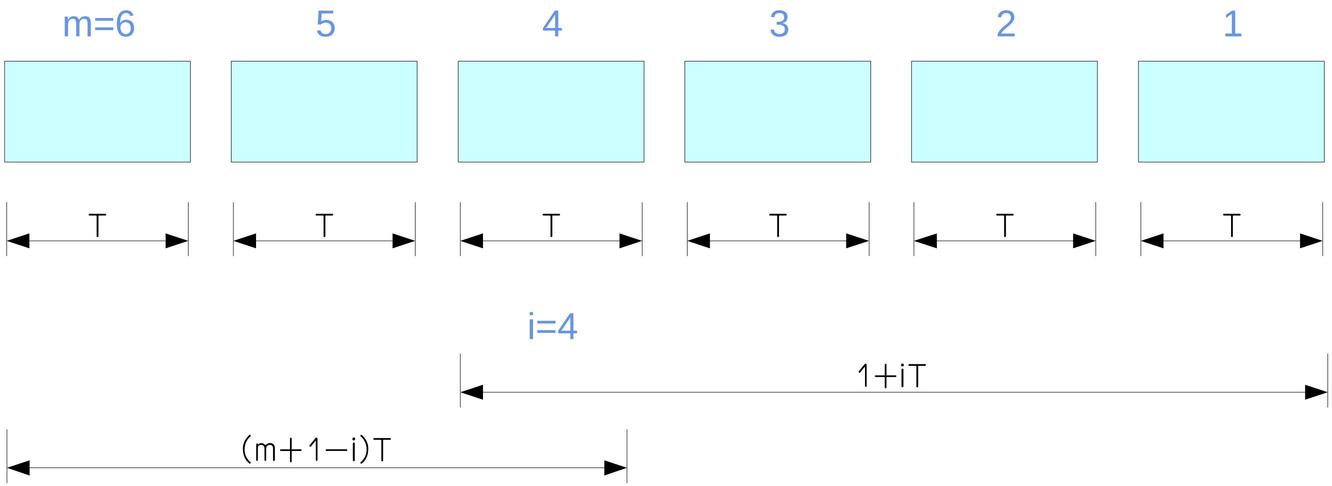
FPGA Variable Block Adder (1C)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Delay model



$$y_i = min\{1 + iT, 1 + (m + 1 - i)T\}$$

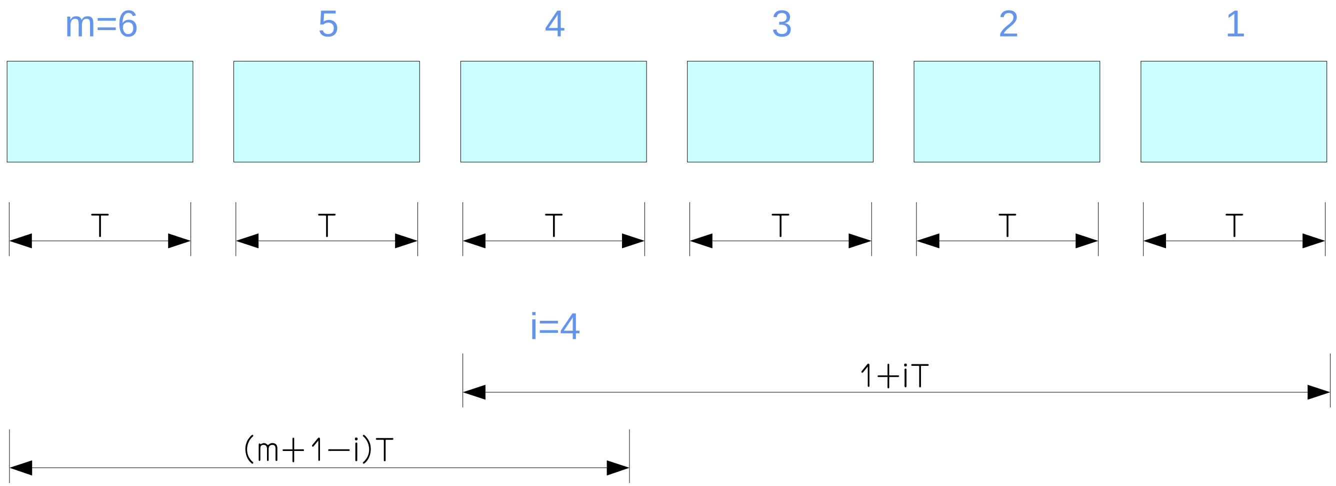
$$y_{1,} \ldots, y_m$$

$$0 \le x_i \le y_i, i = 1, \ldots, m$$

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

$$\sum_{i=1}^{m} x_i = n$$

Delay model



$$y_i = min\{1 + iT, 1 + (m + 1 - i)T\}$$

$$y_{1,} \ldots, y_m$$

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

Given *m*, an optimal division of the carry chain into groups
Can be obtained as follows
Let

$$y_i = min\{1 + iT, 1 + (m+1-i)T\}$$

Given $y_{1,} \ldots, y_m$ solve the minimization problem

$$\min_x max\{x_{1,} \ldots, x_n\}$$

Subject to

$$0 \le x_i \le y_i, i = 1, \ldots, m$$

And

$$\sum_{i=1}^{m} x_i = n$$

Any solution $x_{1,} \ldots, x_m$ gives optimal group sizes
for a division of the carry chain

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

The x’s can be computed iteratively as follows:

Initially take $x_1 = x_m = 0$

At each iteration, increase as many of the x’s as possible by one unit, without violating the constraints

$$0 \le x_i \le y_i, i = 1, \ldots, m \qquad \sum_{i=1}^{m} x_i \le n$$

An easy calculation shows that

$$\sum_{i=1}^{m} y_i = m + \frac{1}{2} mT + \frac{1}{4} m^2 T + \left(1 - (-1)^m\right) \frac{1}{8} T \ge n$$

Thus, at some iteration, we have $\sum_{i=1}^{m} x_i = n$ and
The algorithm terminates

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

For n=32, we have m=7, (y1, y2, y3, y4, y5, y6, y7) = (3,5,7,9,7,5,3)
The above algorithm gives (x1, x2, x3, x4, x5, x6, x7) = (3,5,5,6,5,5,3)

A carry chain divided in this way has maximum delay D = mT =14
Since one unit of delay is 0.8ns, the maximum delay for 32-bit carry chain
is D = 14*0.8ns = 11.2ns
This time involves only the delay in the carry chain

It is easy to check that this is also the delay for a chain divided into groups of sizes 1,3,5,7,7,5,3,1.
Thus this is also an optimal subdivision

The worst case delay includes the time needed to generate p_i and g_i signals
Delay of the carry chain, and the time for producing last sum bit s_n

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

Implement it with a string of multiplexers

The multiplexer cell is designed as very fast

Multiplexers are designed
as very fast structures using buffered pass gates and
in this sense are similar to the Manchester carry chain
which has been shown to be
the most effective implementation of a carry chain

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

The implementation of a single carry block is done
by mixing a 4 to 1 multiplexer (actually used as a 3 to 1)

In the last stage with a string of 2 to 1 multiplexers

a carry bypass is connected to inputs 3 and 4
of the 4:1 multiplexer (group carry multiplexer)
and the selection of the carry bypass is activated
by the NAND gate singaling when the condition
for group propagate is reached and
activating the group multiplexer in turn.

Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Delay model

The32-bit implementation of the VBA adder is obtained
By connecting the groups of the sizes calculated
For the full length of n=32 bits

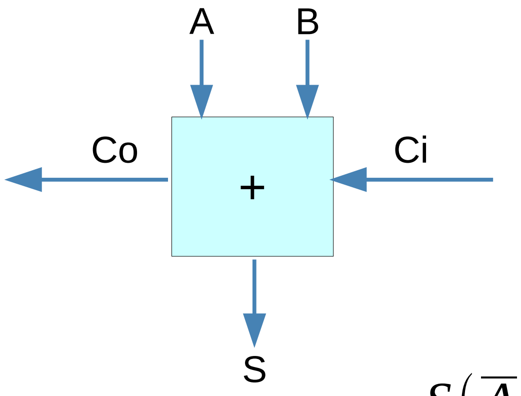
To increase the speed further we used a faster inverting version
Of the multiplexer, alternating between Ci and Cb_i signals

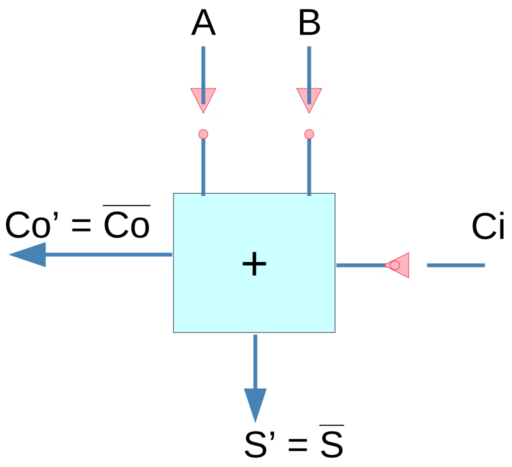
Oklobdzija: High-Speed VLSI arithmetic units : adders and multipliers

Inverting FA inputs

X	Y	Cin	Cout	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

$\overline{X}$	$\overline{Y}$	$\overline{Cin}$	$\overline{Cout}$	$\overline{S}$
1	1	1	1	1
1	0	1	1	0
0	1	1	1	0
0	0	1	0	1
1	1	0	1	0
1	0	0	0	1
0	1	0	0	1
0	0	0	0	0

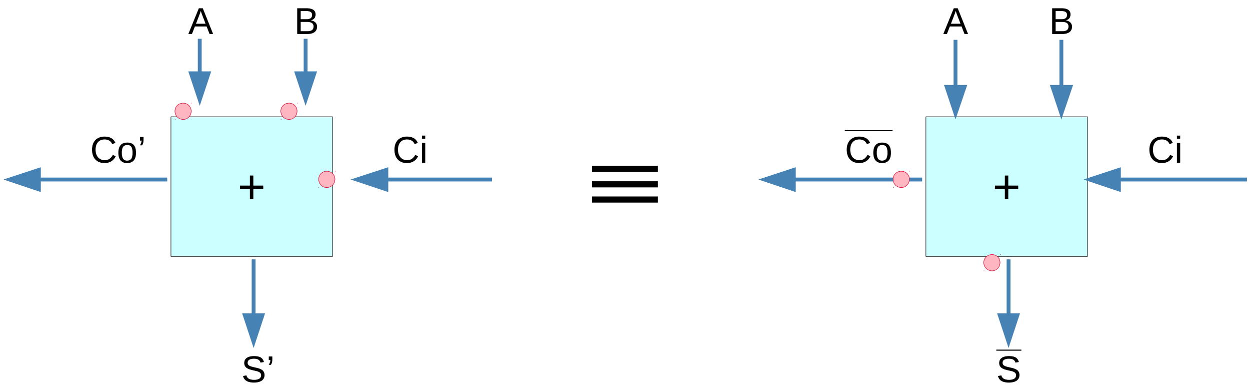




$$S(\overline{A},\overline{B},\overline{C_i}) = \overline{S(A,B,C_i)}$$

$$C_o(\overline{A},\overline{B},\overline{C_i}) = \overline{C_o(A,B,C_i)}$$

Inversion Property



Inverting all inputs to a FA
Results in inverted values for all outputs

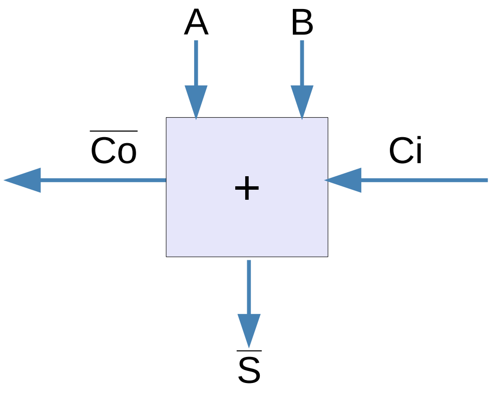
$$S(\overline{A},\overline{B},\overline{C_i}) = \overline{S(A,B,C_i)}$$

$$C_o(\overline{A},\overline{B},\overline{C_i}) = \overline{C_o(A,B,C_i)}$$

http://www.ece.ucdavis.edu/acsel, Oklobdzija

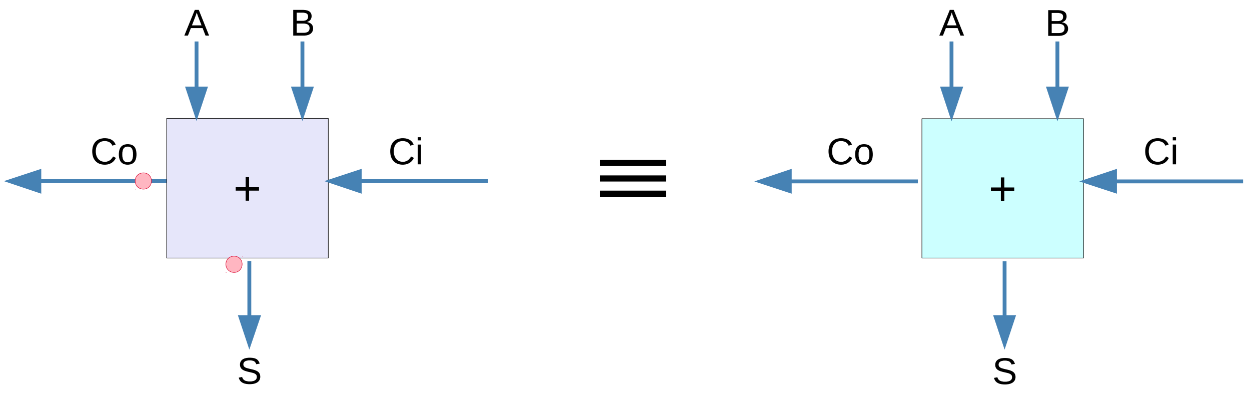
originally from Rabaey

Inverted FA Outputs



Most CMOS transistor level FAs (full adders) have inverted outputs $\overline{\text{Co}}$ and $\overline{\text{S}}$ by default

Need inverter to get normal output

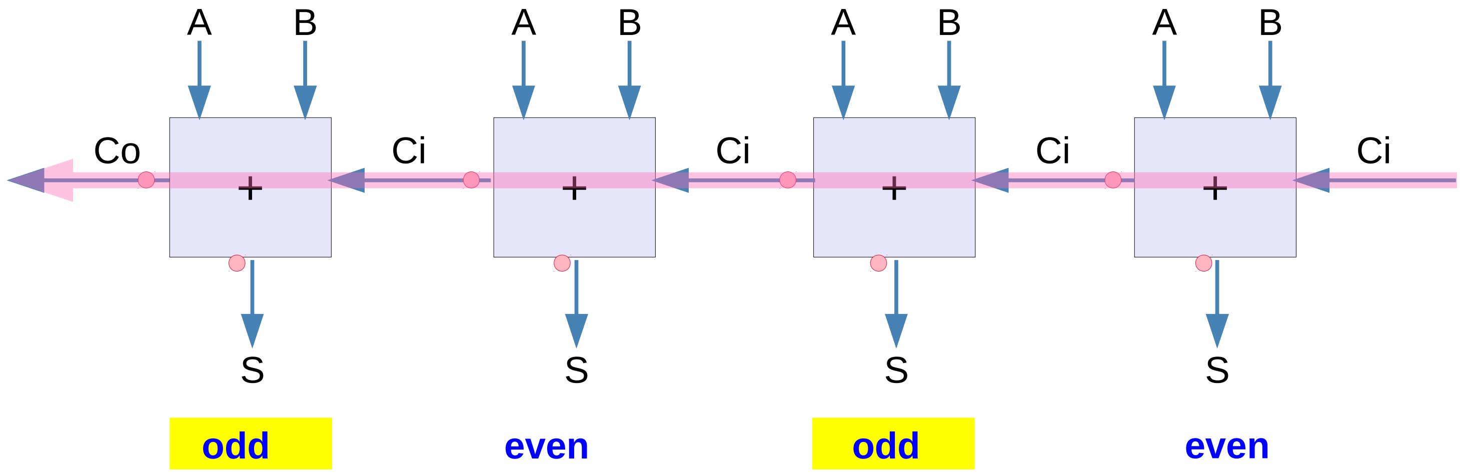


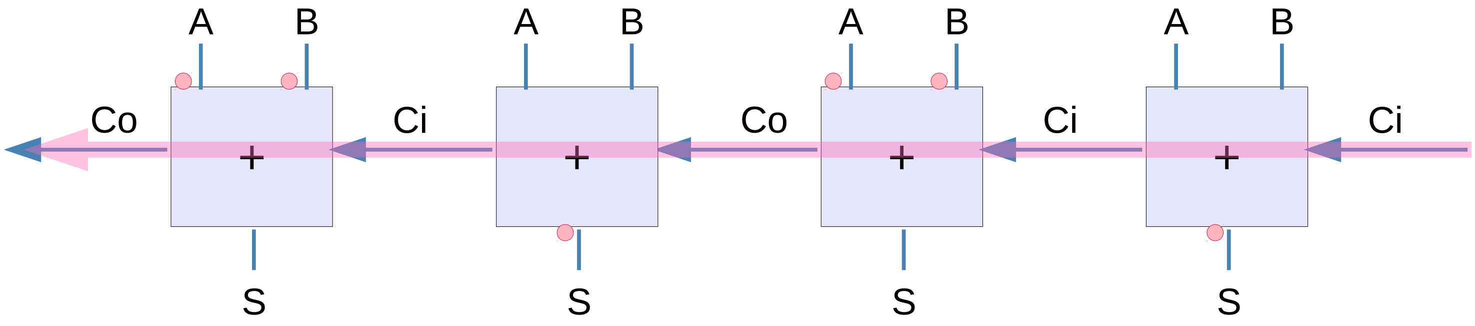
http://www.ece.ucdavis.edu/acsel, Oklobdzija

originally from Rabaey

Inverters on the critical path

4 inverters on the critical path



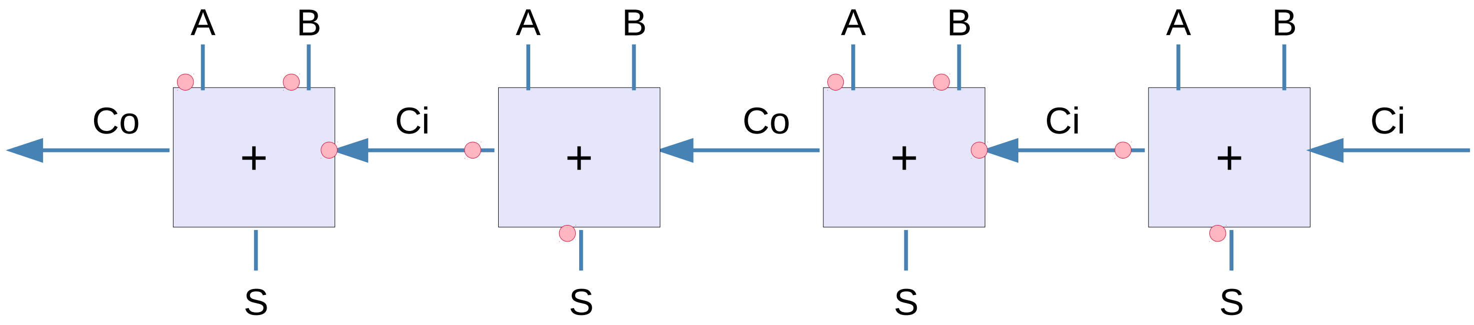


0 inverters on the critical path

http://www.ece.ucdavis.edu/acsel, Oklobdzija

originally from Rabaey

Minimize the critical paths



Minimizes the critical paths (the carry chain)
by eliminating inverters between the FAs
(will need to increase the transistor sizing)

http://www.ece.ucdavis.edu/acsel, Oklobdzija originally from Rabaey

Carry Logic and the FPAG Cell Structure (1)

When carry chains are designed for an FPGA,
inverters can be added within the design
in various places in order to optimize the design.

While adding inverters to a typical logical circuit might
cause problems with the logical correctness of the design,
inverters can be added to the FPGA
without out causing this problem.

an FPGA can support the addition of inverters
because of LUTs.

an n-input LUT can produce any function of n variables.

if inverters are added to the structure of the FPGA,
one can just reprogram the LUT to produce
an inverted function of the input variables instead.

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Carry Logic and the FPAG Cell Structure (2)

Unfortunately, the addition of **extra inverters** in a FPGA cell
could cause logical problems for the carry chains within that cell.

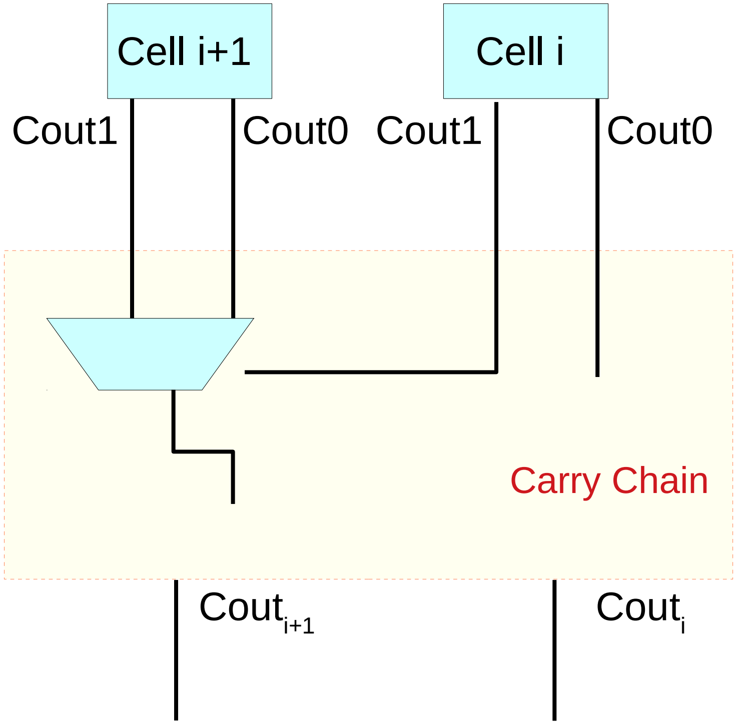
Figure shows a simple ripple carry chain
with the inverted inputs

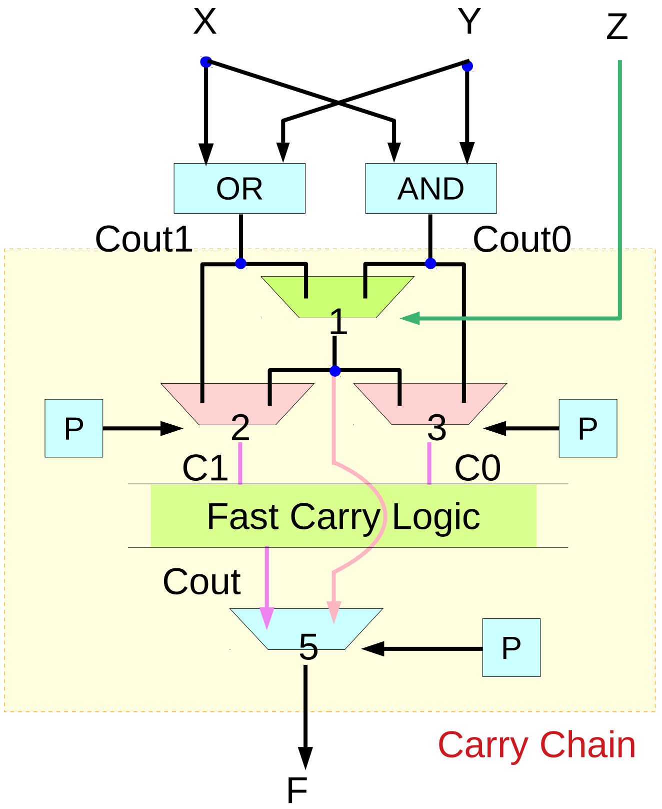
Unfortunately, the carry chain will not just
produce an inverted output.

Instead, the inversion of the Cout0 signal of the left LUT
will cause the select line of Mux 1 to be inverted.
The inversion of Mux 1's select line will cause Mux 1
to choose the wrong input, and therefore the output of Mux 1
will be incorrect.
Thus, the inverters in this example cause
the carry chain to function incorrectly,
instead of just inverting the outputs of the carry chain.

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Carry Logic and the FPAG Cell Structure (2)

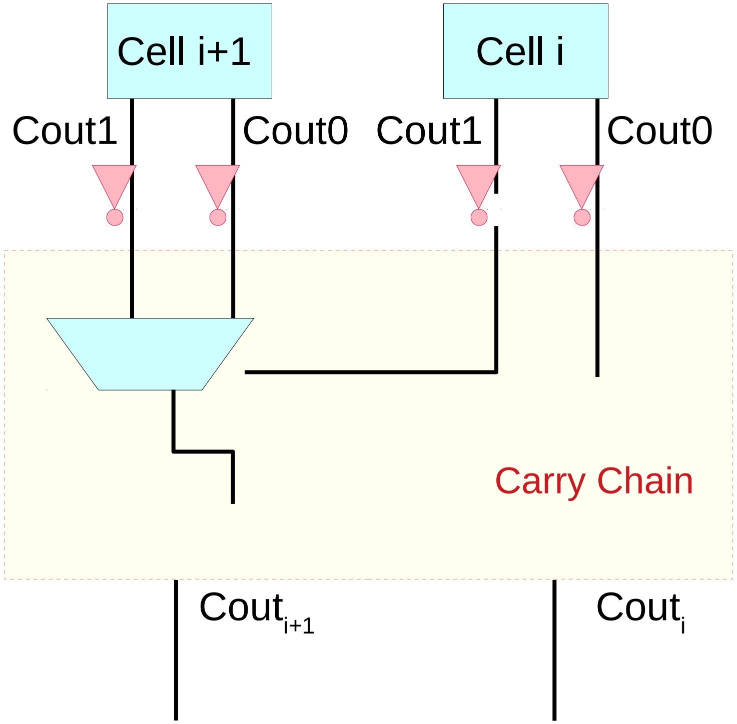




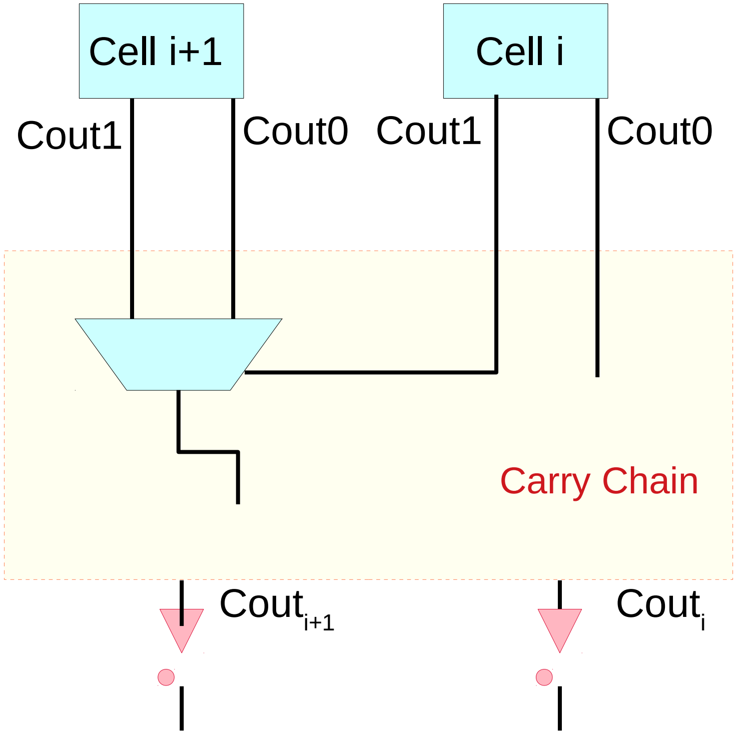
$$Cout_i = (Cout_{i-1} \cdot C1_i) + (\overline{Cout_{i-1}} \cdot C0_i)$$

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Carry Logic and the FPAG Cell Structure (2)



Case 1:
inverters before a simple carry chain



Case 2:
inverters after a simple carry chain

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

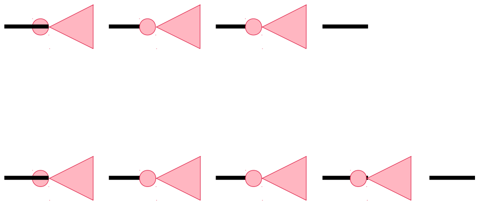
Carry Logic and the FPAG Cell Structure (3)

However, it is possible to fix this problem
so that inverters can be added to the FPGA
and so that the carry chain will still function properly.

Assumption
there are chains of inverters
that are placed within an FPGA cell
either before of after the carry chain.

Because two inverters in series
produce a logical result equivalent to 0 inverters,
any chain of inverters can be reduced
to the logical equivalent of 0 inverters or 1 inverter.

Even number of inverters : 0 inverter
Odd number of inverters : 1 inverter



1 inverter

0 inverter

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Carry Logic and the FPAG Cell Structure (3)

If there are the equivalent of 0 inverters in the FPGA cell,
then there is no problem.

Thus, there are only 2 cases to consider.

Case 1 is that there is the equivalent of 1 inverter
before the carry chain.

Case 2 is that there is the equivalent of 1 inverter
after the carry chain.

Note that the solutions to these two cases
can also be combined,
allowing inverters to appear
both before and after the carry chain.

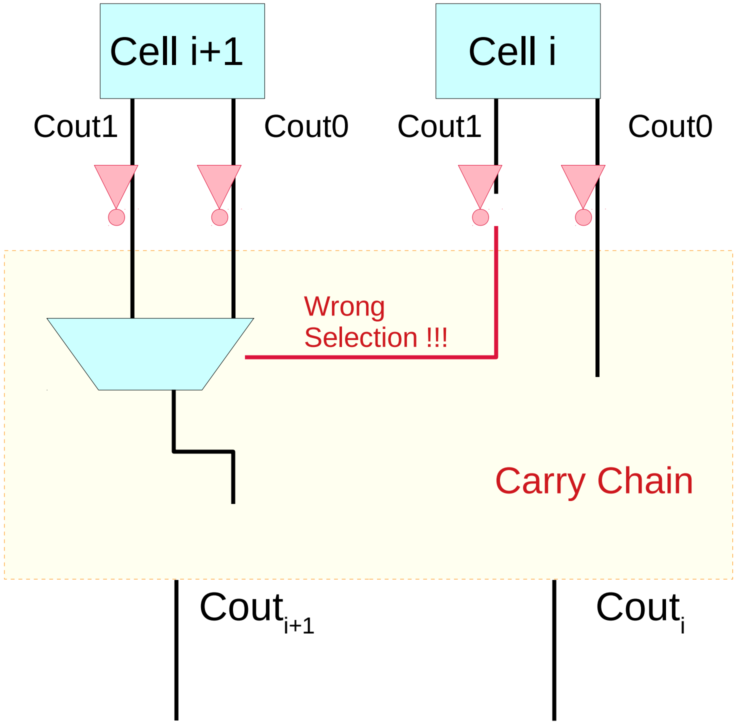
High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case I – inverter before the carry chain (1)

First, **Case 1** will be considered.

As was discussed above,
an inverted signal entering the carry chain
will cause the select lines of a mux
to choose the wrong input.

Therefore, inverted inputs can not be allowed
to enter the carry chain.



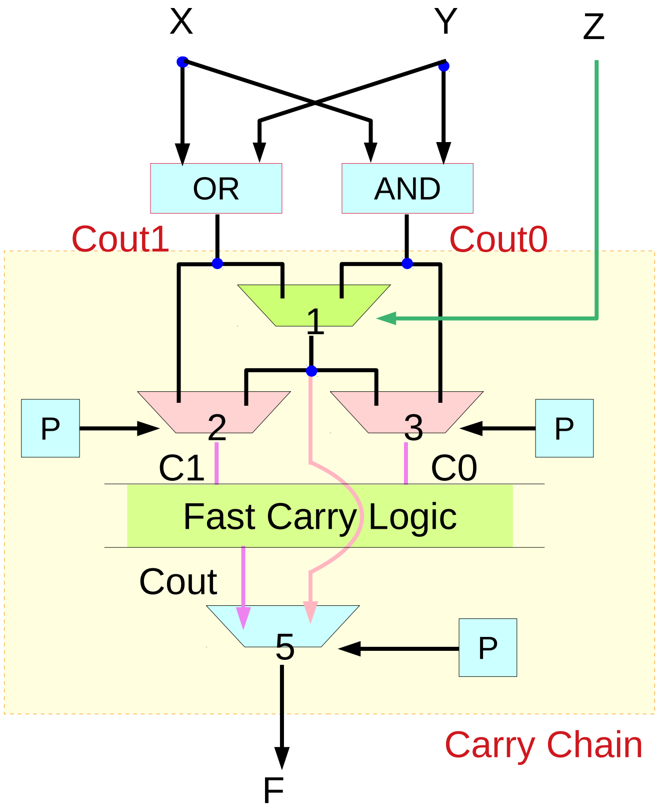
Case 1:
inverters before a simple carry chain

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case I – inverter before the carry chain (2)

As you will recall, the two 2-LUTs in Figure produce signals labeled Cout1 and Cout0.

these outputs are generated
by the 2-LUTs based
on a user-programmable function of X and Y.

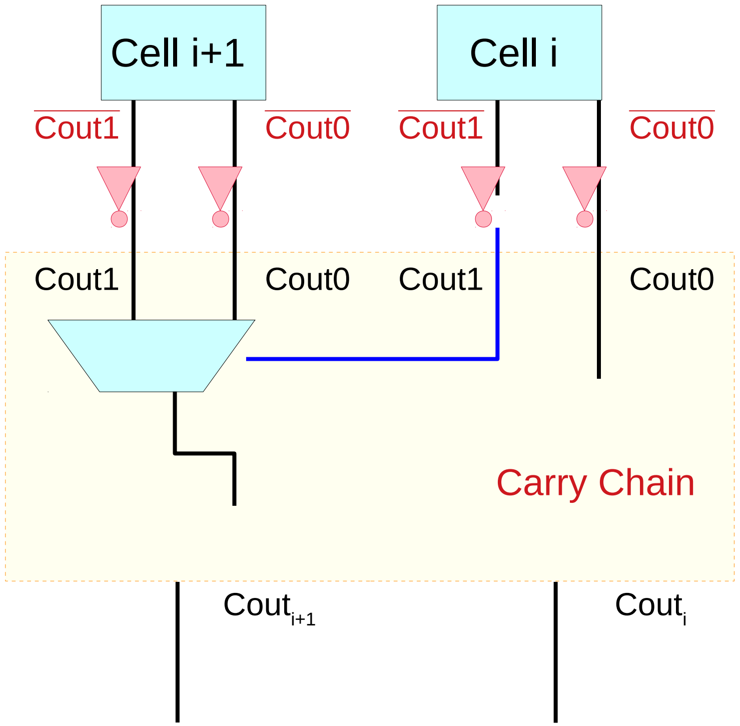


High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case I – inverter before the carry chain (3)

Therefore, the LUTs can just be reprogrammed by the user to produce $\overline{\text{Cout1}}$ and $\overline{\text{Cout0}}$ instead of Cout1 and Cout0, respectively.

Then when the logical inversion takes place before the carry chain, the inputs to the carry chain will still be equivalent to Cout1 and Cout0.



Cell i+1
Cell i
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
Cout1
Cout0
Cout1
Cout0
Carry Chain
Cout_{i+1}
Cout_{i}

Case 1:
inverters before a simple carry chain

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

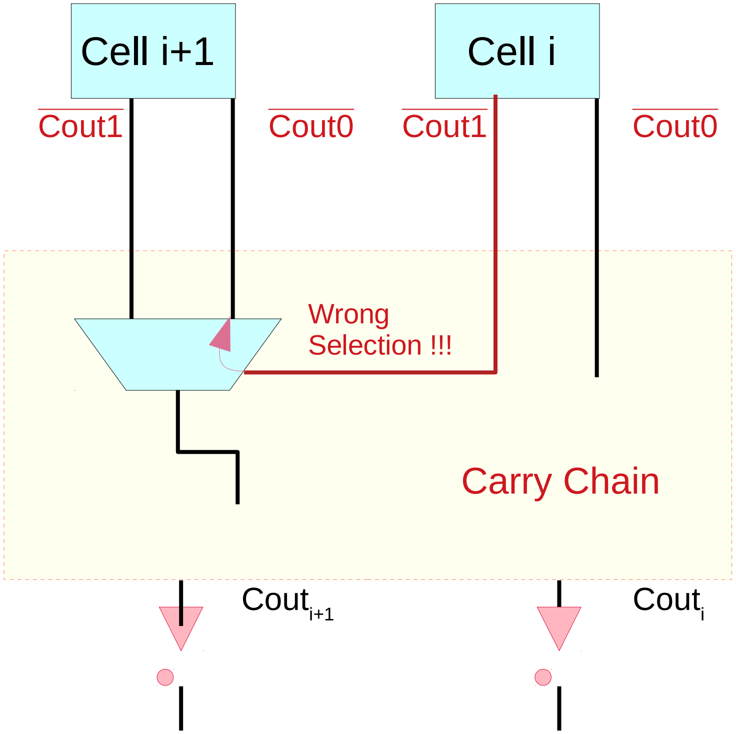
Case II – inverter after the carry chain (1)

Now **Case 2** will be considered.

In this case, 1 inverter is added to the output of the carry chain.

One initial solution might be
to just reprogram the LUTs
to output $\overline{\text{Cout1}}$ and $\overline{\text{Cout0}}$
so that the inversions cancel out.

Unfortunately, this solution does not work,
because if the inputs to the carry chain are inverted
(as the result of changing the LUT outputs),
then the select inputs of the muxes
would again be inverted,
causing the muxes to choose the wrong inputs
and causing logical incorrectness.



Cell i+1
Cell i
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
Wrong Selection !!!
Carry Chain
Cout_{i+1}
Cout_i

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

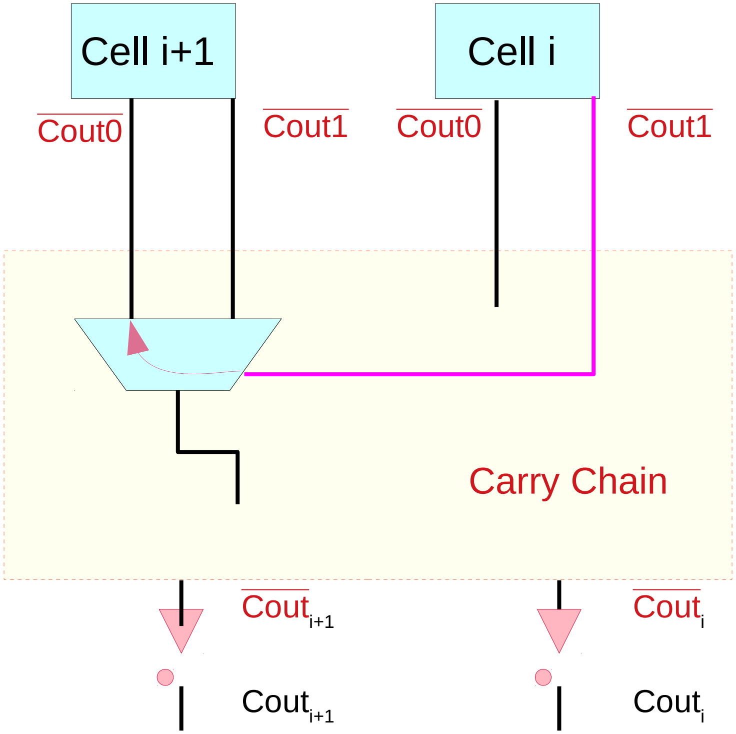
Case II – inverter after the carry chain (2)

The solution to this problem however is
to just reprogram the LUTs in a different manner.

Instead of having the LUTs
output Cout1 and Cout0,
they are instead programmed
to output $\overline{\text{Cout0}}$ and $\overline{\text{Cout1}}$, respectively.

Note that the outputs of the LUTs are
both inverted and exchanged.

The LUT that was previously outputting Cout1 is
now generating the inversion of Cout0,
and vice versa.


Cell i+1
Cell i
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
Carry Chain
$\overline{\text{Cout}}_{i+1}$
$\overline{\text{Cout}}_{i}$
Cout_{i+1}
Cout_{i}

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case II – inverter after the carry chain (3)

Now, the carry chain works properly again.

Inverting the inputs to the carry chain
causes the select lines of the muxes
to choose the wrong inputs.

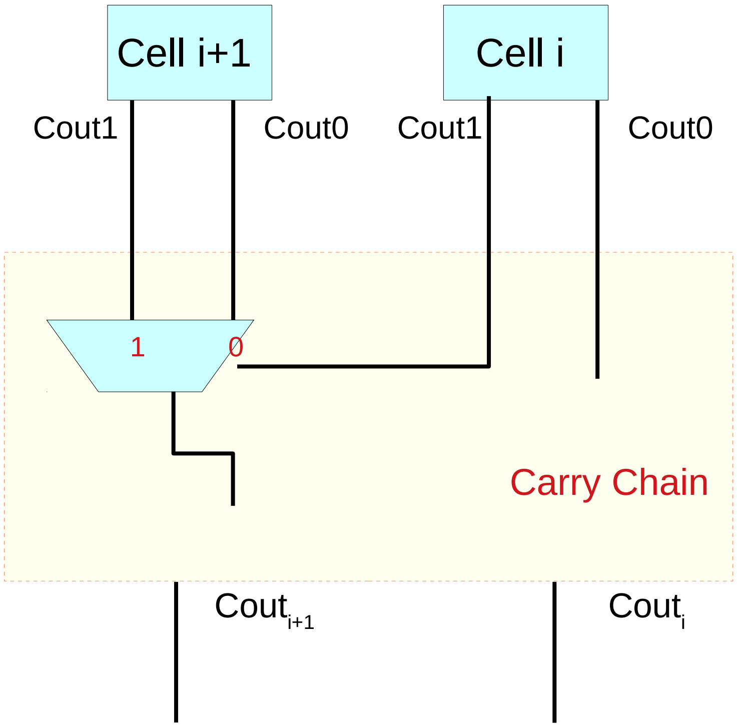
However, by switching the inputs also,
the muxes end up choosing the correct input
after all.

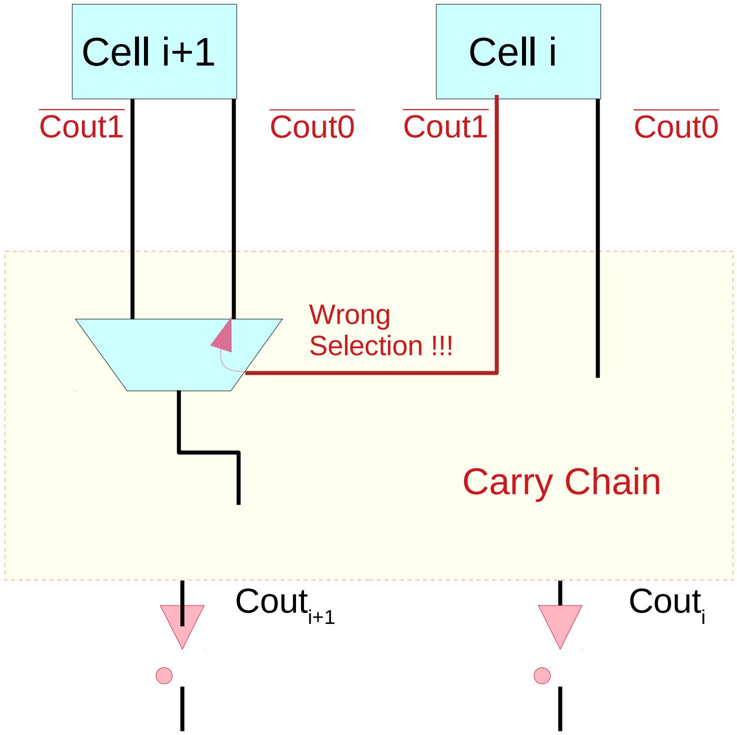
Therefore, all of the outputs of the carry chain
are now inverted.

However, since there is one logical inverter
after the carry chain, the final solution
is equivalent to the original solution.

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

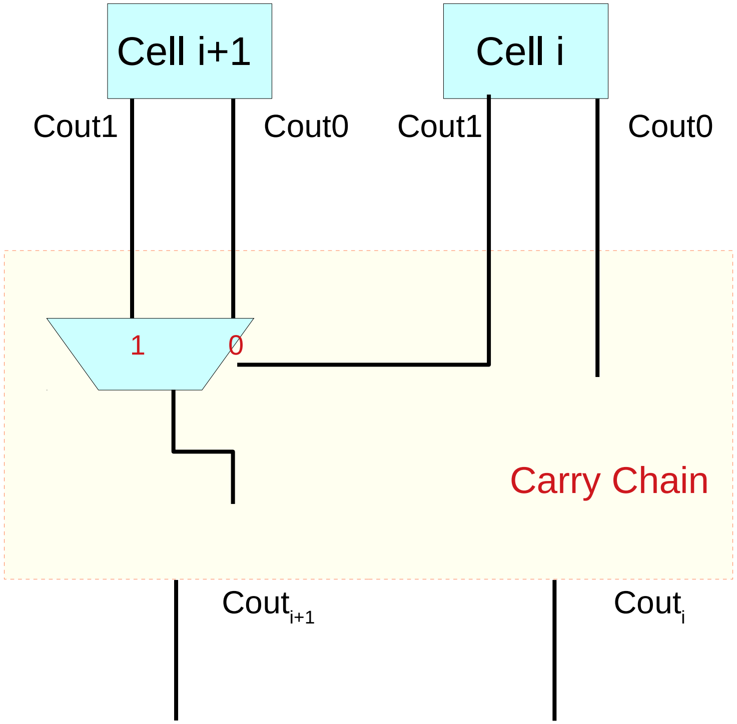
Case II – inverter after the carry chain (4)

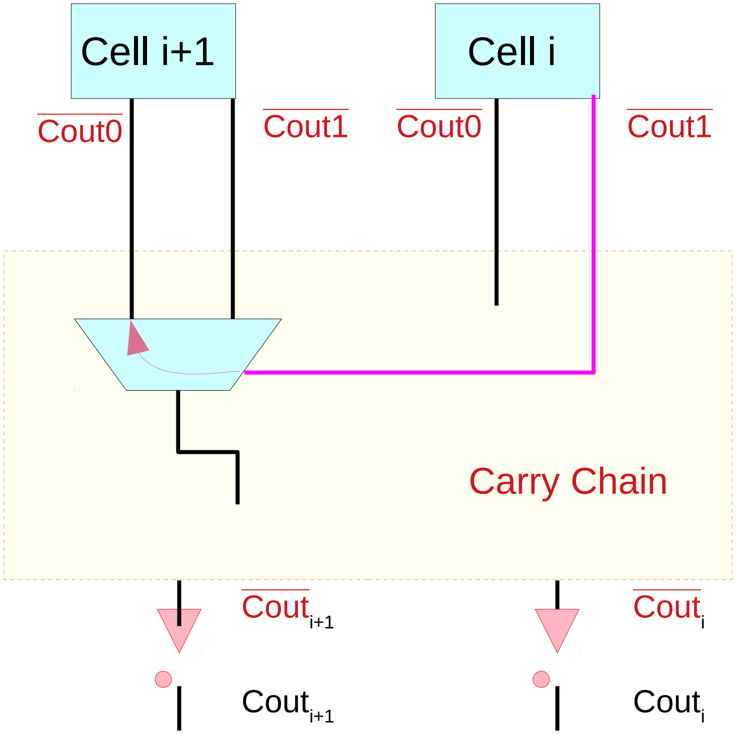



Cell i+1
Cell i
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
Wrong
Selection !!!
Carry Chain
Cout$_{i+1}$
Cout$_{i}$

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case II – inverter after the carry chain (5)





Cell i+1
Cell i
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
Carry Chain
$\overline{\text{Cout}}_{i+1}$
$\overline{\text{Cout}}_{i}$
Cout_{i+1}
Cout_{i}

If Cout1 of Cell i is 1,
then Cout1 of Cell i+1 is selected
If Cout1 of Cell i is 0,
then Cout0 of Cell i+1 is selected

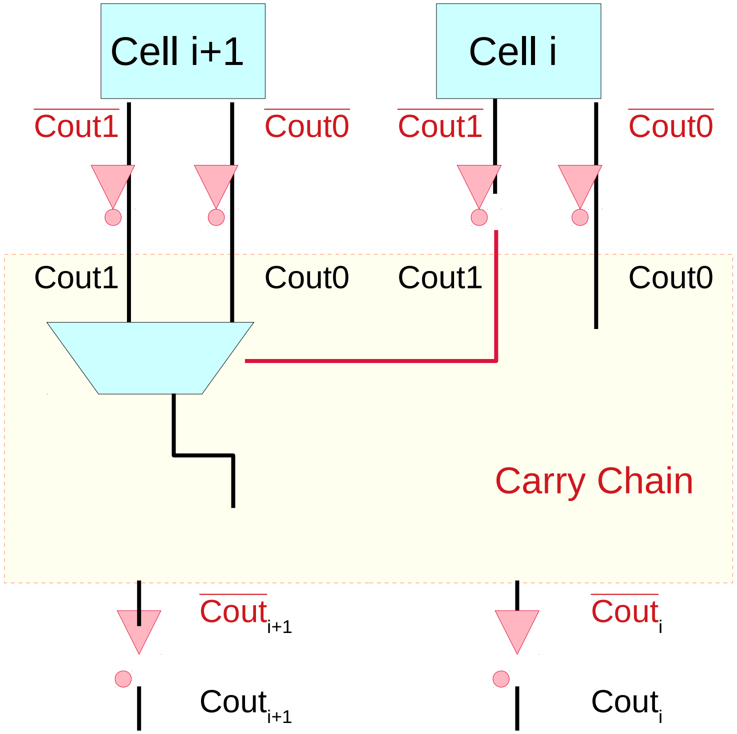
If Cout1 of $\overline{\text{Cell i}}$ is 0,
then $\overline{\text{Cout0}}$ of Cell i+1 is selected
If Cout0 of $\overline{\text{Cell i}}$ is 1,
then $\overline{\text{Cout1}}$ of Cell i+1 is selected

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case I+II (1)

The rules in **Case 1** and **Case 2** can then be applied together to handle any structure of inverters.

For example, if there are inverters both before and after the carry chains, then first **Case 1** is applied to the cells to negate the inverters before the carry chain. Thus, Cout1 and Cout0 are inverted.

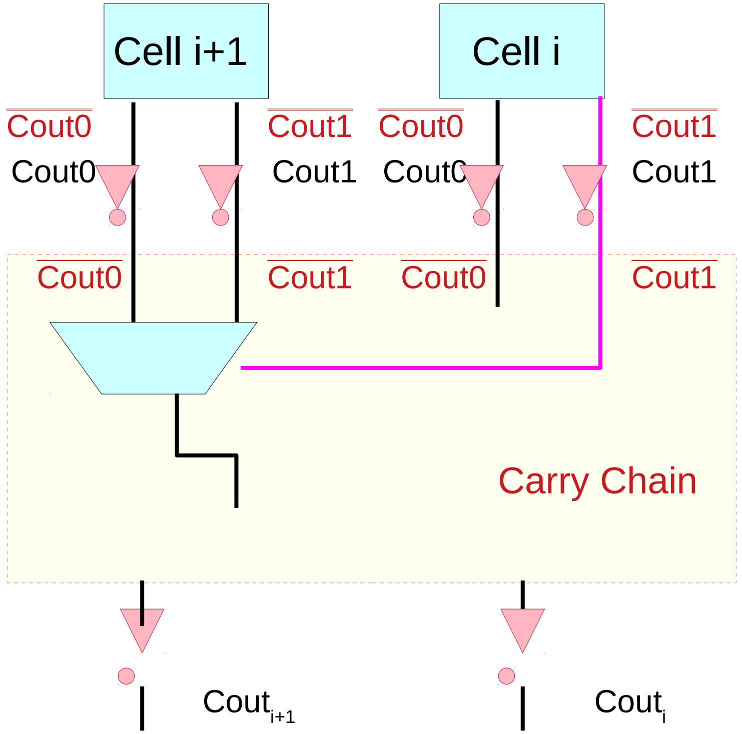

Cell i+1
Cell i
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
Cout1
Cout0
Cout1
Cout0
Carry Chain
$\overline{\text{Cout}}_{i+1}$
$\overline{\text{Cout}}_{i}$
Cout_{i+1}
Cout_{i}

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Case I+II (2)

Then **Case 2** is applied to the cells
so that the outputs of the LUT,
$\overline{\text{Cout1}}$ and $\overline{\text{Cout0}}$ (as produced by **Case 1**),
are inverted and switched.
Thus, the final output of the LUTs
for the case of inverters
before and after the carry chain
is Cout0 and Cout1, respectively.

Therefore, any number of inversions
may be placed before or after the carry chain
without affecting its logical correctness.


Cell i+1
Cell i
$\overline{\text{Cout0}}$
Cout0
$\overline{\text{Cout1}}$
Cout1
$\overline{\text{Cout0}}$
Cout0
$\overline{\text{Cout1}}$
Cout1
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
$\overline{\text{Cout0}}$
$\overline{\text{Cout1}}$
Carry Chain
Cout_{i+1}
Cout_i

High Performance Carry Chains for FPGAs, M. M. Hosler, https:://people.ece.uw.edu

Variable Block

carry select chain,

blocks of ripple carry element
precomputing the Cout value
for each possible Cin value
(**true** Cin or **false** Cin)

a **variable block structure**

blocks of ripple carry element
skip the **carry signal** over intermediate cells
where appropriate.

contiguous blocks are grouped together
to form a unit with a standard ripple carry chain

skip logic allows the value of the block's Cin,
to be bypassed to later blocks.

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Ripple Carry Structure

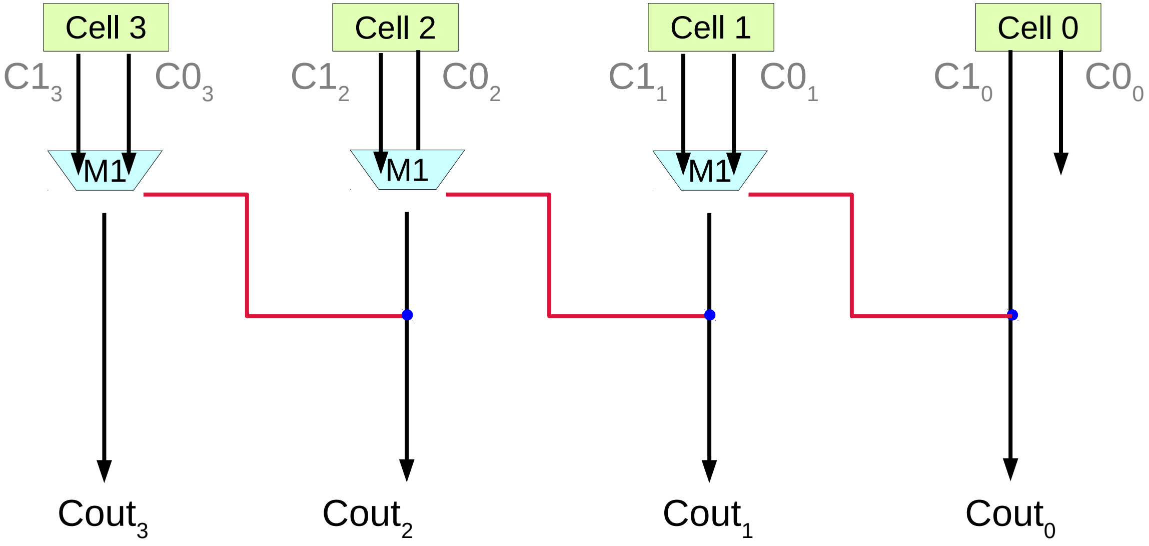
A **Carry Select carry chain** structure for use in FPGAs

the carry computation for the first two cells is performed with the simple **ripple-carry** structure implemented by mux1

$$Cout = (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

$$C1 = X + Y$$
$$C0 = X \cdot Y$$

X	Y	Cout
0	0	0
0	1	Cin
1	0	Cin
1	1	1



High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

X	Y	C1	C0	
0	0	0	0	$\overline{X}\,\overline{Y}$
0	1	1	0	$\overline{X}\,Y$
1	0	1	0	$X\,\overline{Y}$
1	1	1	1	$X\,Y$

$$C1 = X + Y$$
$$C0 = X \cdot Y$$

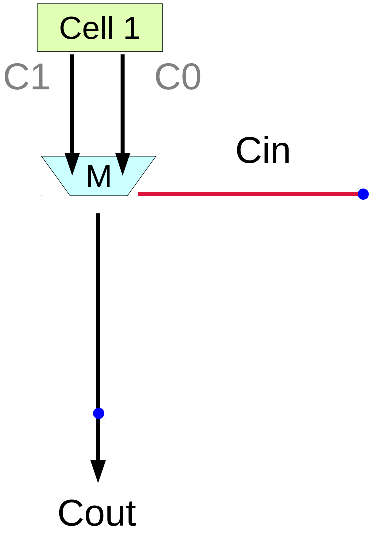
C1	C0	Name	
0	0	0	Kill
0	1	$\overline{\text{Cin}}$	Inverse Propagate
1	0	Cin	Propagate
1	1	1	Generate

$$Cout = (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

$$(Cin \cdot C1) = Cin \cdot (\overline{X}\,Y + X\,\overline{Y} + X\,Y) \rightarrow \ propagate\ Cin$$

$$(\overline{Cin} \cdot C0) = \overline{Cin} \cdot X\,Y \rightarrow \ generate\ a\ new\ carry$$

X	Y	Cin	Cout
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1



$$= (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Ripple Carry Structure

$$\overline{Cout} = \overline{(Cin \cdot C1) + (\overline{Cin} \cdot C0)}$$

$$= \overline{(Cin \cdot C1)} \cdot \overline{(\overline{Cin} \cdot C0)}$$

$$= (\overline{Cin} + \overline{C1}) \cdot (Cin + \overline{C0})$$

$$= \overline{Cin}\,Cin + \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin + \overline{C1}\,\overline{C0}$$

$$= \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin + \overline{C1}\,\overline{C0}$$

$$= \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin$$

redundant

$\overline{((Cin \cdot C1) + (\overline{Cin} \cdot C0))}$ *means*

$((Cin \cdot C1) + (\overline{Cin} \cdot C0))$ *is false*

$[(Cin \cdot C1) = F] \wedge [(\overline{Cin} \cdot C0) = F]$

Two mutually exclusive cases

when Cin *is true*

$C1$ *must be false*

because $(Cin \cdot C1)$ *must be false*

➡ *therefore* $(\overline{C1}\,Cin)$

when $\overline{Cin}$ *is true*

$C0$ *must be false*

because $(\overline{Cin} \cdot C0)$ *must be false*

➡ *therefore* $(\overline{C0}\,\overline{Cin})$

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Ripple Carry Structure

$$Cout = (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

$$C1 = X + Y$$
$$C0 = X \cdot Y$$

X	Y	Cout
0	0	0
0	1	Cin
1	0	Cin
1	1	Cin + $\overline{\text{Cin}}$

$$\overline{Cout} = \overline{(Cin \cdot C1) + (\overline{Cin} \cdot C0)}$$
$$= \overline{(Cin \cdot C1)} \cdot \overline{(\overline{Cin} \cdot C0)}$$
$$= (\overline{Cin} + \overline{C1}) \cdot (Cin + \overline{C0})$$
$$= \overline{Cin}\,Cin + \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin + \overline{C1}\,\overline{C0}$$
$$= \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin + \overline{C1}\,\overline{C0}$$
$$= \overline{Cin}\,\overline{C0} + \overline{C1}\,Cin$$

redundant

	X	Y	$\overline{\text{C1}}$	$\overline{\text{C0}}$	Cin·$\overline{\text{C1}}$	$\overline{\text{Cin}}$·$\overline{\text{C0}}$	Cin·$\overline{\text{C1}}$ + $\overline{\text{Cin}}$·$\overline{\text{C0}}$	$\overline{\text{C1}}$·$\overline{\text{C0}}$
	0	0	1	1	Cin	$\overline{\text{Cin}}$	1	1
Inverse Propagate	0	1	0	1	0	$\overline{\text{Cin}}$	$\overline{\text{Cin}}$	0
	1	0	0	1	0	$\overline{\text{Cin}}$	$\overline{\text{Cin}}$	0
	1	1	0	0	0	0	0	0

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Ripple Carry Structure

$$Cout = Cin \cdot C1 + \overline{Cin} \cdot C0$$

- the wire **Cout** has the value of Cout
- the wire **C0** has the value of C0
- the wire **C0** has the value of C0

	X	Y	C1	C0	Cin·C1	$\overline{Cin}$·C0		C1·C0
	0	0	0	0	0	0	0	0
Propagate	0	1	1	0	Cin	0	Cin	0
	1	0	1	0	Cin	0	Cin	0
	1	1	1	1	Cin	$\overline{Cin}$	1	1

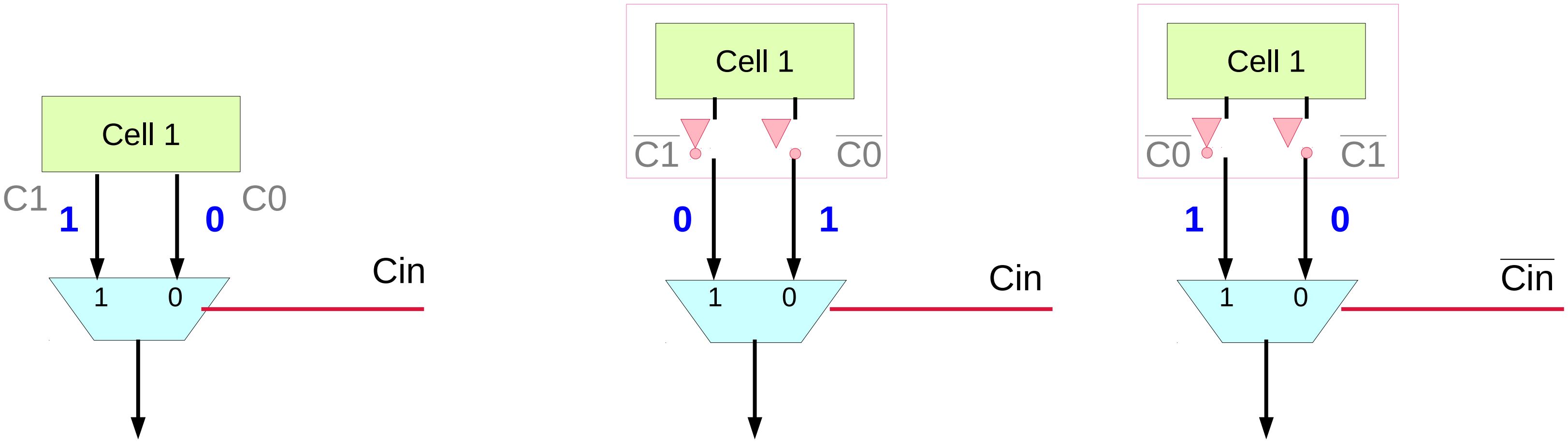
$$\overline{Cout} = \overline{Cin} \cdot \overline{C0} + \overline{C1} \cdot Cin$$

- the wire **Cout** has the value of $\overline{Cout}$
- the wire **C1** has the value of $\overline{C1}$
- the wire **C0** has the value of $\overline{C0}$

	X	Y	$\overline{C1}$	$\overline{C0}$	Cin·$\overline{C1}$	$\overline{Cin}$·$\overline{C0}$		$\overline{C1}$·$\overline{C0}$
	0	0	1	1	Cin	$\overline{Cin}$	1	1
Inverse Propagate	0	1	0	1	0	$\overline{Cin}$	$\overline{Cin}$	0
	1	0	0	1	0	$\overline{Cin}$	$\overline{Cin}$	0
	1	1	0	0	0	0	0	0

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin



During optimization process, inverters may be added before the node **C1** and **C0**

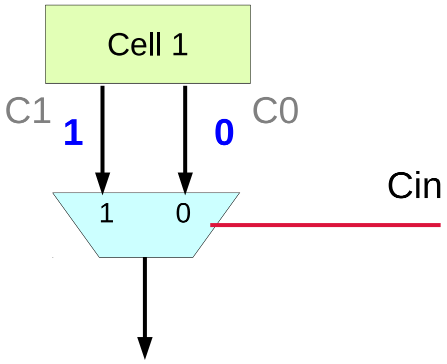
- the wire **C1** has the value of $\overline{C1}$
- the wire **C0** has the value of $\overline{C0}$

EDA synthesis tools may insert
a pair of inverters to the chosen cell
for the optimization purpose (size, time).

we cannot know in advance which cell
has such inverters before the synthesis process

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

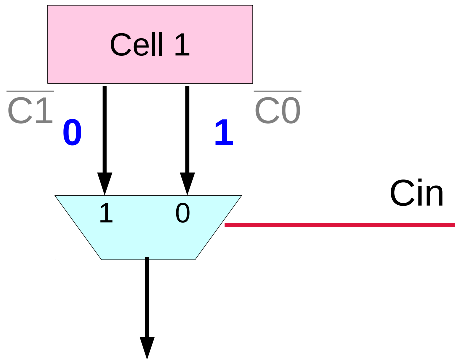


$$\text{Cout} = (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

If Cin then Cout is C1 (= X+Y)
else Cout is C0 (= XY)

[If C1 then Cout is Cin] + (OR)
[If C0 else Cout is $\overline{\text{Cin}}$]

Cin if C1
$\overline{\text{Cin}}$ if C0

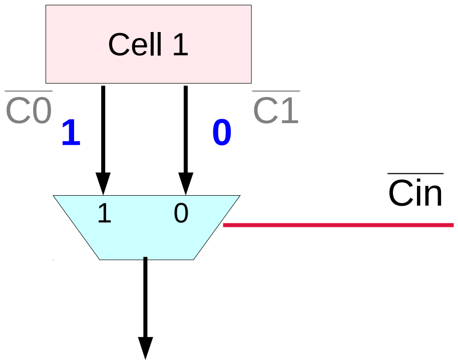


$$\overline{\text{Cout}} = (\overline{Cin} \cdot \overline{C0}) + (Cin \cdot \overline{C1})$$

If Cin then $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{\text{X+Y}}$)
else $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= $\overline{\text{XY}}$)

[If $\overline{\text{C0}}$ then $\overline{\text{Cout}}$ is $\overline{\text{Cin}}$] + (OR)
[If $\overline{\text{C1}}$ else $\overline{\text{Cout}}$ is Cin]

$\overline{\text{Cin}}$ if $\overline{\text{C0}}$
Cin if $\overline{\text{C1}}$



$$\overline{\text{Cout}} = (\overline{Cin} \cdot \overline{C0}) + (Cin \cdot \overline{C1})$$

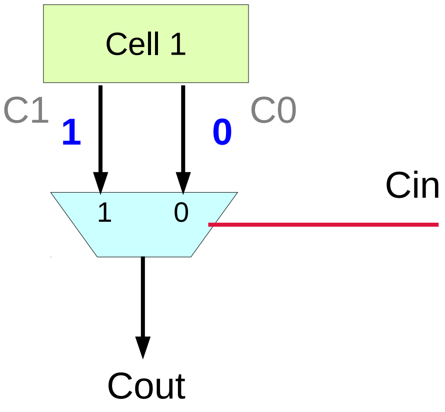
If $\overline{\text{Cin}}$ then $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= $\overline{\text{XY}}$)
else $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{\text{X+Y}}$)

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

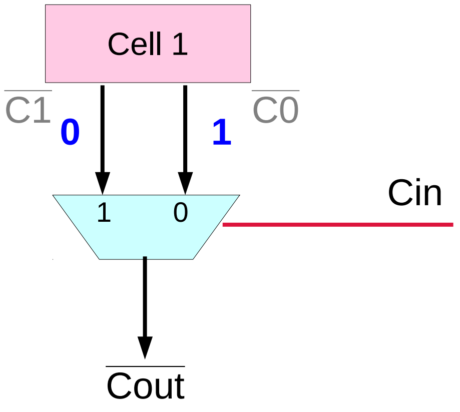
C1	C0	Name		Propagate_i
0	0	0	Kill	0
0	1	$\overline{Cin}$	Inverse Propagate	1
1	0	Cin	Propagate	1
1	1	1	Generate	0

$$C1 = X + Y$$
$$C0 = X \cdot Y$$



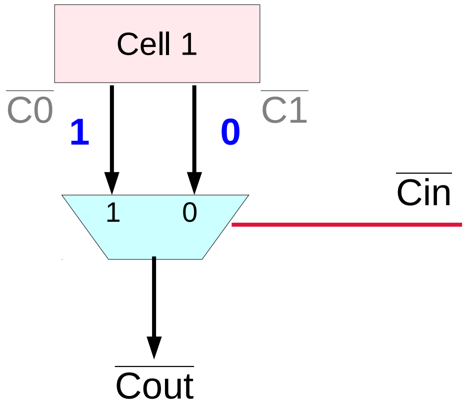
$$\text{Cout} = (Cin \cdot C1) + (\overline{Cin} \cdot C0)$$

If Cin then Cout is C1 (= X+Y)
else Cout is C0 (= XY)



$$\overline{\text{Cout}} = (\overline{Cin} \cdot \overline{C0}) + (Cin \cdot \overline{C1})$$

If Cin then $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{X+Y}$)
else $\overline{\text{Cout}}$ is $\overline{\text{Co}}$ (= $\overline{XY}$)



$$\overline{\text{Cout}} = (\overline{Cin} \cdot \overline{C0}) + (Cin \cdot \overline{C1})$$

If $\overline{\text{Cin}}$ then $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= $\overline{XY}$)
else $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{X+Y}$)

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

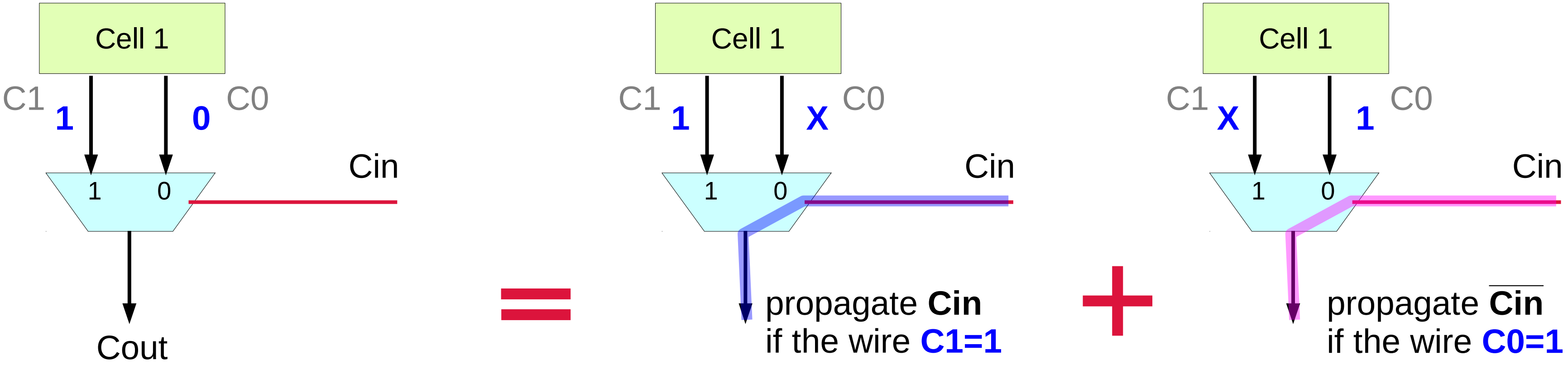
Cout using C1, C0, Cin

$$C1 = X + Y$$
$$C0 = X \cdot Y$$

Cin	X	Y	Cout	C1	C0
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

Cout $= (Cin \cdot C1) + (\overline{Cin} \cdot C0)$

If Cin then Cout is C1 (= X+Y)
else Cout is C0 (= XY)



propagate **Cin**
if the wire **C1=1**

Cin if C1

propagate **$\overline{\text{Cin}}$**
if the wire **C0=1**

$\overline{\text{Cin}}$ if C0

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

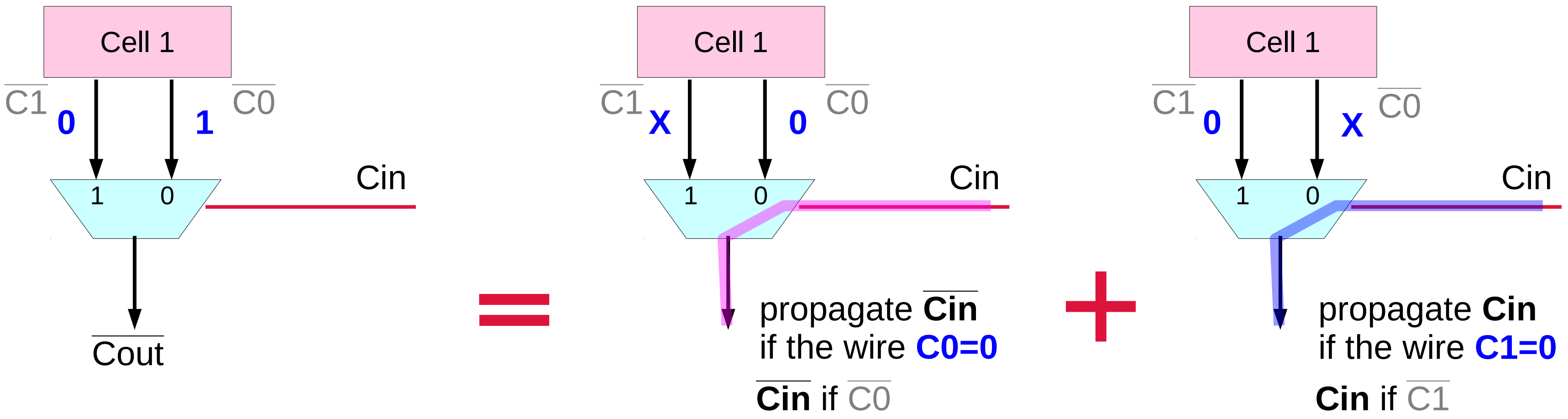
Cout using C1, C0, Cin

$$C1 = X+Y$$
$$C0 = X\cdot Y$$

$\overline{Cin}$	X	Y	$\overline{Cout}$	$\overline{C1}$	$\overline{C0}$
1	0	0	1	1	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	0	0	0
0	0	0	1	1	1
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	0	0	0

$$\overline{\text{Cout}} = \left(\overline{Cin}\cdot\overline{C0}\right) + \left(Cin\cdot\overline{C1}\right)$$

If $\overline{\text{Cin}}$ then $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= $\overline{\text{XY}}$)
else $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{\text{X+Y}}$)



propagate $\overline{\textbf{Cin}}$
if the wire **C0=0**

$\overline{\textbf{Cin}}$ if $\overline{\text{C0}}$

propagate **Cin**
if the wire **C1=0**

Cin if $\overline{\text{C1}}$

High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

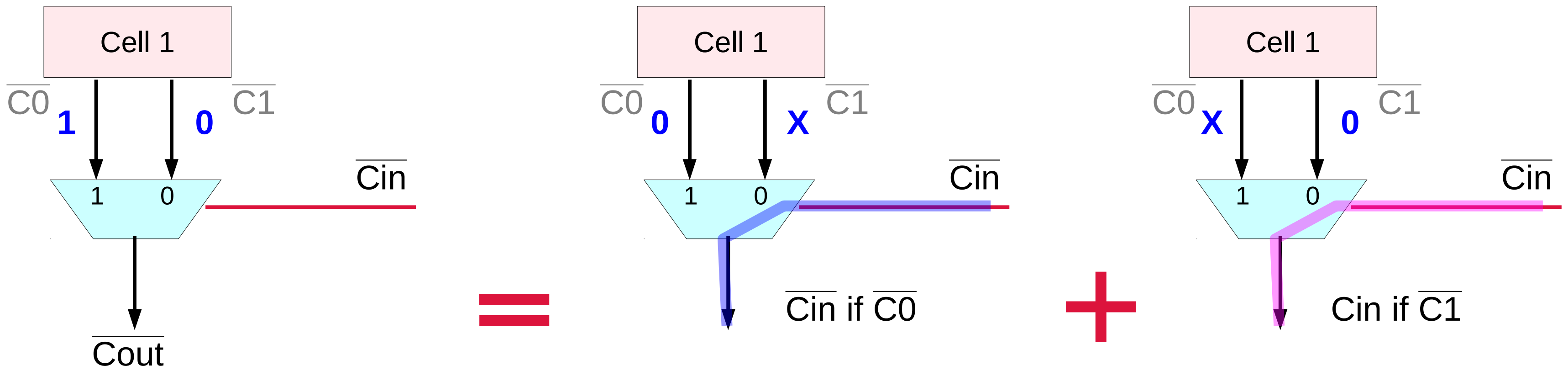
$C1 = X + Y$

$C0 = X \cdot Y$

$\overline{\text{Cin}}$	X	Y	$\overline{\text{Cout}}$	$\overline{\text{C1}}$	$\overline{\text{C0}}$	$\overline{\text{C0}}$	$\overline{\text{C1}}$
1	0	0	1	1	1	1	1
1	0	1	1	0	1	1	0
1	1	0	1	0	1	1	0
1	1	1	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	0	1	1	0
0	1	0	0	0	1	1	0
0	1	1	0	0	0	0	0

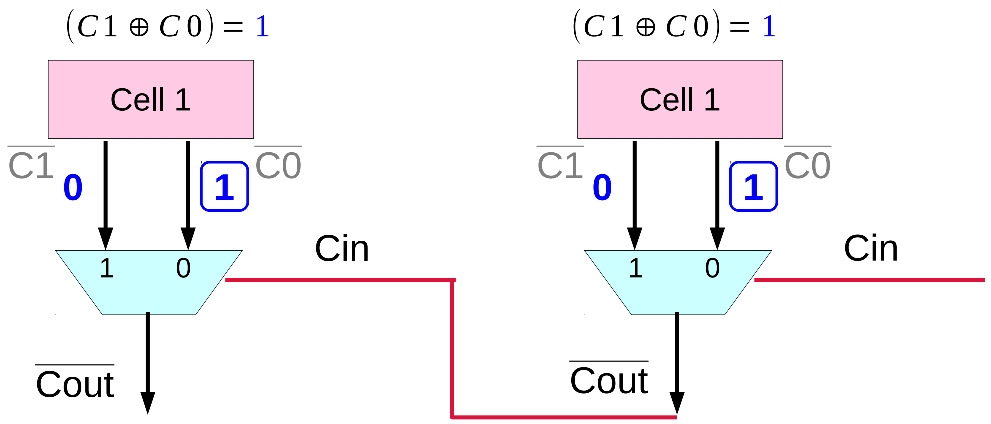
$$\begin{aligned}\overline{\text{Cout}} &= \left(\overline{Cin} \cdot \overline{C0}\right) \\ &+ \left(Cin \cdot \overline{C1}\right)\end{aligned}$$

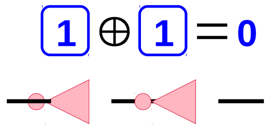
If $\overline{\text{Cin}}$ then $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= $\overline{\text{XY}}$)
else $\overline{\text{Cout}}$ is $\overline{\text{C1}}$ (= $\overline{\text{X+Y}}$)



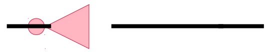
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin



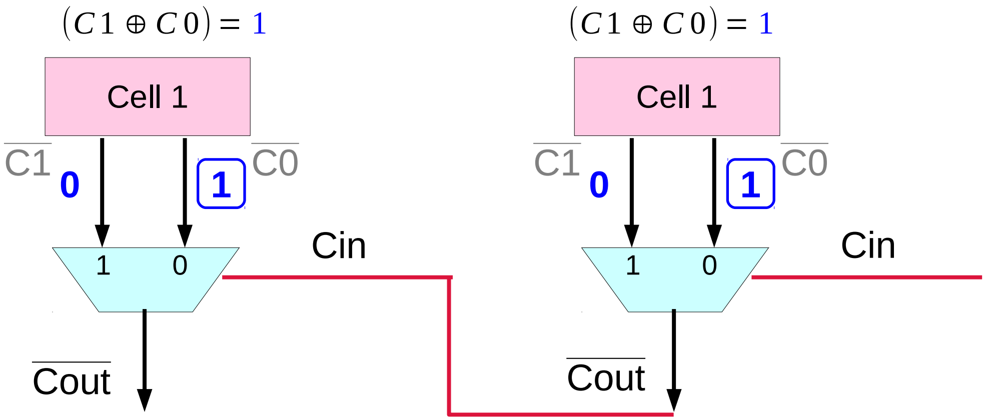


C1	C0	Cout
0	0	0
0	1	$\overline{Cin}$
1	0	Cin
1	1	1

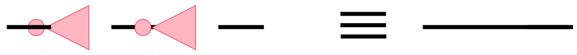


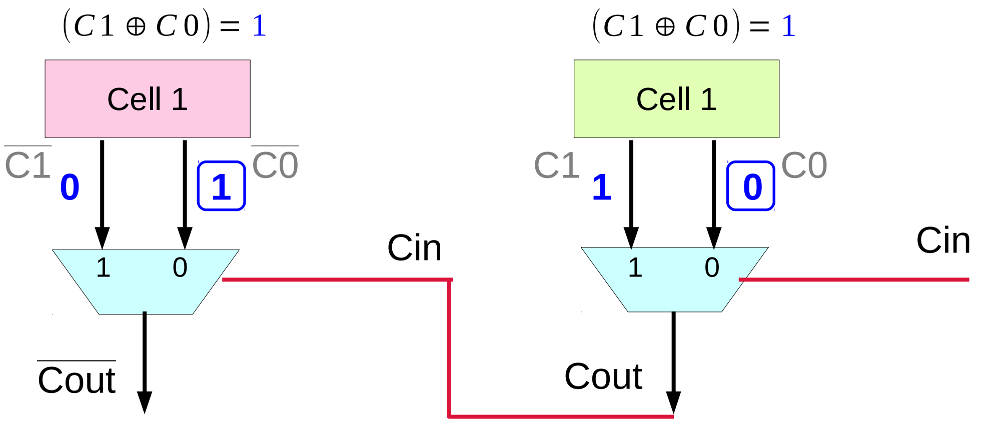
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

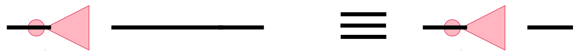


$\boxed{1} \oplus \boxed{1} = 0$



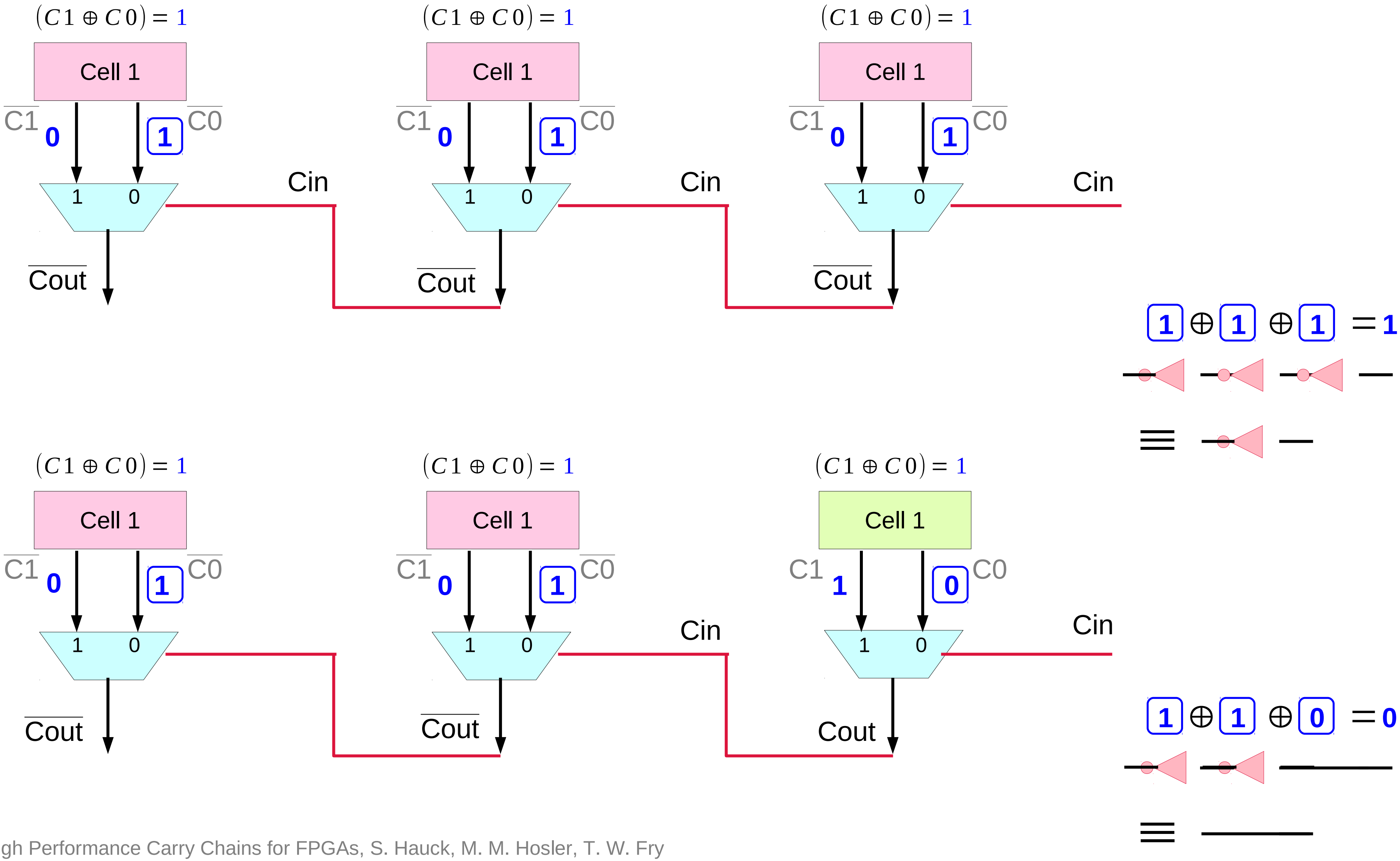


$\boxed{1} \oplus \boxed{0} = 1$



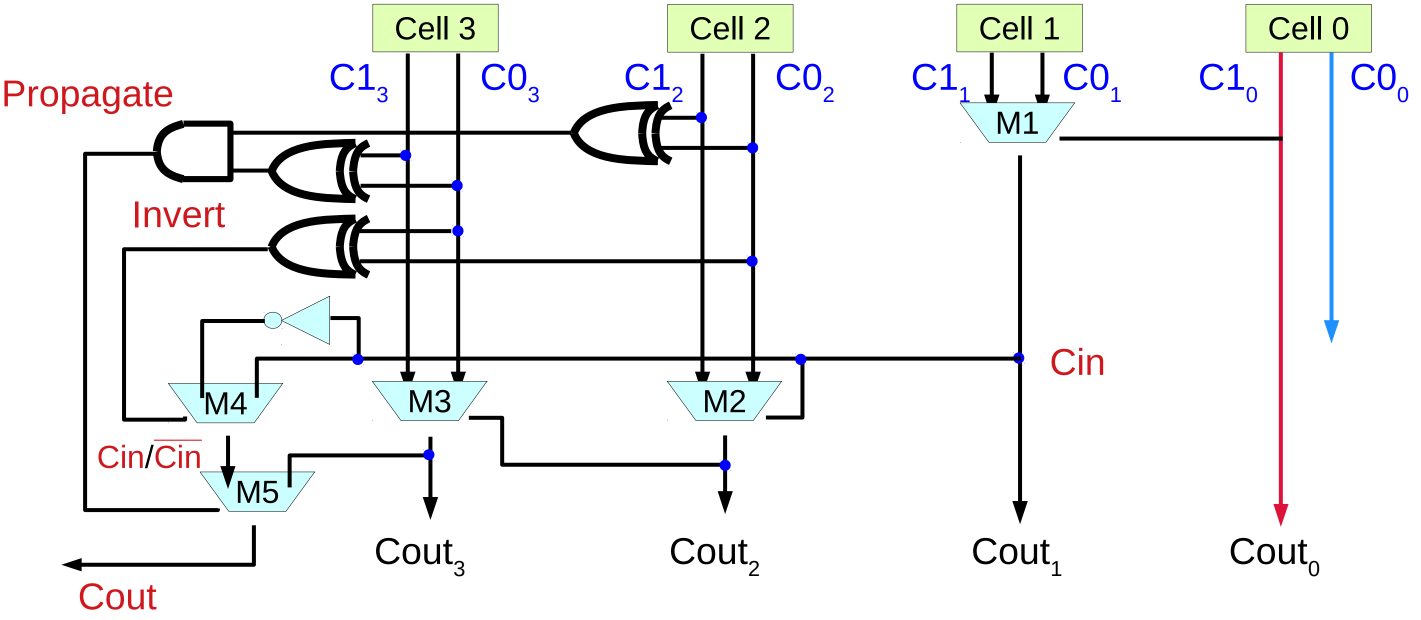
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin



High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Variable Block Carry Structure



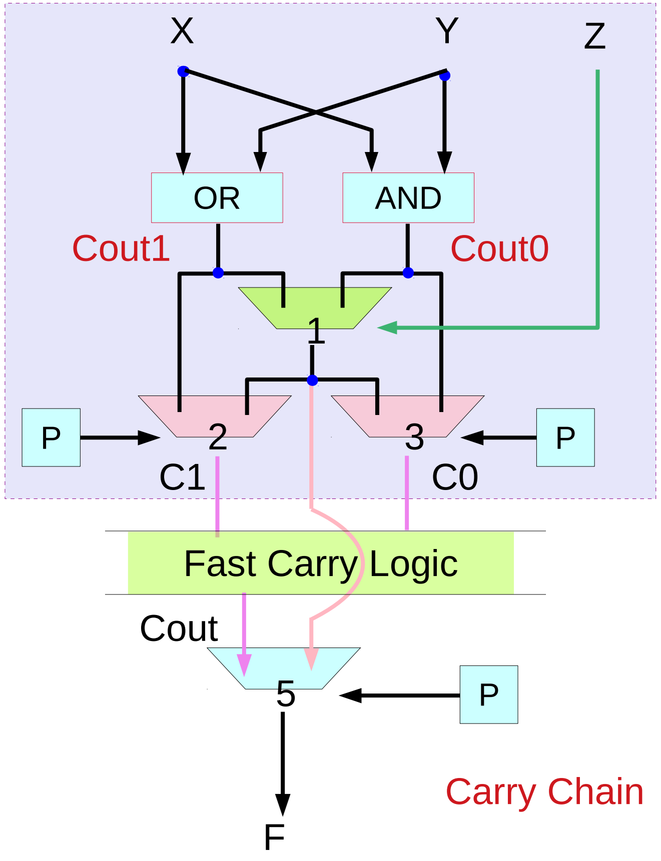
$$C1 = X + Y$$
$$C0 = X \cdot Y$$

original definition:
C1 = XY
C0 = X+Y
in the referenced paper

- M1 performs an initial two single stage ripple carry
- M2 ~ M5 form a 2-bit variable block
- M5 decides whether the Cin / $\overline{\text{Cin}}$ signal should be sent directly to Cout,
- M4 decides whether to invert the Cin signal or not

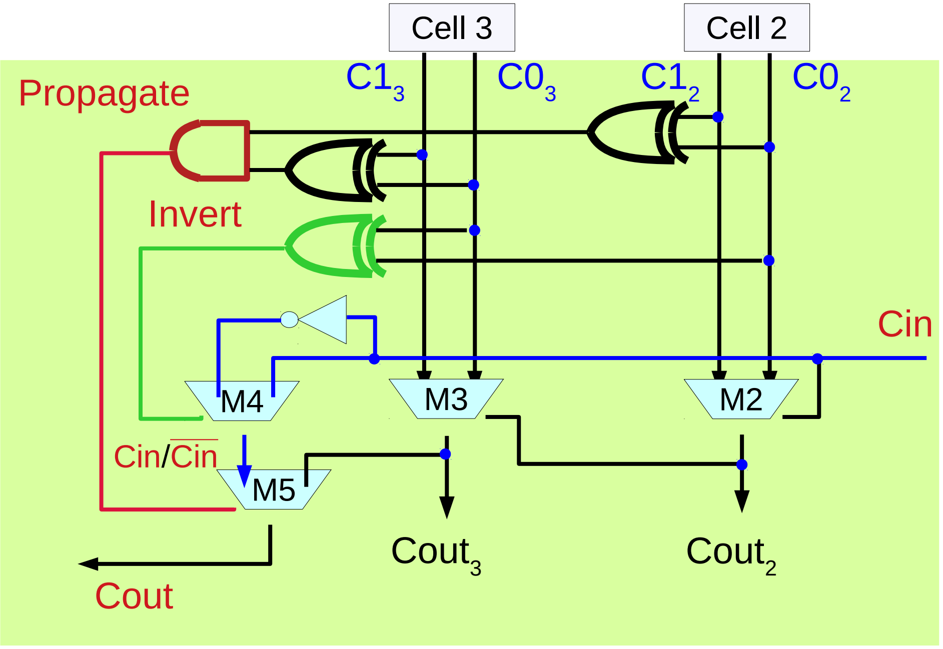
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Variable Block



https://en.wikipedia.org/wiki/Carry-lookahead_adder

C1	C0	Name	
0	0	0	Kill
0	1	$\overline{\text{Cin}}$	Inverse Propagate
1	0	Cin	Propagate
1	1	1	Generate



Fast Carry Logic

Variable Block

a major difficulty in developing a version of the Variable Block carry chain for inclusion in an FPGA's architecture is
the need to support both the propagate and inverse propagate state the cells.

To do this, we compute two values.

check if all the cells are in

- normal propagate
- inverse propagate

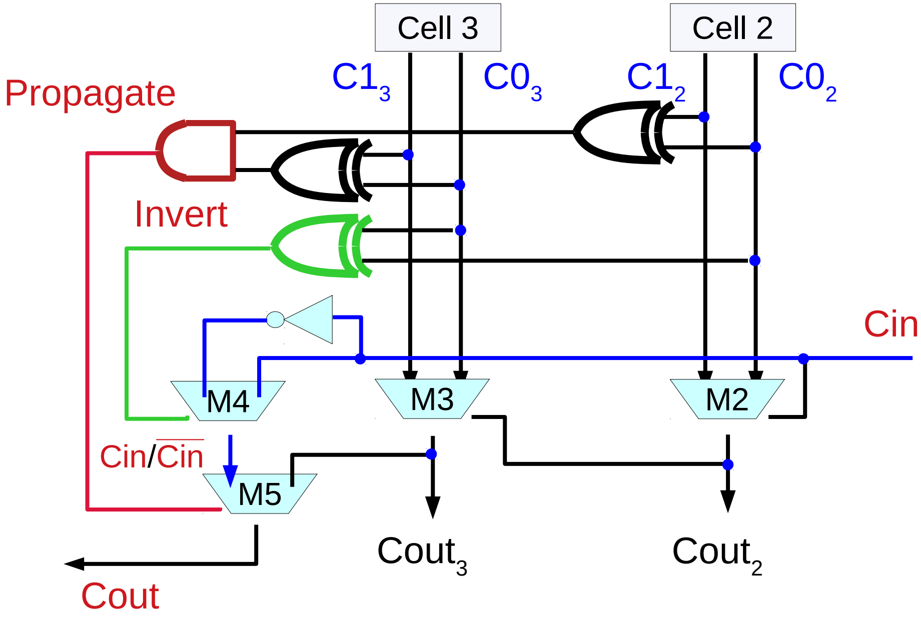
by ANDing together the XOR of each stage's C1 and C0 signal

If so, we know that the Cout function

- Cin
- Cin bar.

https://en.wikipedia.org/wiki/Carry-lookahead_adder

C1	C0	Name	
0	0	0	Kill
0	1	$\overline{\text{Cin}}$	Inverse Propagate
1	0	Cin	Propagate
1	1	1	Generate

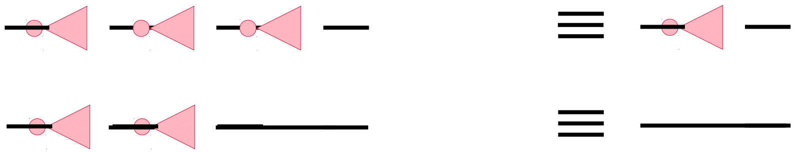


Variable Block

Invert is used only when
Propagate is true

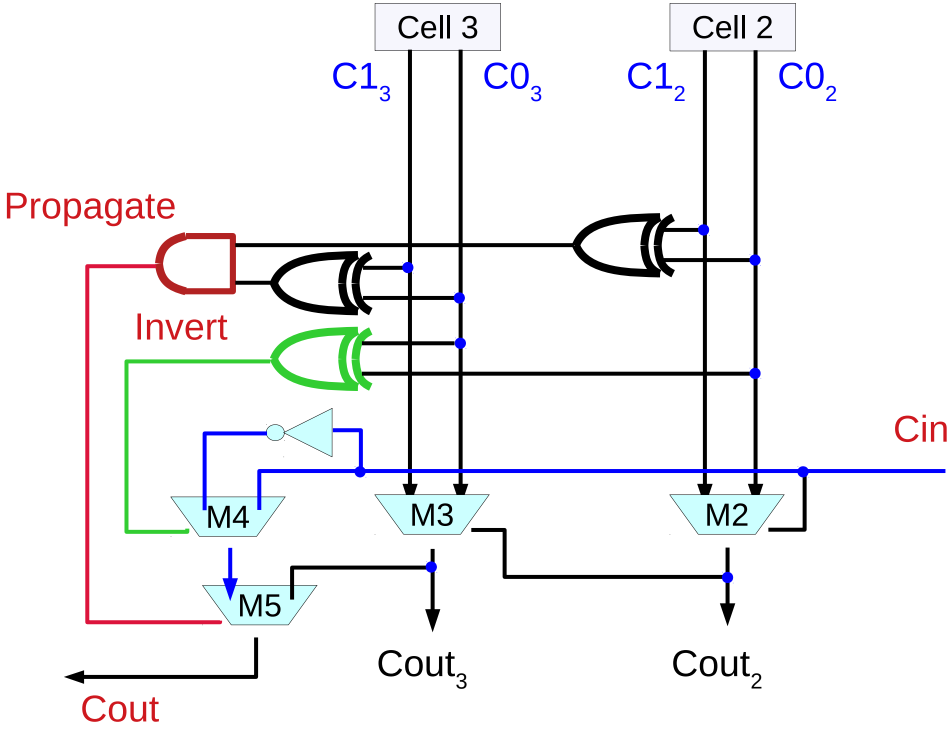
Propagate is true
each cell in the carry chain
has either propagate condition (C1=1, C0=0)
or inverse propagate condition (C1=0, C0=1)

If the number of inverse propagate cells is **odd**
then Invert becomes true



https://en.wikipedia.org/wiki/Carry-lookahead_adder

$$C1 = X + Y$$
$$C0 = X \cdot Y$$

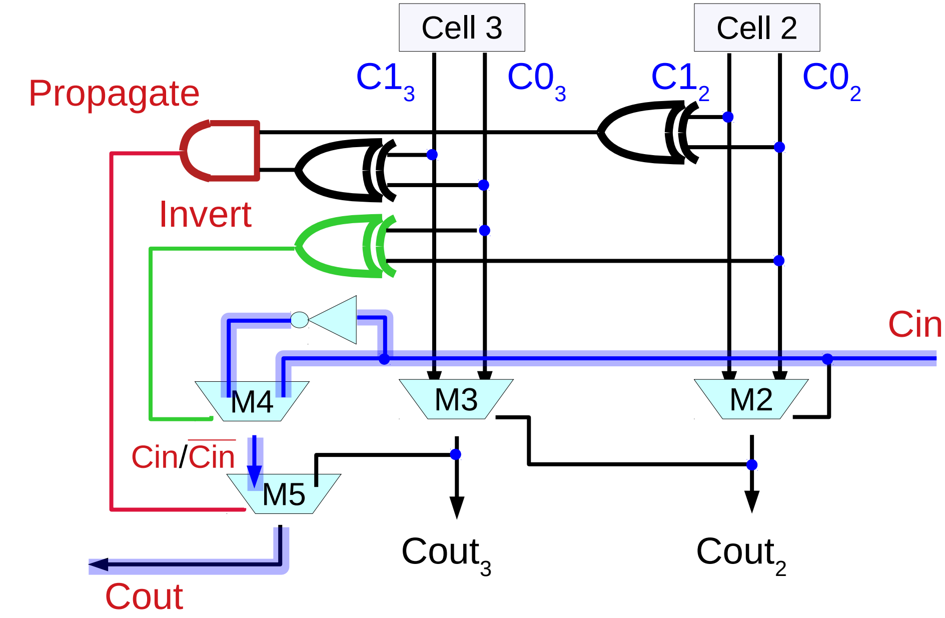


Variable Block

to decide whether to invert the signal or not, we must determine how many cells are in inverse propagate mode.

if the number is even (including zero), the output is *not* inverted, while if the number is odd, the output is inverted.

Propagate	bypass	$Cin/\overline{Cin}$
No propagate	ripple carry	



https://en.wikipedia.org/wiki/Carry-lookahead_adder

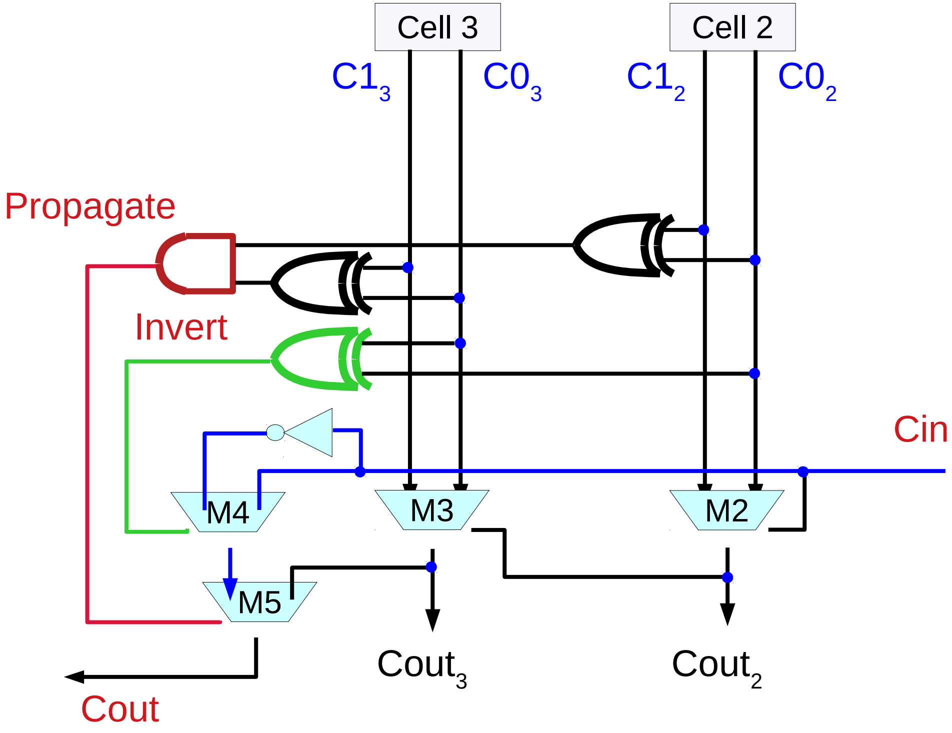
Variable Block

the inversion check can be done
by looking for inverse signal C0 from each cell.

if this signal C0 is true,
the cell is in either generate or
inverse propagate mode,

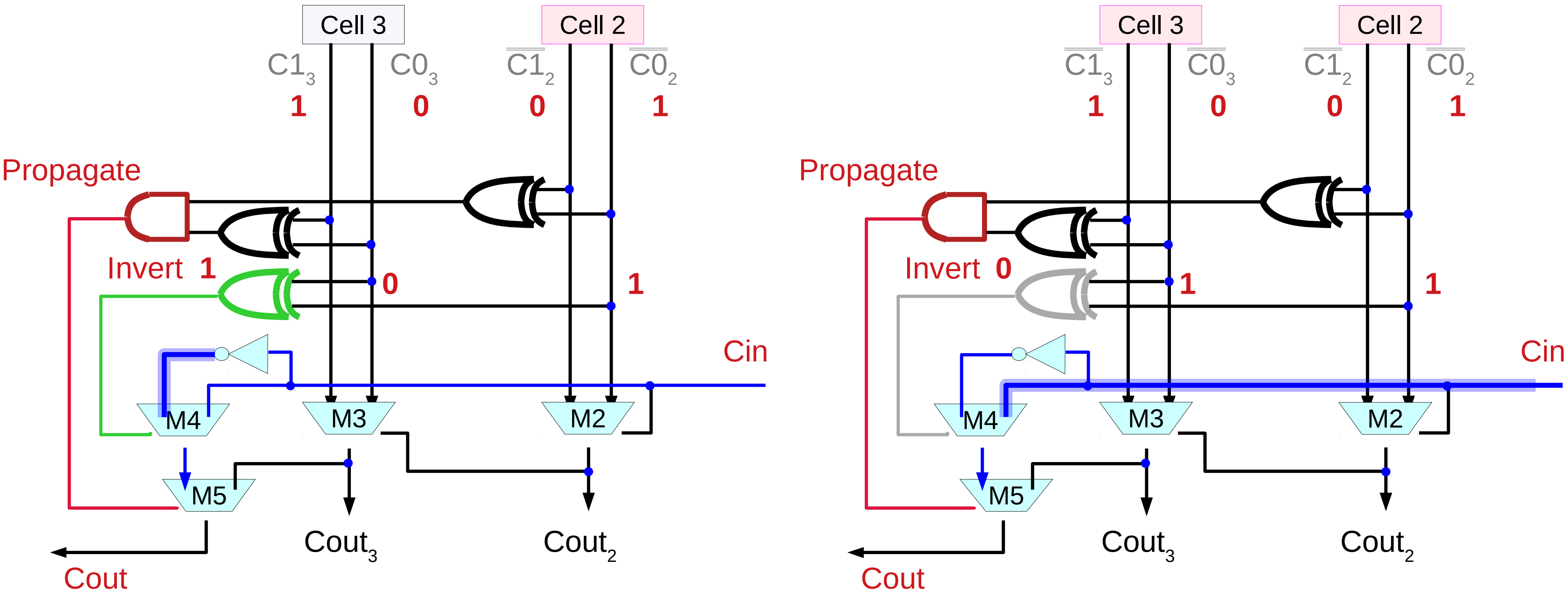
if it is in generate mode
inversion signal will be ignored anyway

we only consider inverting the Cin signal
if all cells are in some form of propagate mode



https://en.wikipedia.org/wiki/Carry-lookahead_adder

Variable Block



https://en.wikipedia.org/wiki/Carry-lookahead_adder

Not in propagate mode (1)

The Cin still ripples through the block itself, since
the intermediate carry values must also be computed

P	p
G	G
P’G’	P’G’

If any of the cells in the carry chain
are not in propagate mode, (**G** or **P’G’**)
the Cout output is generated normally
by the ripple carry chain.

that is since there is some cell in the block
that is not in propagate mode,
it must be in generate or kill mode, (**G** or **P’G’**)
and thus the block's Cout output does not depend
on the block's Cin input

P	p
P	G
P	P’G’
G	p
G	G
G	P’G’
P’G’	p
P’G’	G
P’G’	P’G’

https://en.wikipedia.org/wiki/Carry-lookahead_adder

Not in propagate mode (2)

While this carry chain does start
at the block's Cin signal,
and leads to the block's Cout,
this long path is a false path

P	p
G	G
P’G’	P’G’

P	p
P	G
P	P’G’
G	p
G	G
G	P’G’
P’G’	p
P’G’	G
P’G’	P’G’

https://en.wikipedia.org/wiki/Carry-lookahead_adder

Variable Block

note that for both of these tests
we can use a tree of gates to compute the result.

Also, since we ignore the inversion signal
when we are not bypassing the carry chain
we can use C1 as the inverse of C0
for the inversion signal's computation,
which avoids the added inverter in the XOR gate

C1	C0	Name	
0	0	0	Kill
0	1	$\overline{Cin}$	Inverse Propagate
1	0	Cin	Propagate
1	1	1	Generate

$$C1 = X + Y$$

$$C0 = X \cdot Y$$

X	Y	C1	C0	
0	0	0	0	$\overline{X}\ \overline{Y}$
0	1	1	0	$\overline{X}\ Y$
1	0	1	0	$X\ \overline{Y}$
1	1	1	1	$X\ Y$

Bypassing mode

C0 as the inverse of C1
When not bypassing

https://en.wikipedia.org/wiki/Carry-lookahead_adder

Propagate

Propagate $(C1_3 \oplus C0_3) \cdot (C1_2 \oplus C0_2)$

P P

$C1 = X + Y$

$C0 = X \cdot Y$

$C1 \oplus C0 = X \oplus Y$ = P

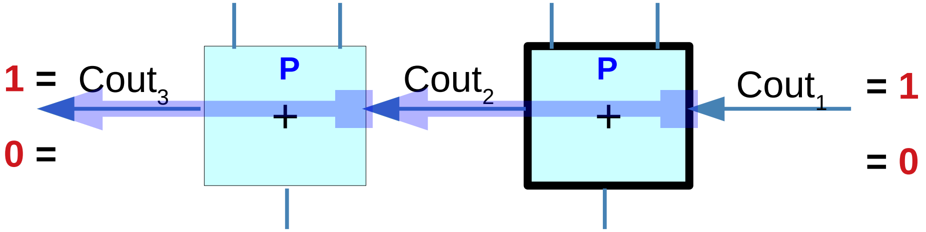
P G

G

P G' = P

G G' = 0

P'G' G = 0



Invert

Invert

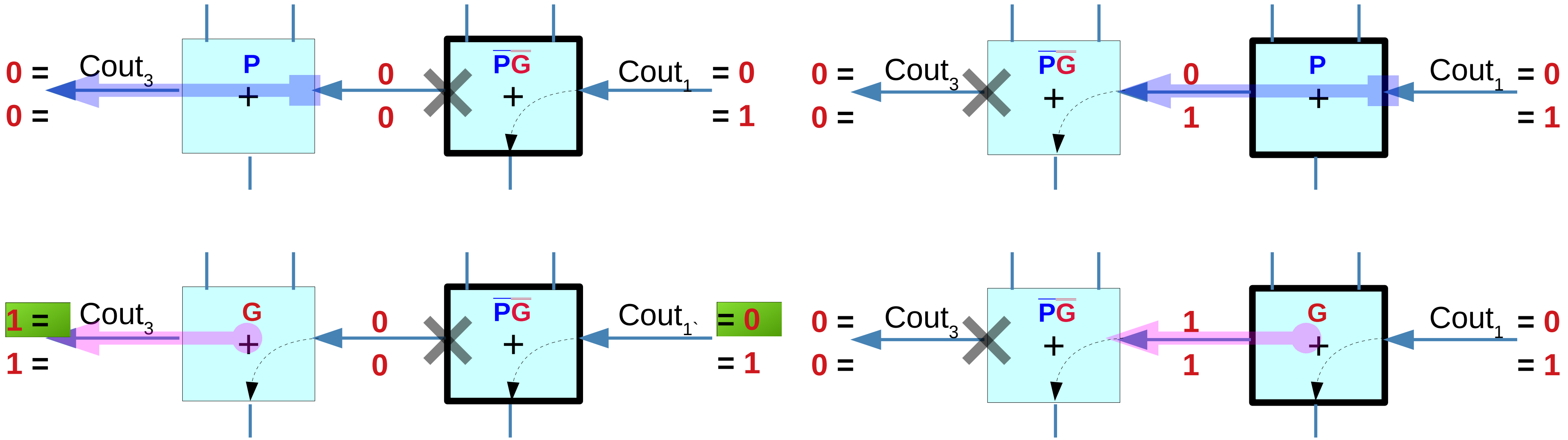
$$(C1_3 \oplus C1_2)$$

P	P
G	G
P	P'G'
G	P'G'
P'G'	P
P'G'	G

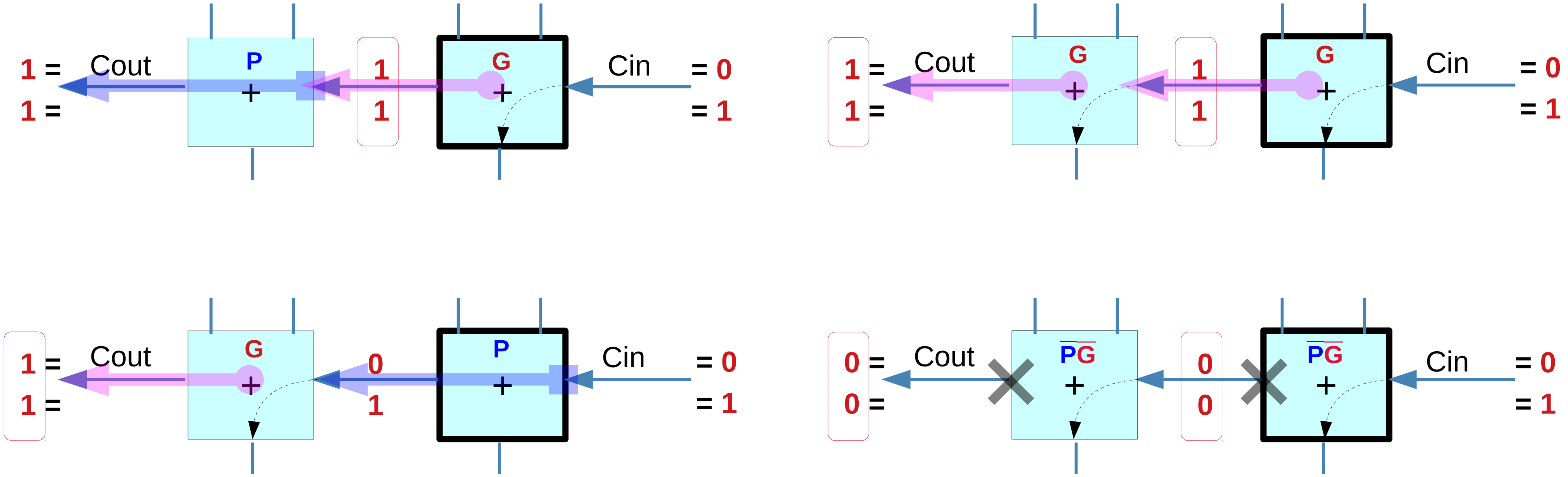
$$C1 = X+Y$$
$$C0 = X \cdot Y$$

$$C1 \oplus C0 = X \oplus Y$$ = P

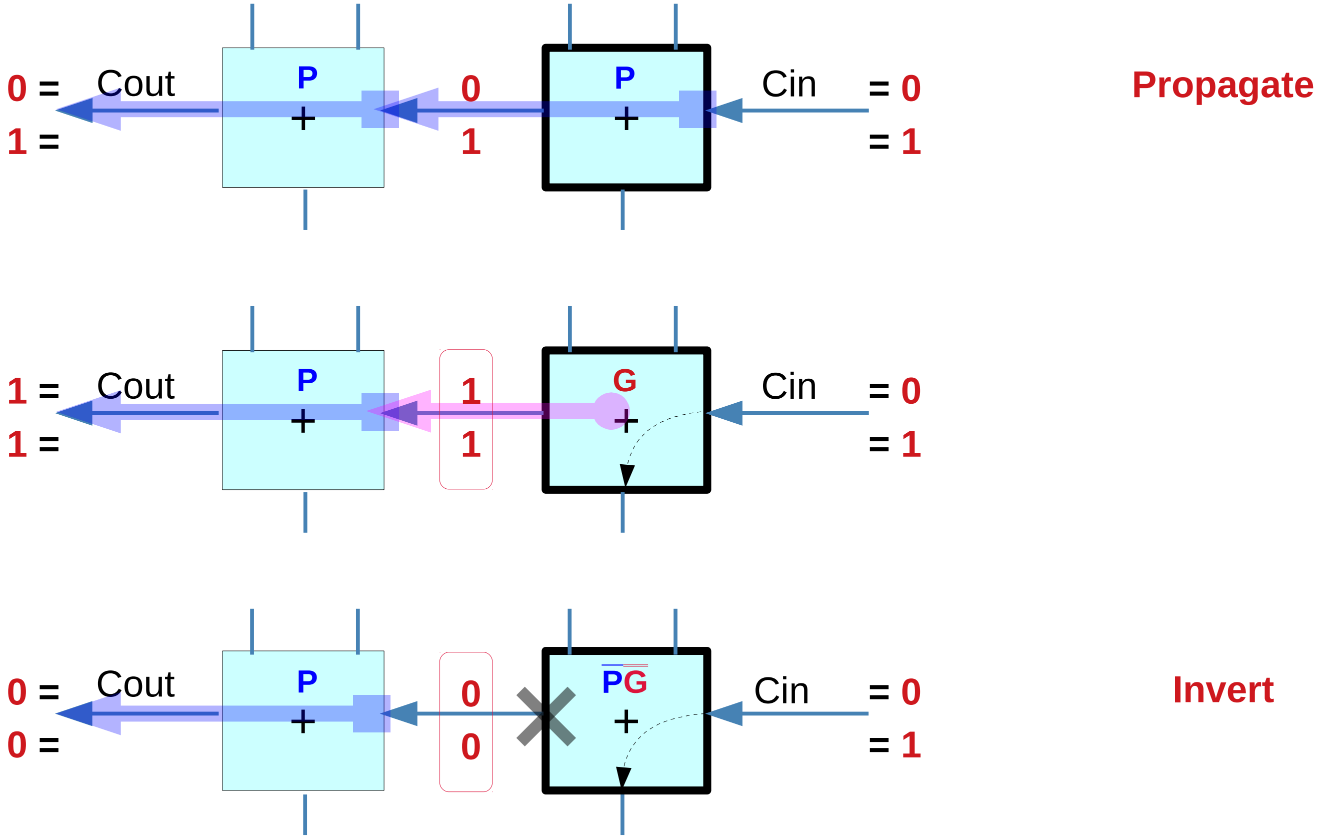
P	G
G	



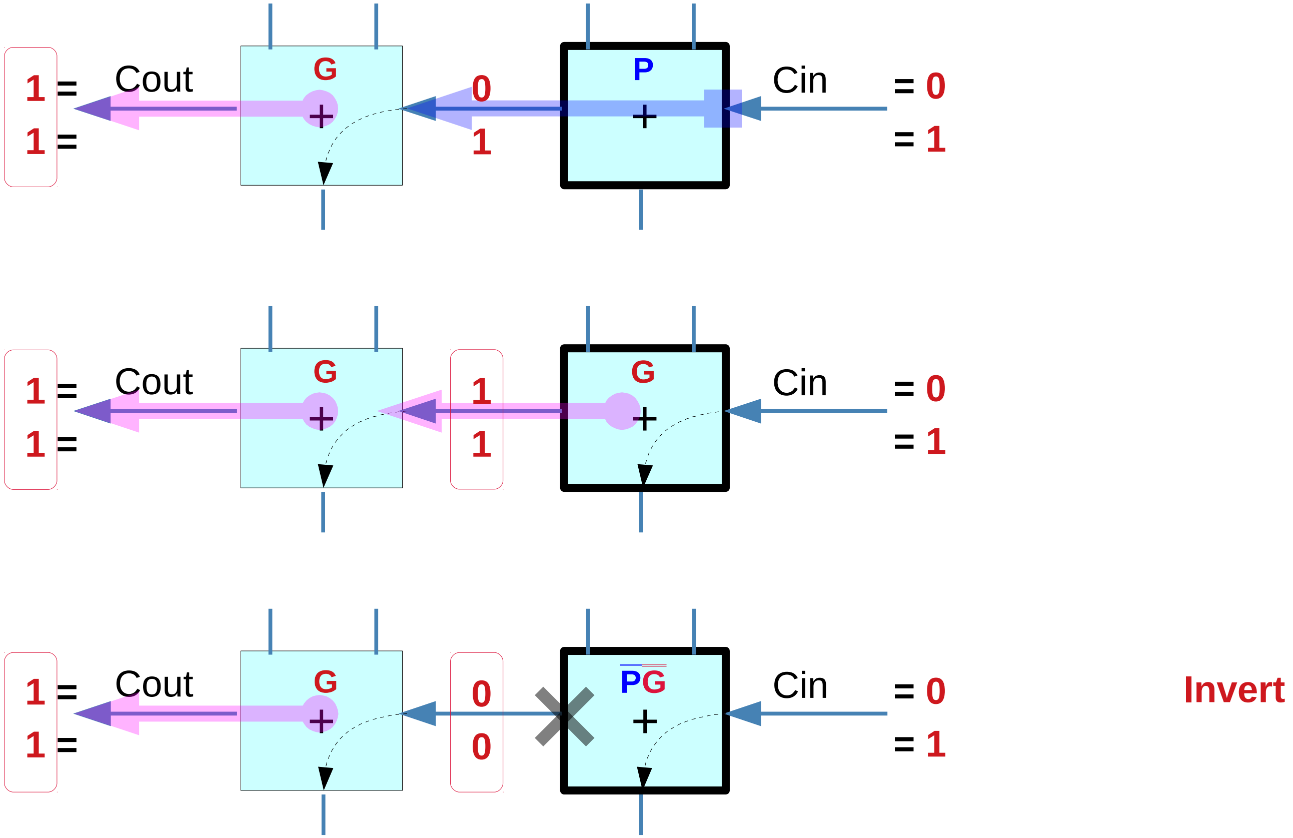
Variable Block



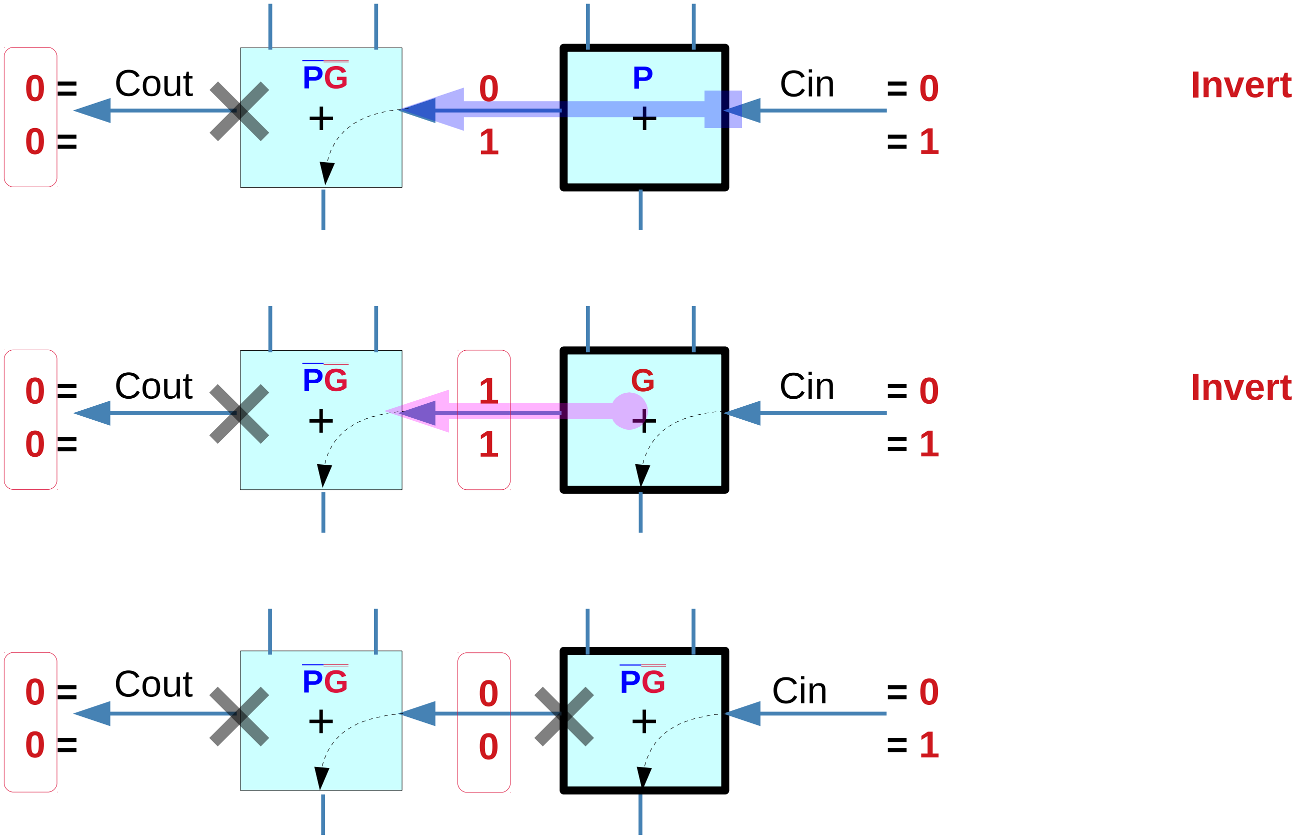
Variable Block



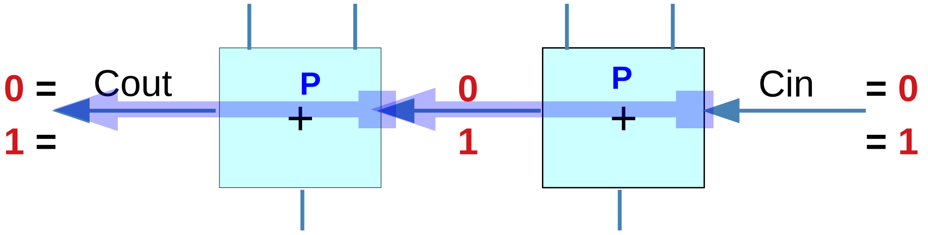
Variable Block



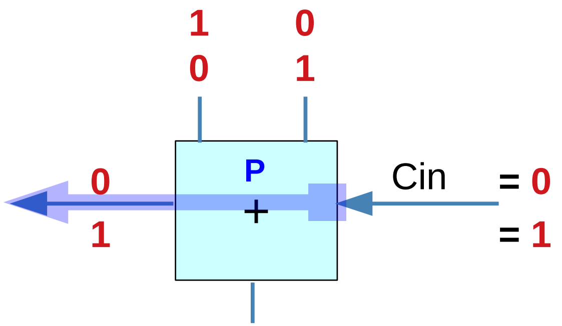
Variable Block

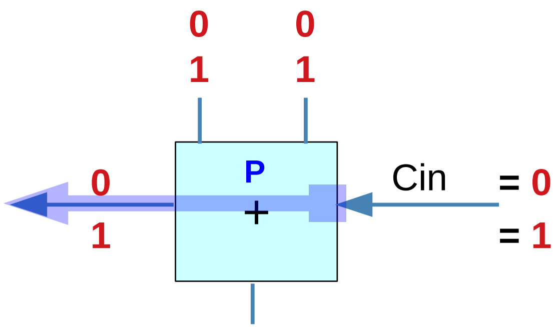

0 =
0 =
Cout
$\overline{P}\,\overline{G}$
+
0
1
P
+
Cin
= 0
= 1
Invert
0 =
0 =
Cout
$\overline{P}\,\overline{G}$
+
1
1
G
+
Cin
= 0
= 1
Invert
0 =
0 =
Cout
$\overline{P}\,\overline{G}$
+
0
0
$\overline{P}\,\overline{G}$
+
Cin
= 0
= 1

Variable Block



Propagate


1
0
0
1
0
1
P
+
Cin
= 0
= 1


0
1
0
1
0
1
P
+
Cin
= 0
= 1