2
GTLP	*	1.0	1.5
HSTL_I	1.5	0.75	0.75
HSTL_II	1.5	0.75	0.75
HSTL_III	1.5	0.9	1.5
HSTL_IV	1.5	0.9	1.5
HSTL_I_18	1.8	0.9	0.9
HSTL_II_18	1.8	0.9	0.9
HSTL_III_18	1.8	1.1	1.8
HSTL_IV_18	1.8	1.1	1.8
SSTL2_I	2.5	1.25	1.25
SSTL2_II	2.5	1.25	1.25
SSTL3_I	3.3	1.5	1.5
SSTL3_II	3.3	1.5	1.5
AGP-2X/AGP	3.3	1.32	N/A

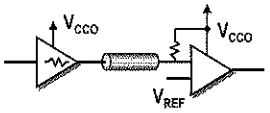
V_{TT} = Board Termination Voltage



SR control (LVTTL, LVCMOS)

DCI single-ended I/O

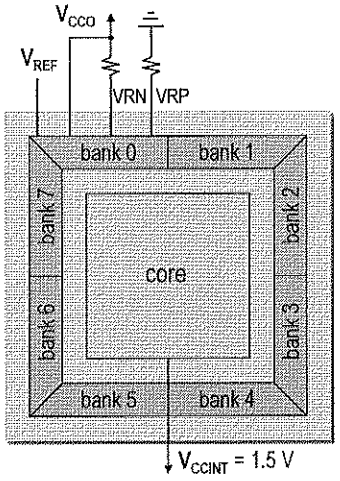
I/O Standard	V_{CCO}	V_{REF}	Termination
LVDCI_33[1]	3.3	N/A	Series
LVDCI_DV2_33[1]	3.3	N/A	Series
LVDCI_25[1]	2.5	N/A	Series
LVDCI_DV2_25[1]	2.5	N/A	Series
LVDCI_18[1]	1.8	N/A	Series
LVDCI_DV2_18[1]	1.8	N/A	Series
LVDCI_15[1]	1.5	N/A	Series
LVDCI_DV2_15[1]	1.5	N/A	Series
GTL_DCI	1.2	0.8	*Single*
GTLP_DCI	1.5	1.0	*Single*
HSTL_I_DCI	1.5	0.75	Split
HSTL_II_DCI	1.5	0.75	Split
HSTL_III_DCI	1.5	0.9	*Single*
HSTL_IV_DCI	1.5	0.9	*Single*
HSTL_I_DCI_18	1.8	0.9	Split
HSTL_II_DCI_18	1.8	0.9	Split
HSTL_III_DCI_18	1.8	1.08	*Single*
HSTL_IV_DCI_18	1.8	1.08	*Single*
SSTL2_I_DCI[2]	2.5	1.25	Split
SSTL2_II_DCI[2]	2.5	1.25	Split
SSTL3_I_DCI[2]	3.3	1.5	Split
SSTL3_II_DCI[2]	3.3	1.5	Split

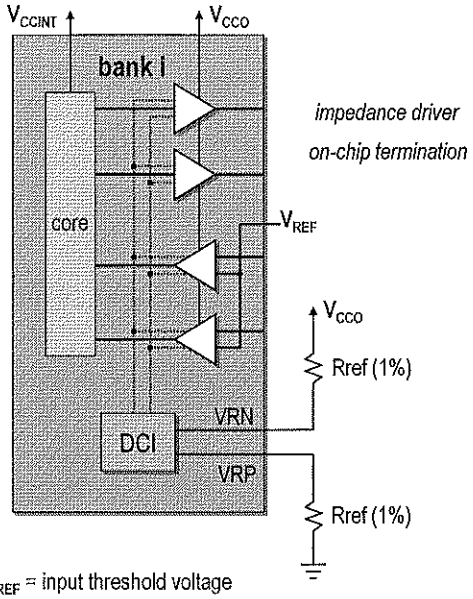


IOB blocks are designed for high performances I/Os, supporting 19 single-ended standards, as well as differential signaling with LVDS, LDT, Bus LVDS, and LVPECL.

Virtex-II IOB blocks feature SelectI/O-Ultra inputs and outputs that support a wide variety of I/O signaling standards. In addition to the internal core supply voltage (V_{CCINT} = 1.5V), output driver supply voltage (V_{CCO}) is dependent on the I/O standard (see Table). An auxiliary supply voltage (V_{CCAUX} = 3.3 V) is required, regardless of the I/O standard used.

DCI: Digitally Controlled Impedance





DCI provides controlled impedance drivers and on-chip termination for single-ended and differential I/Os. This eliminates the need for external resistors, and improves signal integrity (prevent reflections).

IOBs are grouped in banks.
DCI operates independently on each bank.

V_{REF} = input threshold voltage
V_{CCO} = power-supply for the output drivers (per bank).
V_{CCINT} = power-supply for the internal core logic
VRN(P) = DCI voltage reference resistor of N(P) transistor

Today's chip output signals with fast edge rates require termination to prevent reflections and maintain signal integrity. High pin count packages (especially ball grid arrays) can not accommodate external termination resistors.

Virtex-II XCITE DCI provides controlled impedance drivers and on-chip termination for single-ended and differential I/Os. This eliminates the need for external resistors, and improves signal integrity. The DCI feature can be used on any IOB by selecting one of the DCI I/O standards.

When applied to inputs, DCI provides input parallel termination. When applied to outputs, DCI provides controlled impedance drivers (series termination) or output parallel termination.

Some of the I/O standards require V_{CCO} and V_{REF} voltages. These voltages are externally supplied and connected to device pins that serve groups of IOB blocks, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank. Eight I/O banks result from dividing each edge of the FPGA into two banks. Each bank has multiple V_{CCO} pins, all of which must be connected to the same voltage. This voltage is determined by the output standards in use.

DCI operates independently on each I/O bank. When a DCI I/O standard is used in a particular I/O bank, external reference resistors must be connected to two dual-function pins on the bank. These resistors, voltage reference of N transistor (VRN) and the voltage reference of P transistor (VRP) are shown in the figure.

When used with a terminated I/O standard, the value of resistors are specified by the standard (typically 50 Ω).Whe n used with a controlled impedance driver, the resistors set the output impedance of the driver within the specified range (25 Ω to 100 Ω). For all series and parallel terminations, the reference resistors must have the same value for any given bank. One percent resistors are recommended.

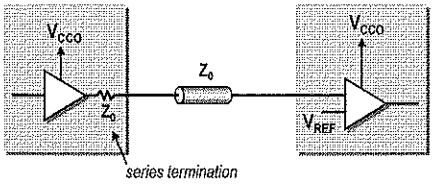
The DCI system adjusts the I/O impedance to match the two external reference resistors, or half of the reference resistors, and compensates for impedance changes due to voltage and/or temperature fluctuations. The adjustment is done by turning parallel transistors in the IOB on or off.

DCI: *Controlled Impedance Drivers and Terminations*

Series Termination

V_{CCO}	DCI	DCI Half Z
3.3 V	LVDCI_33	LVDCI_DV2_33
2.5 V	LVDCI_25	LVDCI_DV2_25
1.8 V	LVDCI_18	LVDCI_DV2_18
1.5 V	LVDCI_15	LVDCI_DV2_15

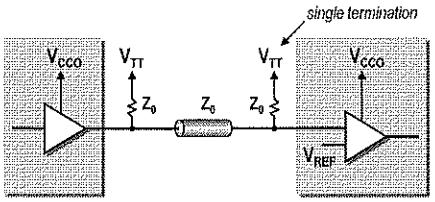
on-chip termination



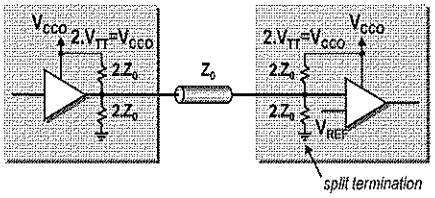
Parallel (Shunt) Termination

I/O Standard	External Termination	On-Chip Termination
HSTL Class I	HSTL_I	HSTL_I_DCI
HSTL Class II	HSTL_II	HSTL_II_DCI
HSTL Class III	HSTL_III	HSTL_III_DCI
HSTL Class IV	HSTL_IV	HSTL_IV_DCI
SSTL3 Class I	SSTL3_I	SSTL3_I_DCI(1)
SSTL3 Class II	SSTL3_II	SSTL3_II_DCI(1)
SSTL2 Class I	SSTL2_I	SSTL2_I_DCI(1)
SSTL2 Class II	SSTL2_II	SSTL2_II_DCI(1)
GTL	GTL	GTL_DCI
GTLP	GTLP	GTLP_DCI

external termination



on-chip termination



on-chip termination advantages:

- reduced board routing and component count
- improve signal integrity by eliminating stub reflection

Output Driver

DCI can be used to provide a buffer with a controlled output impedance. It is desirable for this output impedance to match the transmission line impedance (Z).

For controlled impedance output drivers, the impedance can be adjusted either to match the reference resistors or half the resistance of the reference resistors.

DCI can configure output drivers to be the following types:

1. Controlled Impedance Driver (Source Termination)
2. Controlled Impedance Driver with Half Impedance (Source Termination)

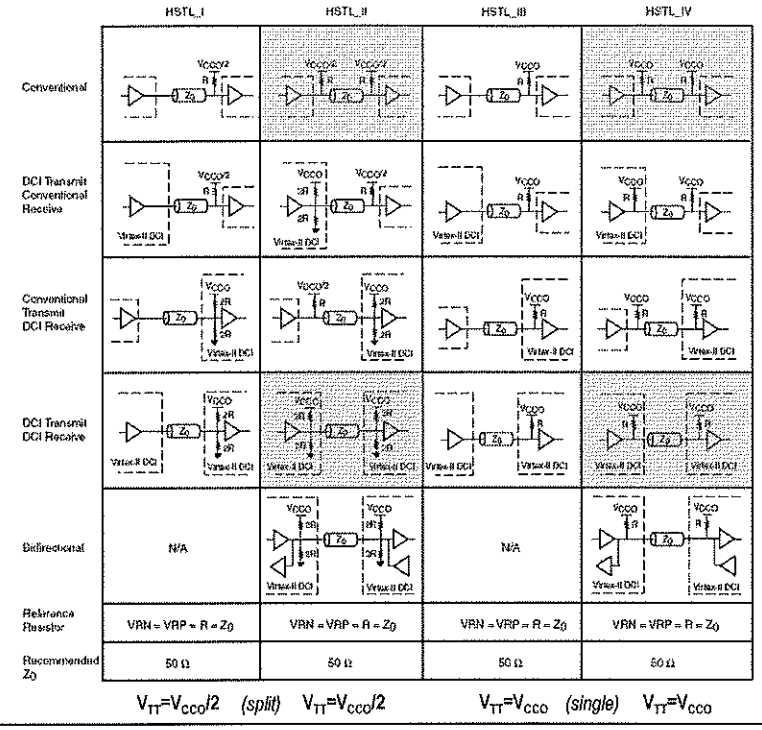
Inputs

For on-chip termination, the termination is always adjusted to match the reference resistors.

It can also configure inputs to have he following types of on-chip terminations:

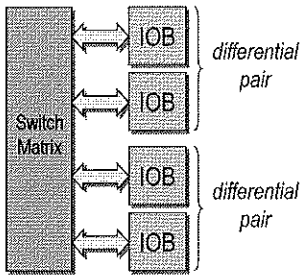
1. Input termination to V_{CCO} (Single Termination)
2. Input termination to $V_{CCO}/2$ (Split Termination, Thevenin equivalent)

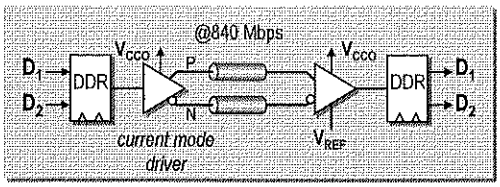
HSTL: *High Speed Transceiver Logic*

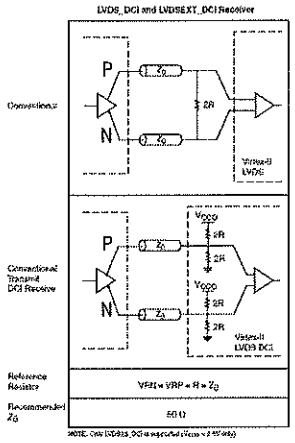


general purpose high speed bus
- 1.5V, 1.8 V
- sponsored by IBM
- standard: EIA/JESD8-6
- ⟊ = differential amplifier
- O= push-pull

LVDS: *Low Voltage Differential Signaling*







DC Specification, V_{DD} = 2.5V

DC Parameter	Conditions	MIN	TYP	MAX	Units
Output High Voltage for P and N	2R = 100 Ω	-	1.38	1.6	V
Output Low Voltage for P and N	2R = 100 Ω	0.90	1.03	-	V
Differential-Mode Output (P - N)	2R = 100 Ω	250	350	450	mV
Common-Mode Output (P + N)/2	2R = 100 Ω	1.125	1.25	1.375	V
Differential-Mode Input (P - N)	CM input = 1.25 V	100	350	-	mV
Common-Mode Input (P + N)/2	DM input = ±350 mV	0.25	1.25	2.25	V

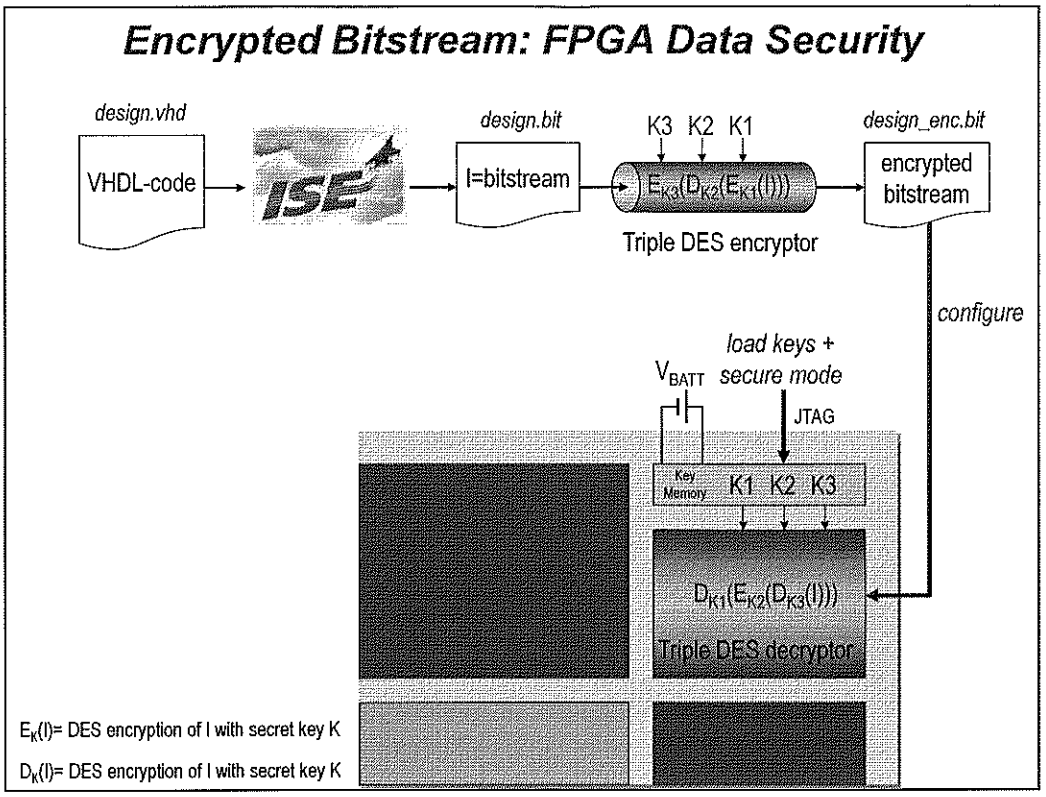
IOBs are provided in groups of 2 or 4 on the perimeter of each Virtex-II device. Two IOBs can be used as a differential pair. A differential pair is always connected to the same switch matrix. Differential I/Os must use the same clock.

LVDS provides higher noise immunity than single-ended techniques, allowing for higher transmission speeds, smaller signal swings, lower power consumption, and less electro-magnetic interference than single-ended signaling. Differential data can be transmitted at these rates using inexpensive connectors and cables. LVDS provides robust signaling for high-speed data transmission between chassis, boards, and peripherals using standard ribbon cables and IDC connectors with 100 mil header pins.Point -to-point LVDS signaling is possible at speeds of up to 622 Mb/s.

An LVDS driver on the left drives the two 50 Ω transmission lines into an LVDS receiver on the right. The P and N outputs of the LVDS driver pass to the corresponding inputs of the LVDS receiver. The two 50 Ω single-ended transmission lines can be microstrip, stripline, or a 100 Ω differential twisted pair or similar balanced differential transmission line. A 100 Ω resistor R_T terminates the P and N signals.

LVDS uses a current-mode driver, behaving like two equal and opposite current sources with a high output impedance. LVDS outputs typically drive ±3.5 mA to flow through the 100 Ω resistor R_T generating a ±350 mV voltage swing (P - N).This results in ≈1.2 mW of power delivered to the load.

The terms "common-mode voltage" and "offset voltage" refer to the average of P and N, (P+N)/2. LVDS has a typical output common-mode voltage of 1.25 V, determined by the LVDS driver.



Virtex-II devices have an on-chip decryptor using one or two sets of three keys for triple-key Data Encryption Standard (DES) operation. Xilinx software tools offer an optional encryption of the configuration data (bitstream) with a triple-key DES determined by the designer.

The keys are stored in the FPGA by JTAG instruction and retained by a battery connected to the V_{BATT} pin, when the device is not powered. Virtex-II devices can be configured with the corresponding encrypted bitstream, using any of the configuration modes described previously.

Virtex-II solves design security issue for FPGAs

- Bitstream secured using triple DES (3x56 bit keys)
- Prevents SRAM FPGA design theft
- Enables protected customer-specific chip-sets

2 banks of 3 DES keys, allows 2 users with different keys

All configuration methods (Serial, JTAG) support encryption at full speed (50MHz @ 8 bits wide)

PROCEDURE

1. Encrypt your bitstream with triple DES using 56 bit keys
2. Load 1 or 2 banks of keys into SRAM registers using JTAG
3. (Optional) Read back values of keys to ensure they are correct
4. Enter secure mode in FPGA by issuing JTAG instruction
   - Allows one encrypted bitstream write, locks out all subsequent bitstream reads and writes
   - Prevents reading or writing of keys
5. Load encrypted bitstream into FPGA specifying which triple DES key bank to use for decryption
6. Operate chip as normal FPGA (without possibility of reading and writing bitstreams)

Virtex-II Packaging

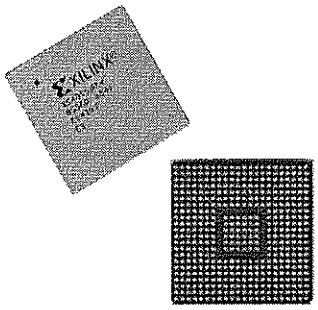
- Wire-bond packages
- Flip-chip packages
  - Higher device I/O count
  - Higher thermal capacity
- Ball-grid arrays:
  - FGxxx: wire-bond fine-pitch BGA (1.00 mm pitch)
  - BGxxx: wire-bond BGA (1.27 mm pitch)
  - FFxxx: flip-chip fine-pitch BGA (1.00 mm pitch)
  - BFxxx: flip-chip BGA (1.27 mm pitch)

Offerings include ball grid array (BGA) packages with 0.80 mm, 1.00 mm, and 1.27 mm pitches. In addition to traditional wire-bond interconnects, flip-chip interconnect is used in some of the BGA offerings. The use of flip-chip interconnect offers more I/Os than is possible in wire-bond versions of the similar packages. Flip-chip construction offers the combination of high pin count with high thermal capacity.

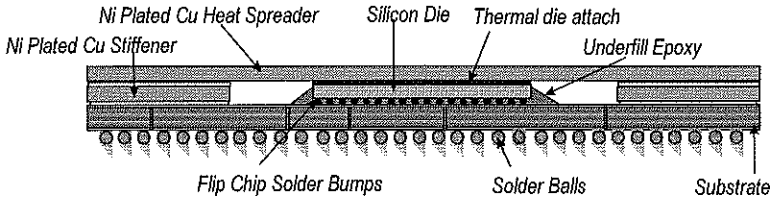
Flip-Chip and Wire-Bond Ball Grid Array (BGA) Packages in Three Standard Fine Pitches (0.80 mm, 1.00 mm, and 1.27 mm):

- CS denotes wire-bond chip-scale ball grid array (BGA) (0.80 mm pitch).
- FG denotes wire-bond fine-pitch BGA (1.00 mm pitch).
- FF denotes flip-chip fine-pitch BGA (1.00 mm pitch).
- BG denotes standard BGA (1.27 mm pitch).
- BF denotes flip-chip BGA (1.27 mm pitch).

Flip-Chip Packaging: Best Thermals & Small Package



- Better electrical performance due to improved supply voltage distribution to core
- Dissipate up to 30 Watts
- 2x more I/Os than SBGA
- Higher frequency switching with better noise control



Flip-chip construction offers the combination of high pin count with high thermal capacity.

Virtex-II Packaging: Overview

Device XC2V	40	80	250	500	1000	1500	2000	3000	4000	6000	8000
Max user I/Os	88	120	200	264	432	528	624	720	912	1,104	(1296) 1108
CS144	88	92	92								
FG256	88	120	172	172	172						
FG456			200	264	324						
FG676						392	456	484			
FF896					432	528	624				
FF1152								720	824	824	824
FF1517									912	1,104	1,108
BG575					328	392	408				
BG728							456	516			
BF957							624	684	684	684	684

- **FF** and **BF** are flip-chip ball grid arrays packages
- Pinout compatibility inside same color rectangle

Wire-bond and flip-chip packages are available. The table shows the maximum possible number of user I/Os for all device/package (wire-bond and flip-chip) combinations.

The number of I/Os per package include all user I/Os except the 15 control pins (CCLK, DONE, M0, M1, M2, PROG_B, PWRDWN_B, TCK, TDI, TDO, TMS, HSWAP_EN, DXN, DXP, and RSVD) and VBATT.

- CS denotes wire-bond chip-scale ball grid array (BGA) (0.80 mm pitch).
- FG denotes wire-bond fine-pitch BGA (1.00 mm pitch).
- FF denotes flip-chip fine-pitch BGA (1.00 mm pitch).
- BG denotes standard BGA (1.27 mm pitch).
- BF denotes flip-chip BGA (1.27 mm pitch).