

FPGA Variable Block Adder (1C)

-
-

Copyright (c) 2021 - 2010 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using OpenOffice and Octave.

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**.

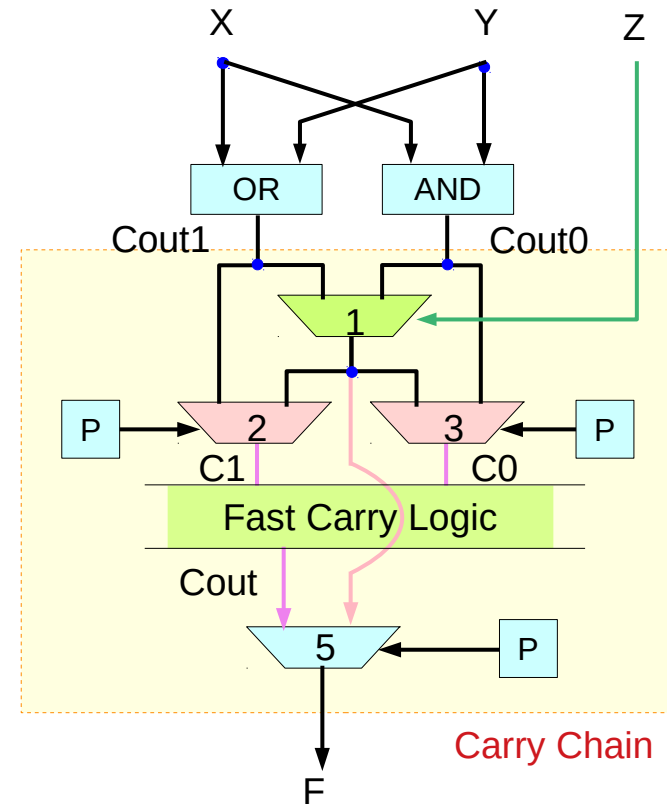
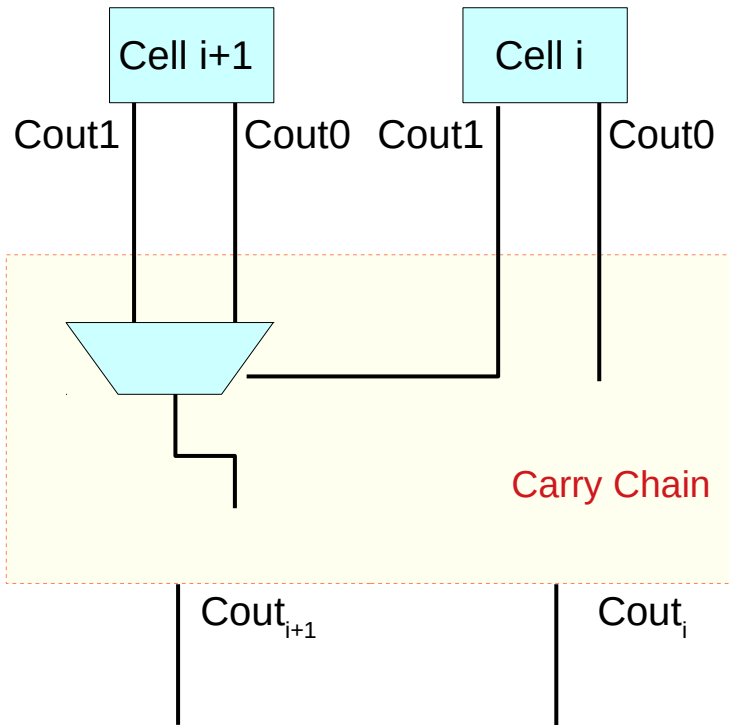
Instead, the inversion of the Cout0 signal of the left LUT will cause the select line of Mux 1 to be inverted.

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>

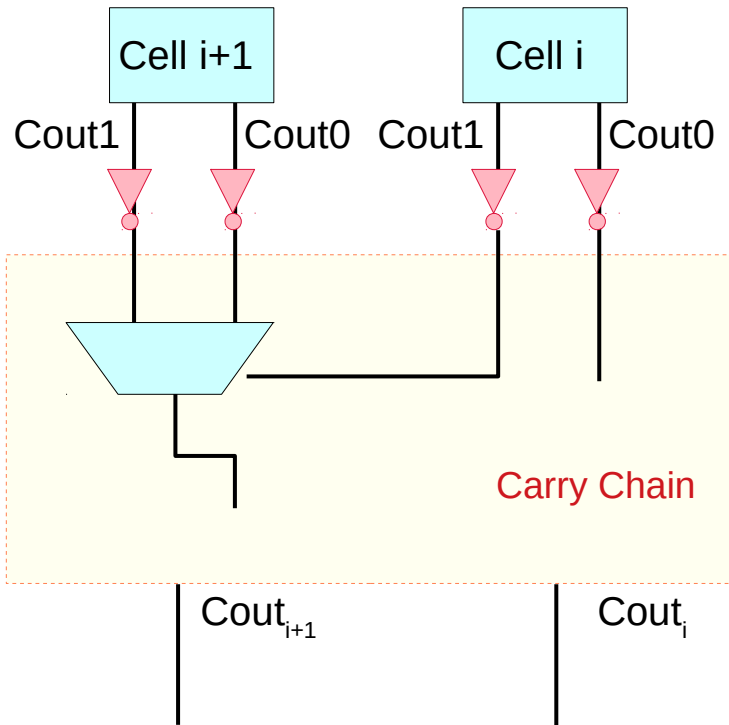
Carry Logic and the FPAG Cell Structure (2)



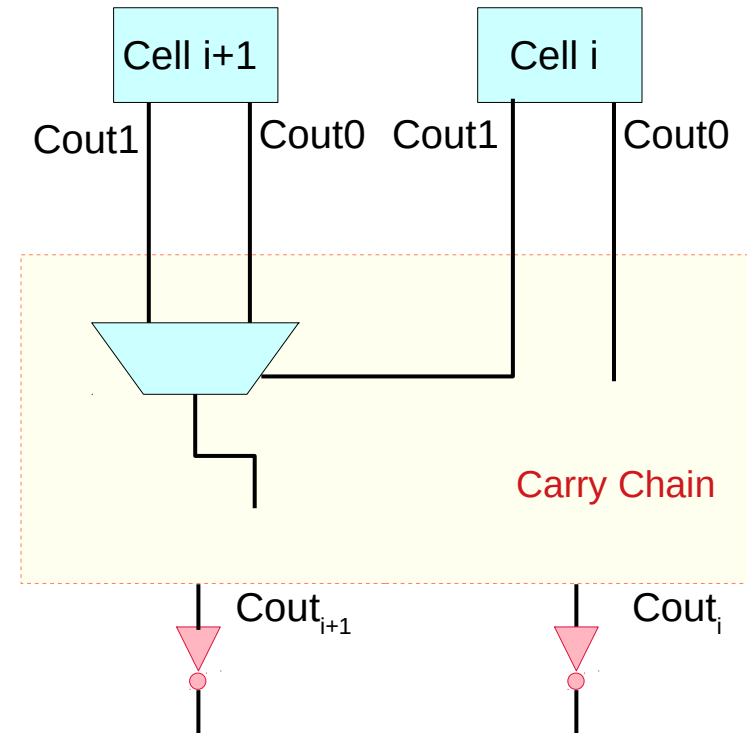
$$Cout_i = (Cout_{i-1} \cdot C1_i) + (\overline{Cout_{i-1}} \cdot C0_i)$$

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.

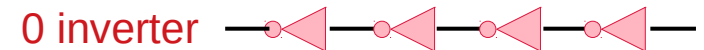
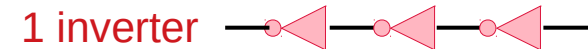
Assumption

there are chains of **inverters** that are placed within an FPGA cell either **before** or **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



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.

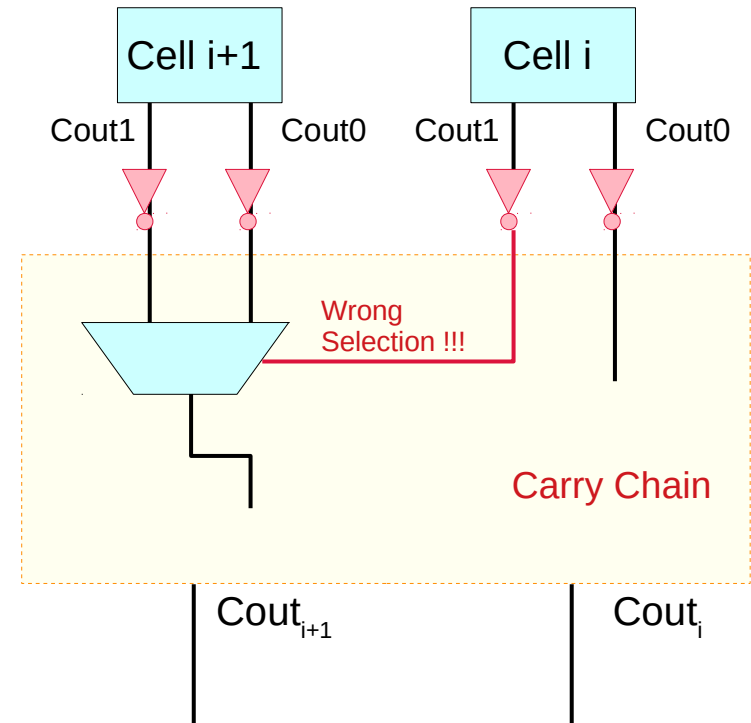
High Performance Carry Chains for FPGAs, M. M. Hosler, <https://people.ece.uw.edu>

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.

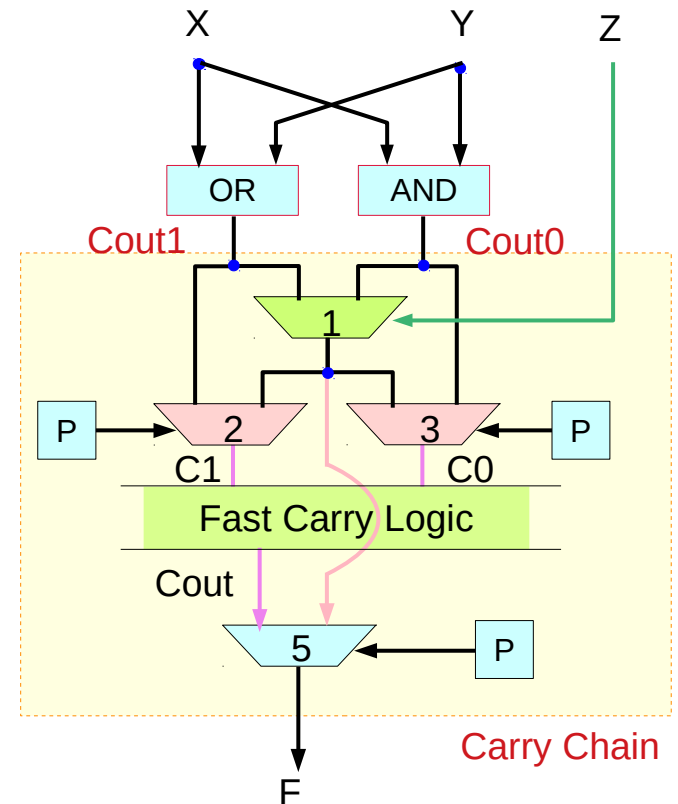


Case 1:
inverters **before** a simple carry chain

Case I – inverter before the carry chain (2)

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

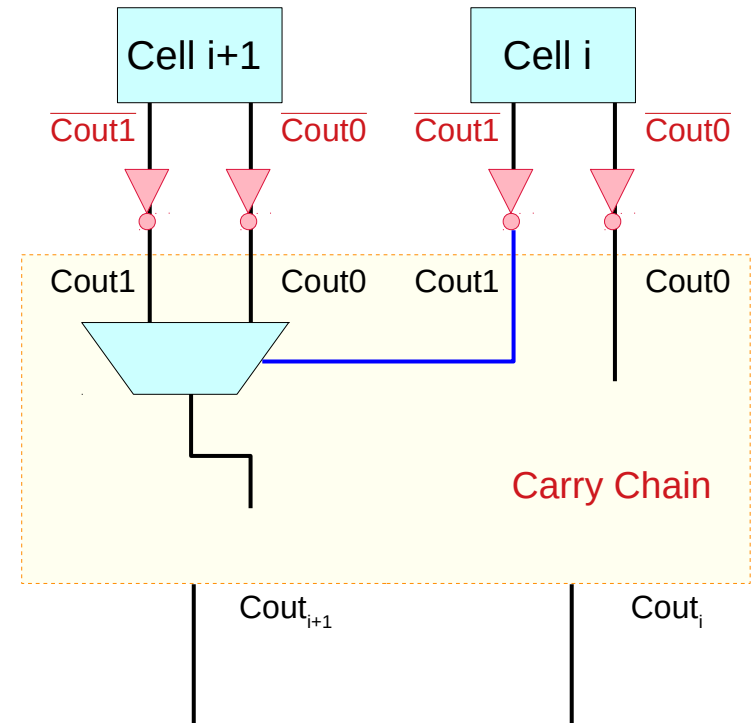
these outputs are generated by the 2-LUTs based on a user-programmable function of X and Y.



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 .



Case 1:
inverters **before** a simple carry chain

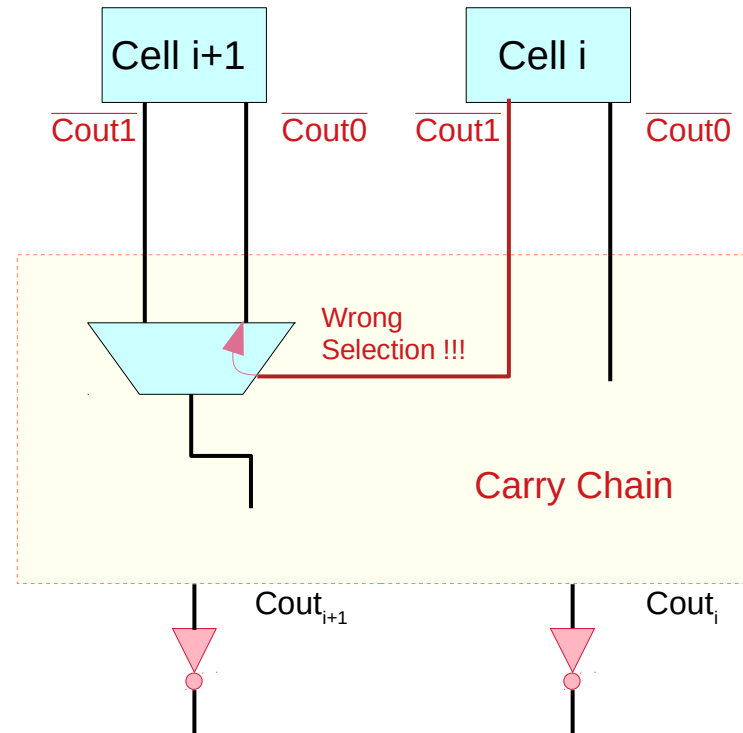
Case II – inverter after the carry chain (1)

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.



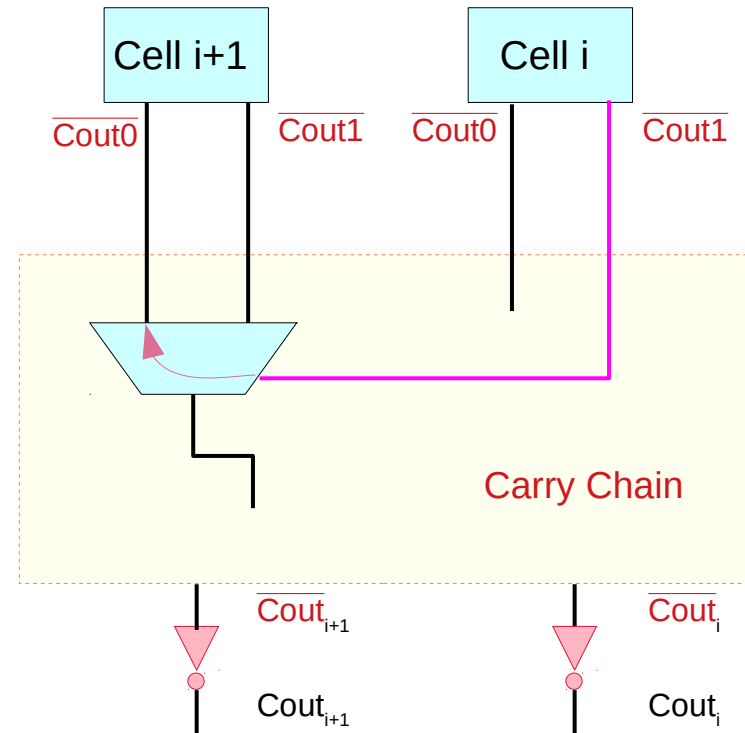
Case II – inverter after the carry chain (2)

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.



Case II – inverter after the carry chain (3)

Now, the carry chain works properly again.

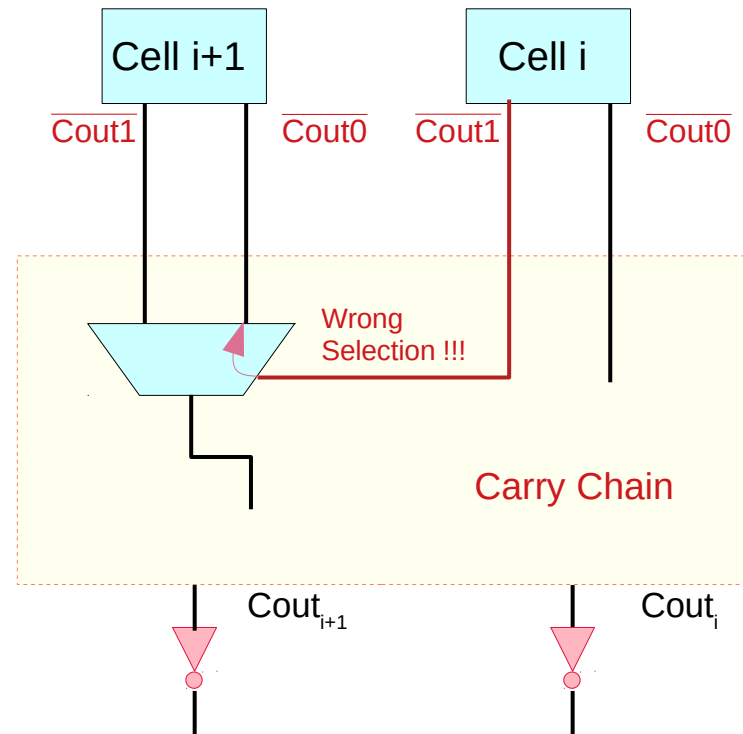
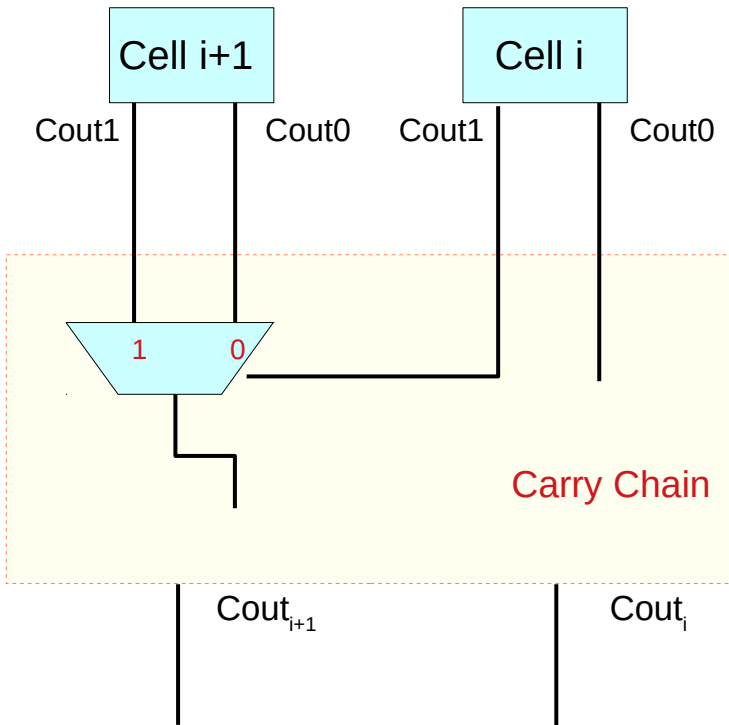
Inverting the inputs to the carry chain causes the select lines of the muxes to choose the wrong inputs.

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.

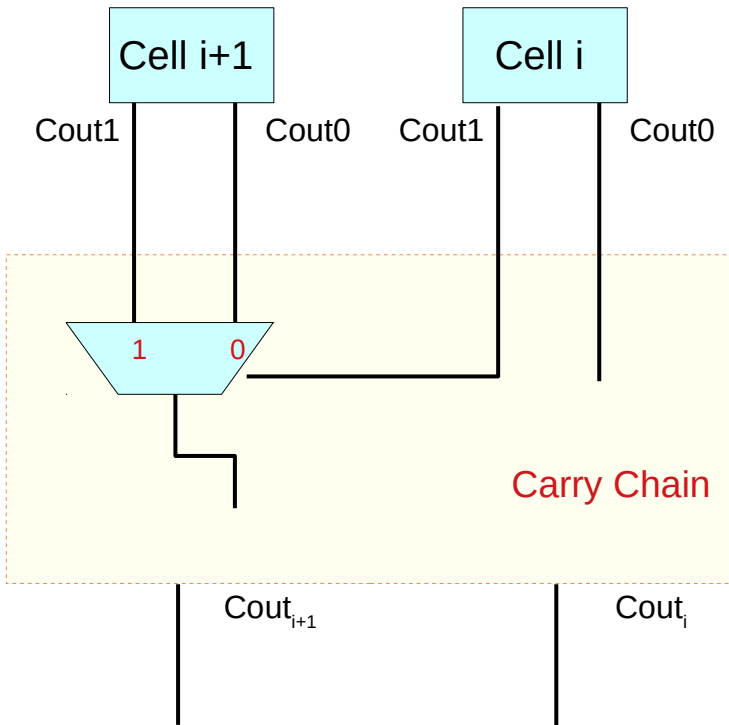
However, since there is one logical inverter after the carry chain, the final solution is equivalent to the original solution.

Case II – inverter after the carry chain (4)

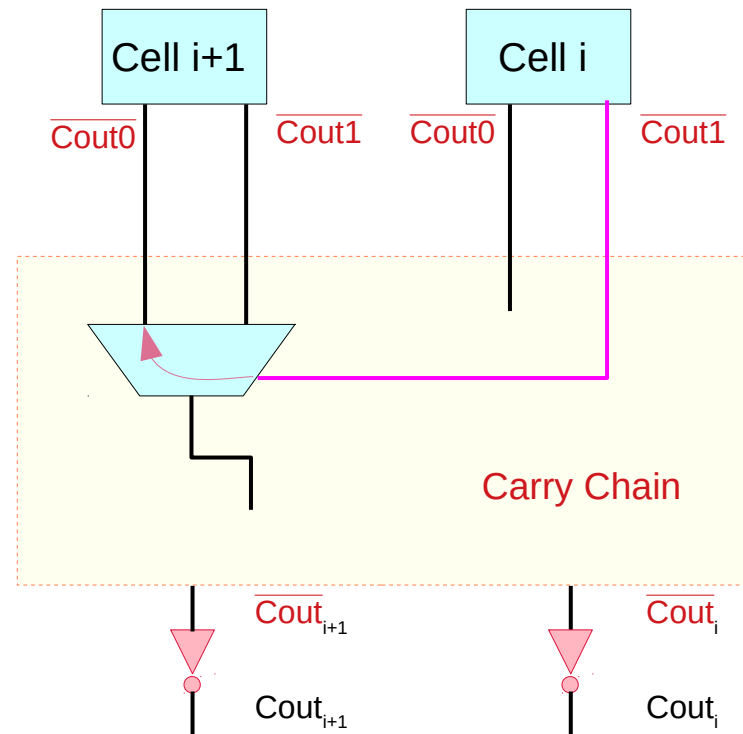


High Performance Carry Chains for FPGAs, M. M. Hosler, <https://people.ece.uw.edu>

Case II – inverter after the carry chain (5)



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



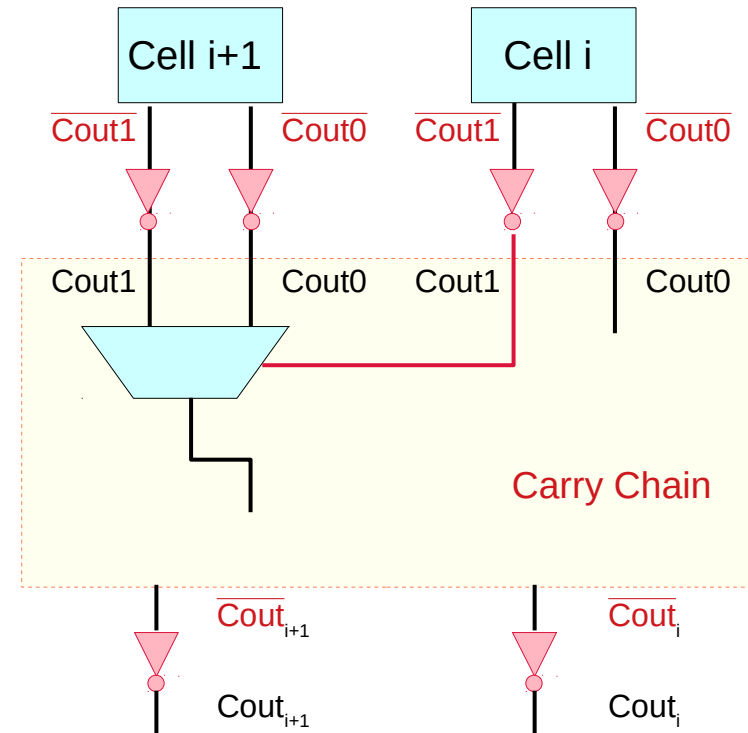
If Cout1 of Cell i is **0**,
then **Cout0** of Cell i+1 is selected
If Cout0 of Cell i is **1**,
then **Cout1** of Cell i+1 is selected

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.

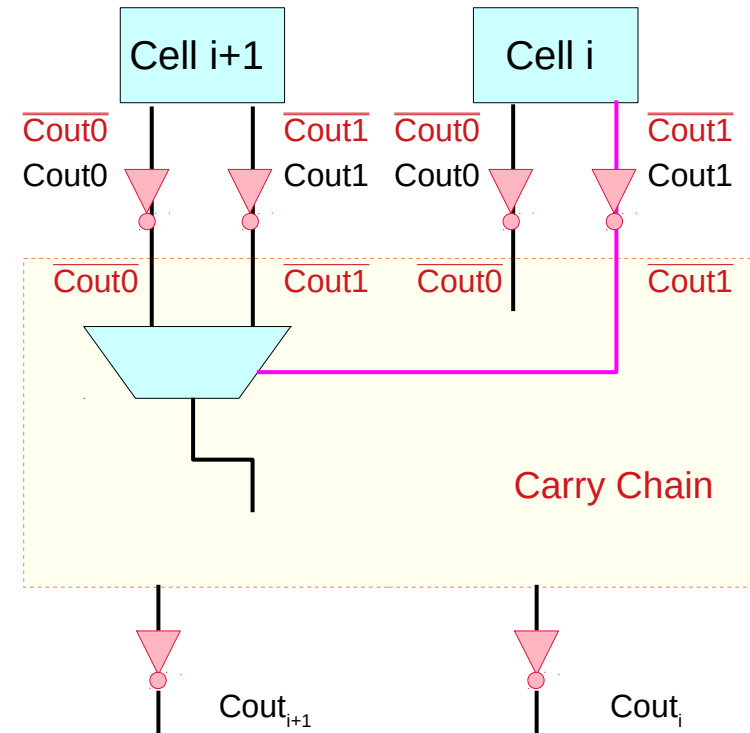
For example, if there are inverters both **before** and **after** the carry chains, then first **Case 1** is applied to the cells to **negate** the **inverters before** the carry chain. Thus, **Cout1** and **Cout0** are **inverted**.



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.



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.

Ripple Carry Structure

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

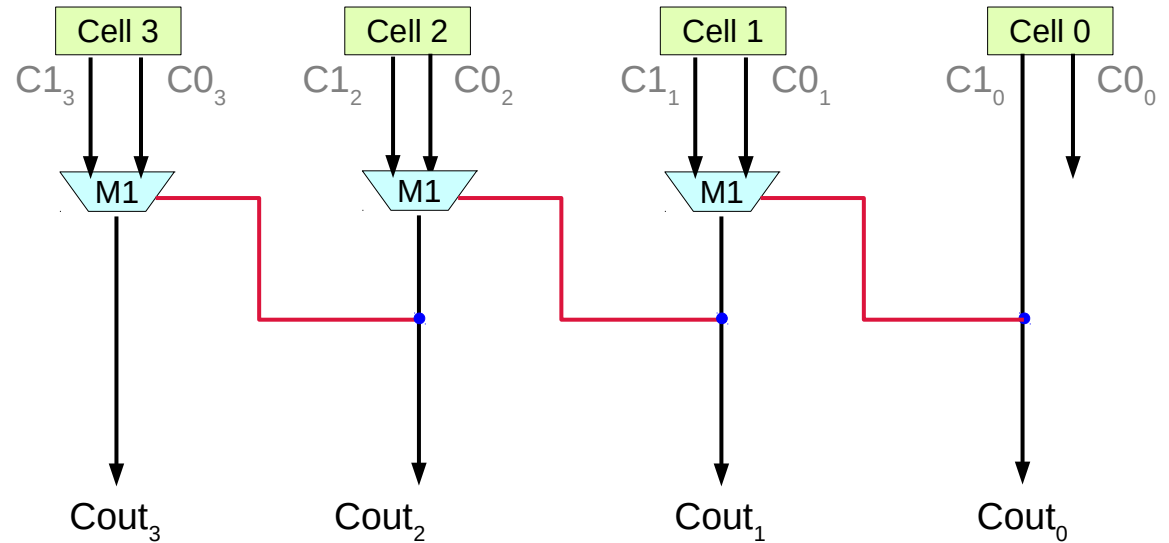
the carry computation for the first two cells is performed with the simple **ripple-carry** structure implemented by **mux1**

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

$$C1 = X + Y$$

$$C0 = X \cdot Y$$

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



Cout using C1, C0, Cin

X	Y	C1	C0	
0	0	0	0	$\bar{X}\bar{Y}$
0	1	1	0	$\bar{X}Y$
1	0	1	0	$X\bar{Y}$
1	1	1	1	XY

$$C1 = X + Y$$

$$C0 = X \cdot Y$$

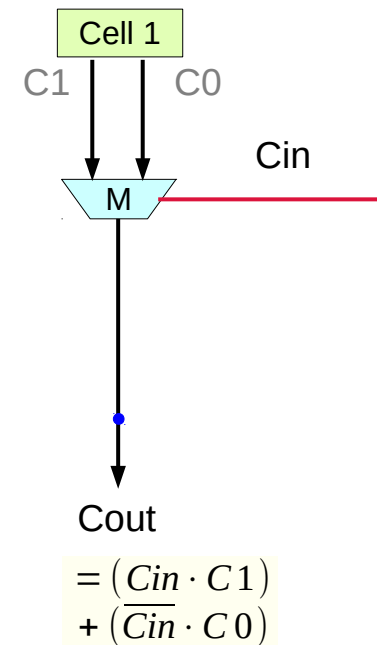
C1	C0		Name
0	0	0	Kill
0	1	\bar{Cin}	Inverse Propagate
1	0	Cin	Propagate
1	1	1	Generate

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

$$(Cin \cdot C1) = Cin \cdot (\bar{X}Y + X\bar{Y} + XY) \rightarrow \text{propagate } Cin$$

$$(\bar{Cin} \cdot C0) = \bar{Cin} \cdot XY \rightarrow \text{generate a new carry}$$

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



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

Ripple Carry Structure

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

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

Ripple Carry Structure

$$C_{out} = (C_{in} \cdot C_1) + (\overline{C_{in}} \cdot C_0)$$

$$C_1 = X + Y$$

$$C_0 = X \cdot Y$$

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

$$\overline{C_{out}} = \overline{(C_{in} \cdot C_1) + (\overline{C_{in}} \cdot C_0)}$$

$$= \overline{(C_{in} \cdot C_1)} \cdot \overline{(\overline{C_{in}} \cdot C_0)}$$

$$= (\overline{C_{in}} + \overline{C_1}) \cdot (C_{in} + \overline{C_0})$$

$$= \overline{C_{in}}C_{in} + \overline{C_{in}}\overline{C_0} + \overline{C_1}C_{in} + \overline{C_1}\overline{C_0}$$

$$= \overline{C_{in}}\overline{C_0} + \overline{C_1}C_{in} + \overline{C_1}\overline{C_0}$$

$$= \overline{C_{in}}\overline{C_0} + \overline{C_1}C_{in}$$

	X	Y	$\overline{C_1}$	$\overline{C_0}$	$C_{in} \cdot \overline{C_1}$	$\overline{C_{in}} \cdot \overline{C_0}$	$C_{in} \cdot \overline{C_1} + \overline{C_{in}} \cdot \overline{C_0}$	$\overline{C_1} \cdot \overline{C_0}$
	0	0	1	1	Cin	$\overline{C_{in}}$	1	1
Inverse Propagate	0	1	0	1	0	$\overline{C_{in}}$	$\overline{C_{in}}$	0
	1	0	0	1	0	$\overline{C_{in}}$	$\overline{C_{in}}$	0
	1	1	0	0	0	0	0	0

redundant

Ripple Carry Structure

$$C_{out} = C_{in} \cdot C_1 + \overline{C_{in}} \cdot C_0$$

- the wire **Cout** has the value of Cout
- the wire **C0** has the value of C0
- the wire **C1** has the value of C1

	X	Y	C1	C0	Cin·C1	$\overline{C_{in}} \cdot C_0$	C1·C0
	0	0	0	0	0	0	0
Propagate	0	1	1	0	Cin	0	Cin
	1	0	1	0	Cin	0	Cin
	1	1	1	1	Cin	$\overline{C_{in}}$	1

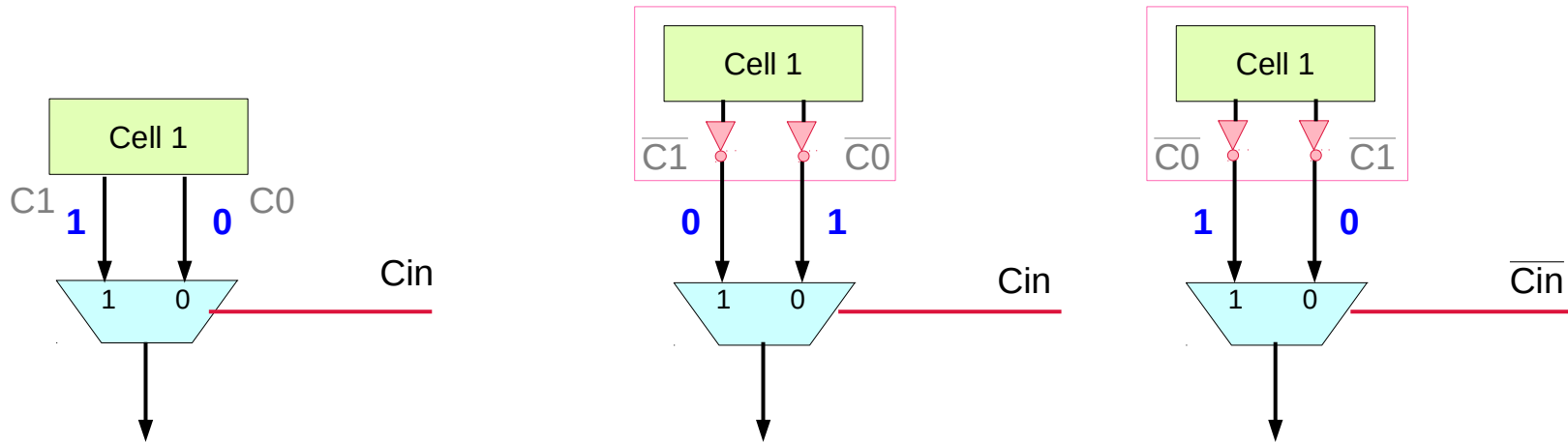
$$\overline{C_{out}} = \overline{C_{in}} \cdot \overline{C_0} + \overline{C_1} \cdot C_{in}$$

- the wire **Cout** has the value of $\overline{C_{out}}$
- the wire **C1** has the value of $\overline{C_1}$
- the wire **C0** has the value of $\overline{C_0}$

	X	Y	$\overline{C_1}$	$\overline{C_0}$	Cin· $\overline{C_1}$	$\overline{C_{in}} \cdot \overline{C_0}$	$\overline{C_1} \cdot C_{in}$
	0	0	1	1	Cin	$\overline{C_{in}}$	1
Inverse Propagate	0	1	0	1	0	$\overline{C_{in}}$	$\overline{C_{in}}$
	1	0	0	1	0	$\overline{C_{in}}$	$\overline{C_{in}}$
	1	1	0	0	0	0	0

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

Cout using C1, C0, Cin



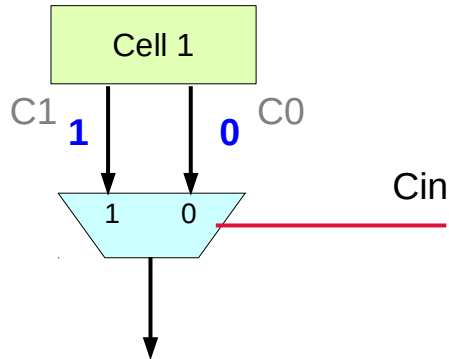
During optimization process, inverters may be added before the node **C1** and **C0**

- 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

Cout using C1, C0, Cin

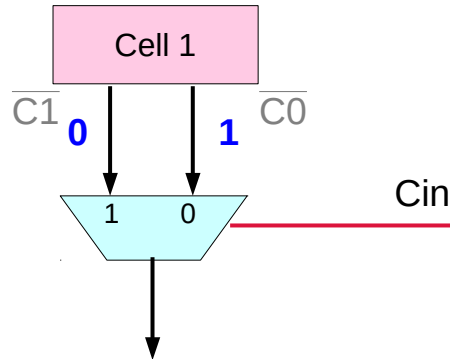


$$\text{Cout} = (\text{Cin} \cdot \text{C1}) + (\overline{\text{Cin}} \cdot \text{C0})$$

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

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

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

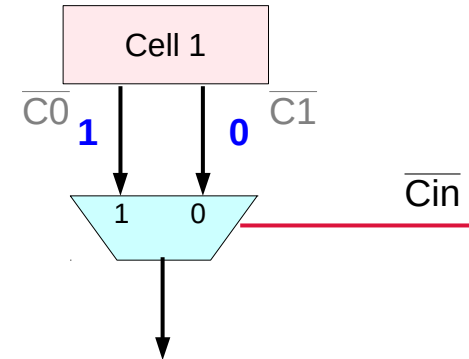


$$\overline{\text{Cout}} = (\overline{\text{Cin}} \cdot \overline{\text{C0}}) + (\text{Cin} \cdot \overline{\text{C1}})$$

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

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

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



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

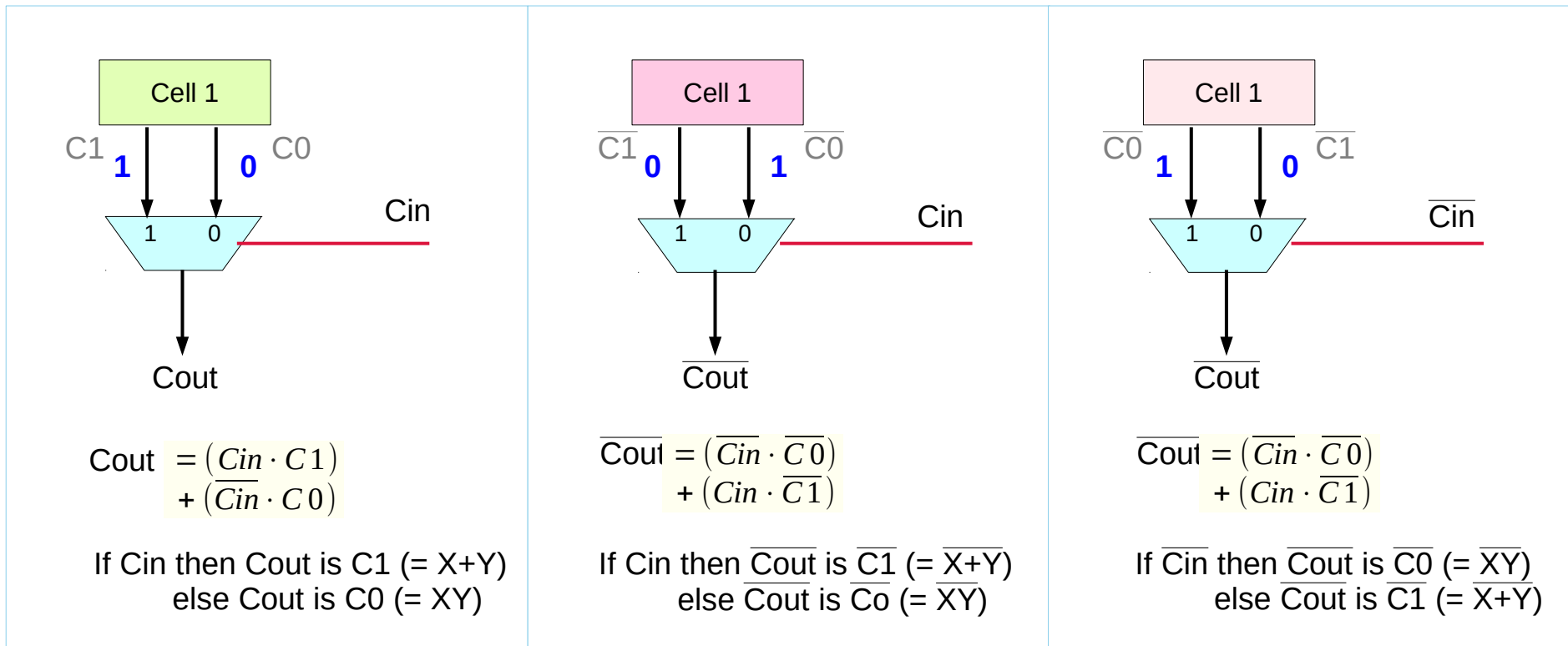
If $\overline{\text{Cin}}$ then $\overline{\text{Cout}}$ is $\overline{\text{C0}}$ (= \overline{XY})
else Cout is C1 (= X+Y)

Cout using C1, C0, Cin

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

$$C1 = X+Y$$

$$C0 = X \cdot Y$$



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

Cout using C1, C0, Cin

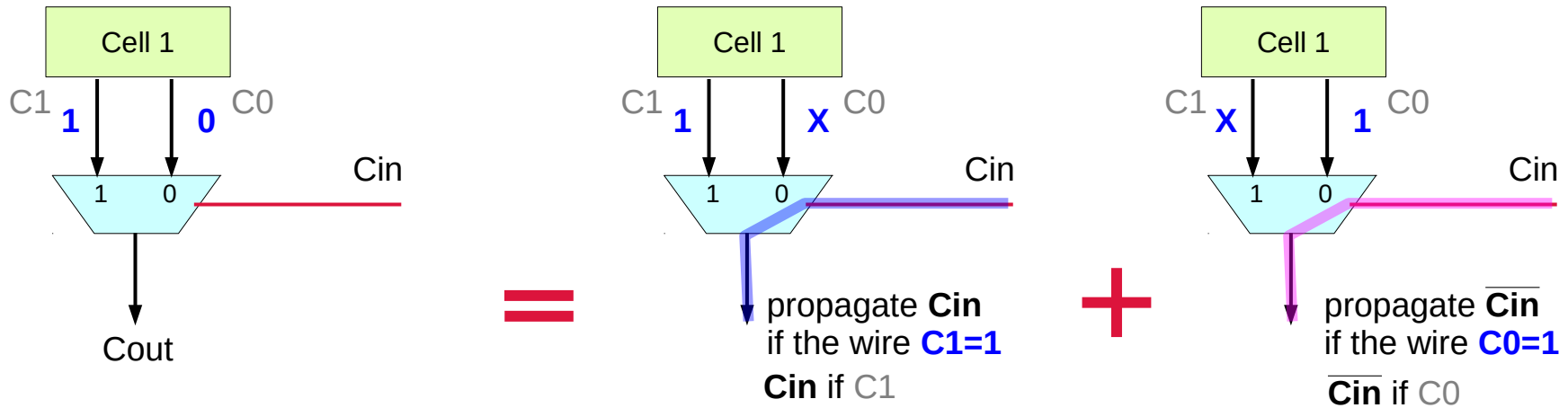
$$C1 = X + Y$$

$$C0 = X \cdot Y$$

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

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

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



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

Cout using C1, C0, Cin

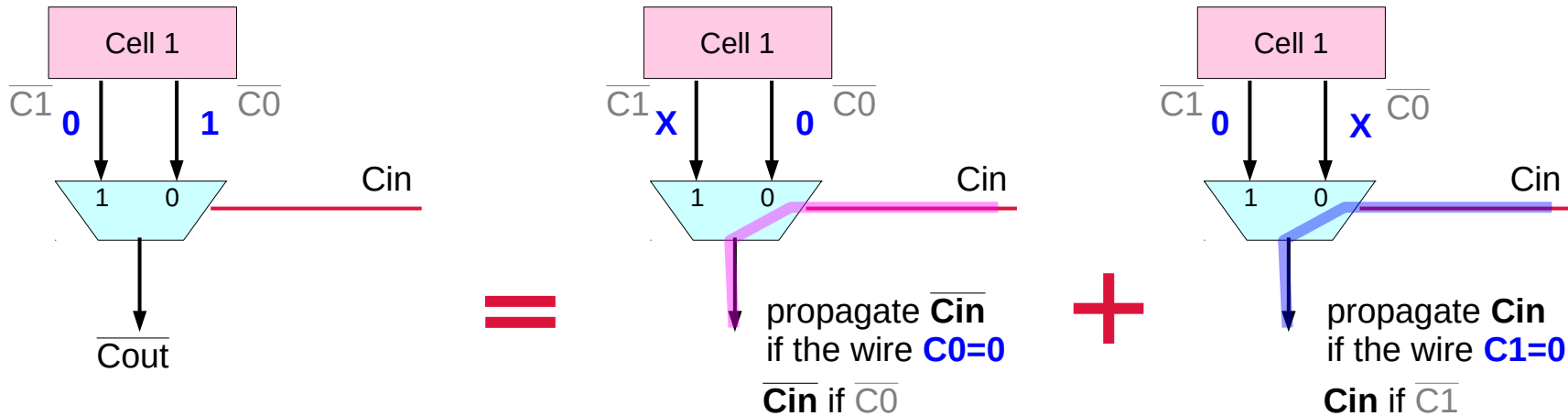
$$C1 = X + Y$$

$$C0 = X \cdot Y$$

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

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

\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



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

Cout using C1, C0, Cin

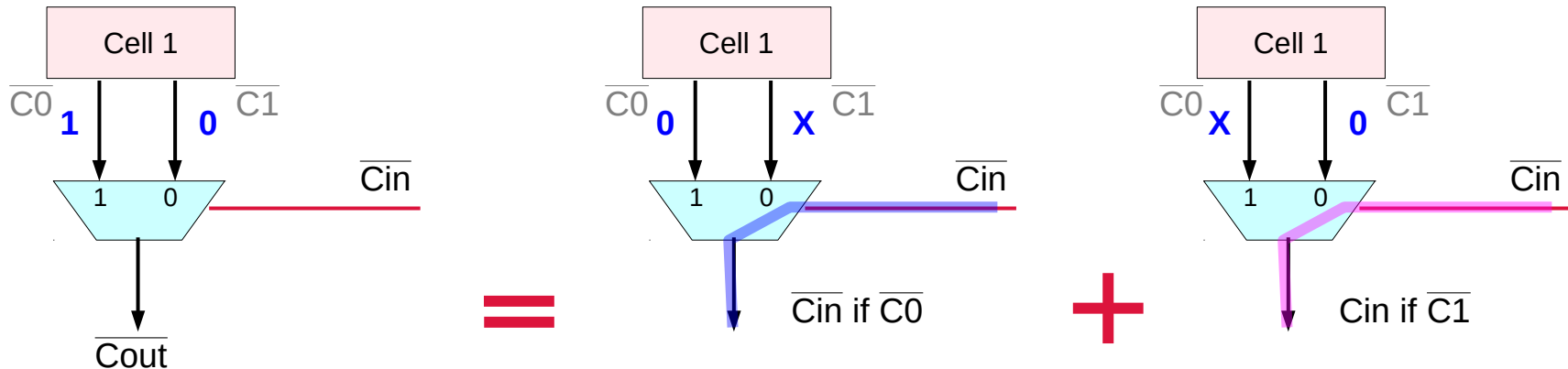
$$C1 = X + Y$$

$$C0 = X \cdot Y$$

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

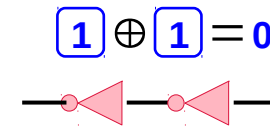
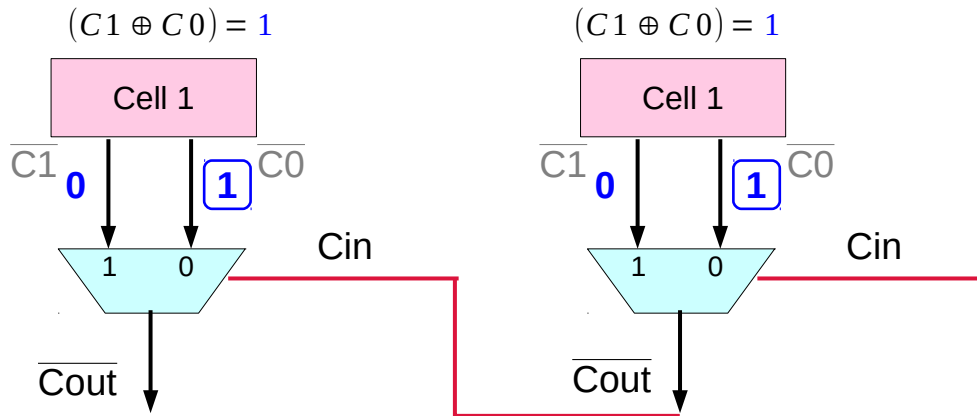
If \overline{Cin} then \overline{Cout} is $\overline{C0}$ ($= \overline{XY}$)
 else \overline{Cout} is $\overline{C1}$ ($= \overline{X+Y}$)

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



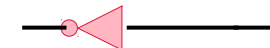
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin

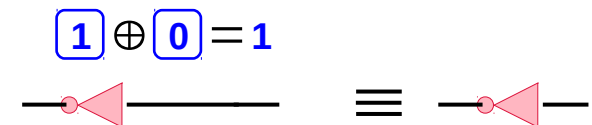
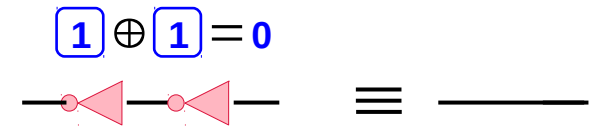
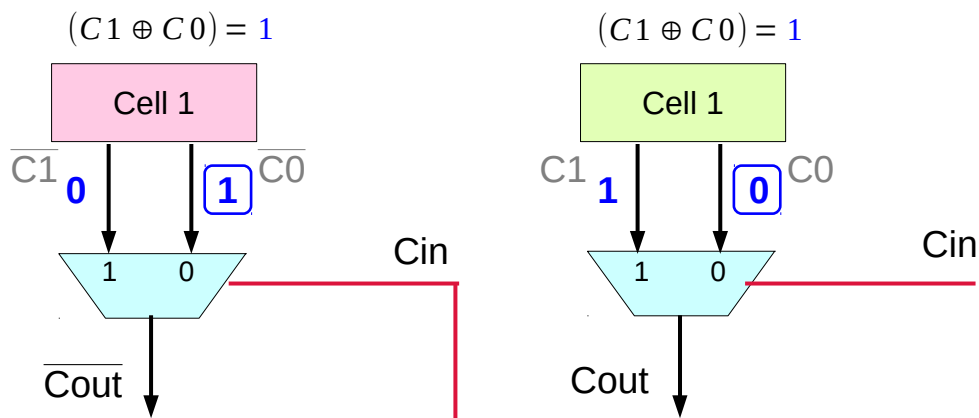
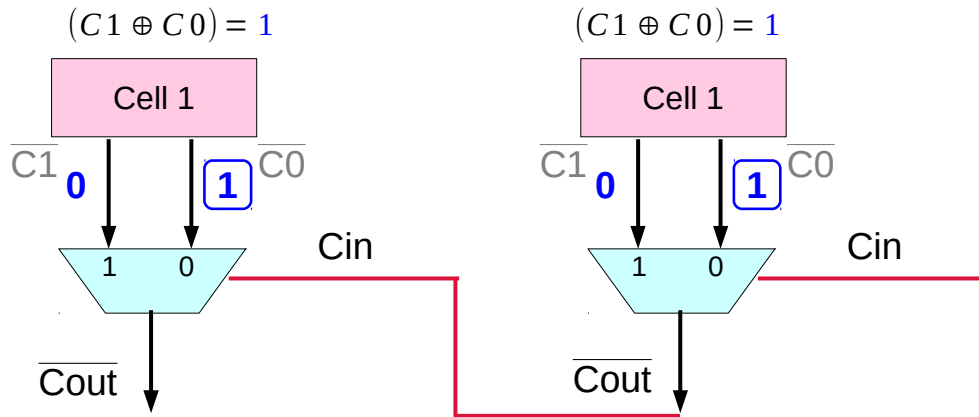


C1	C0	Cout
0	0	0
0	1	\overline{Cin}
1	0	Cin
1	1	1

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

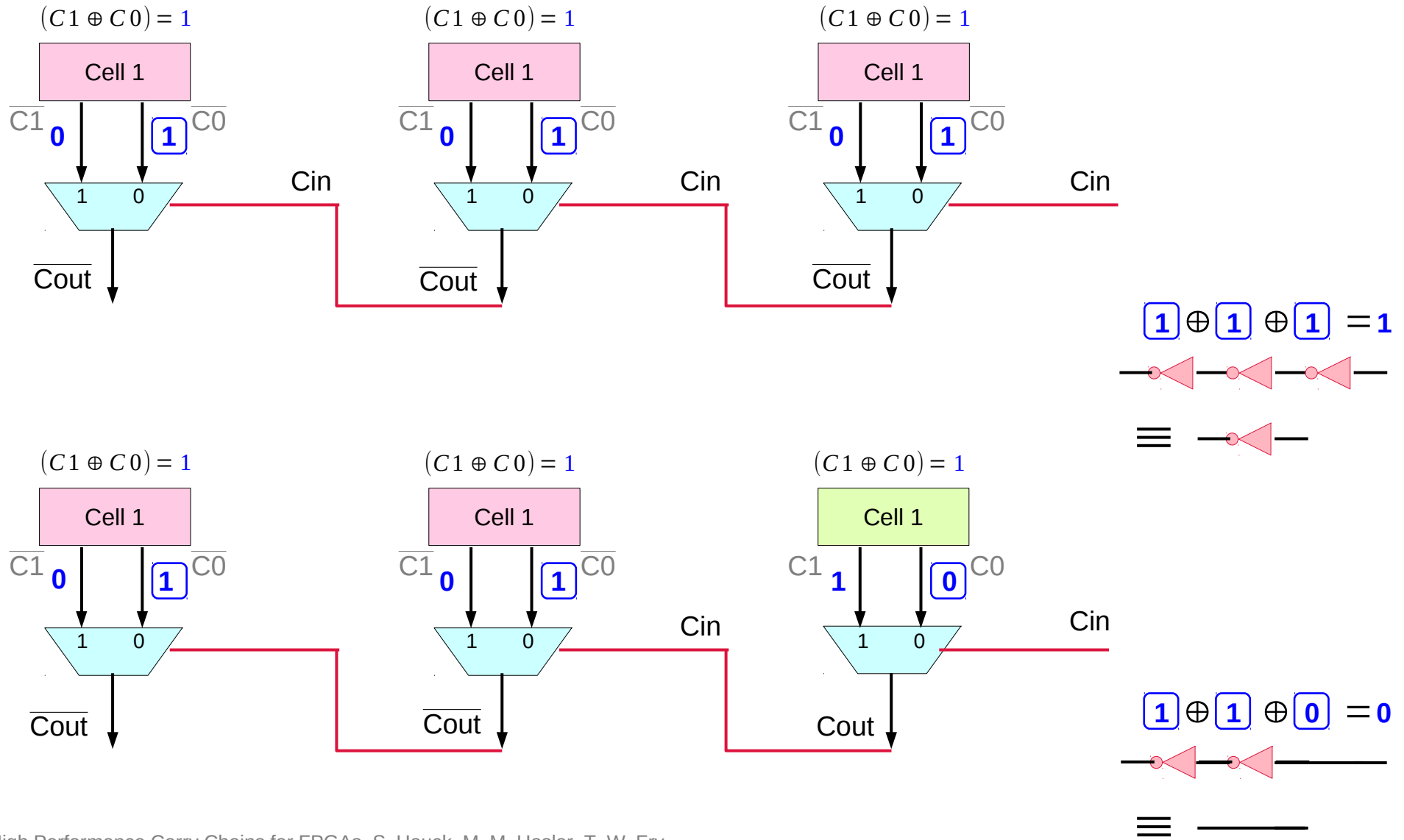


Cout using C1, C0, Cin



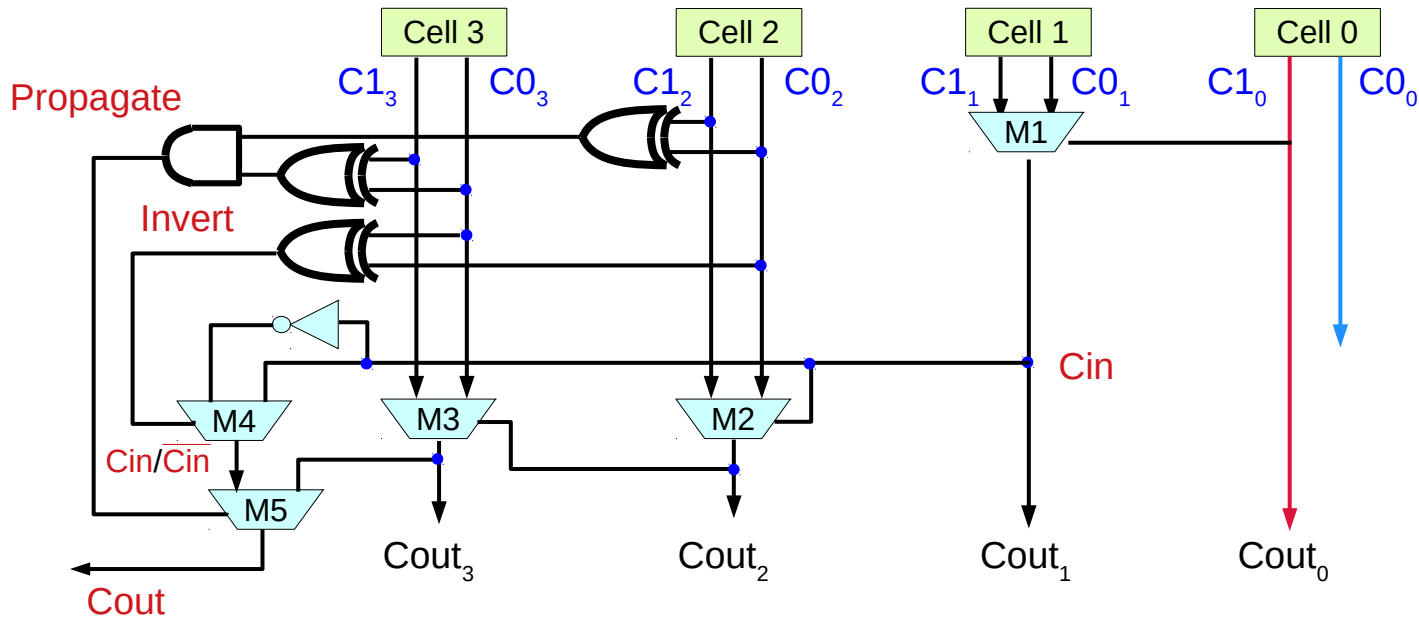
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Cout using C1, C0, Cin



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

Variable Block Carry Structure



$$C1 = X+Y$$

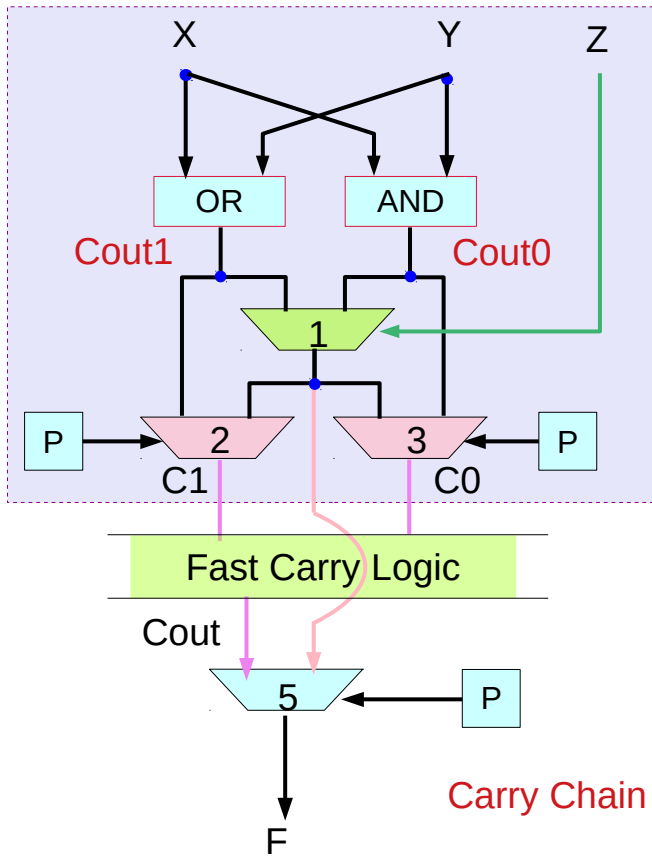
$$C0 = X \cdot Y$$

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{Cin} signal should be sent directly to $Cout$,
- M4 decides whether to invert the Cin signal or not

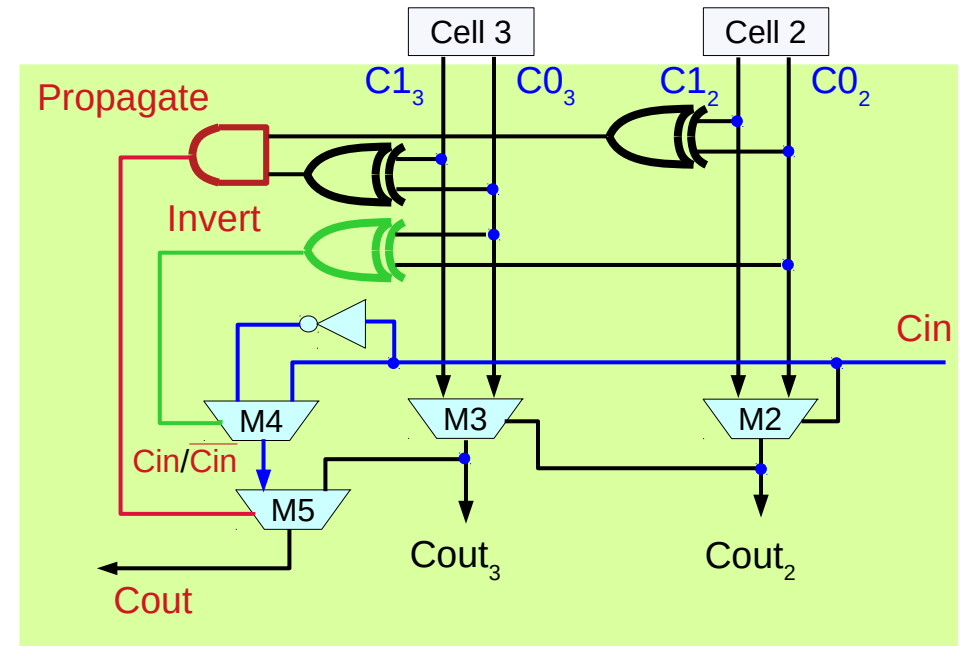
High Performance Carry Chains for FPGAs, S. Hauck, M. M. Hosler, T. W. Fry

Variable Block



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

C1	C0		Name
0	0	0	Kill
0	1	\overline{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

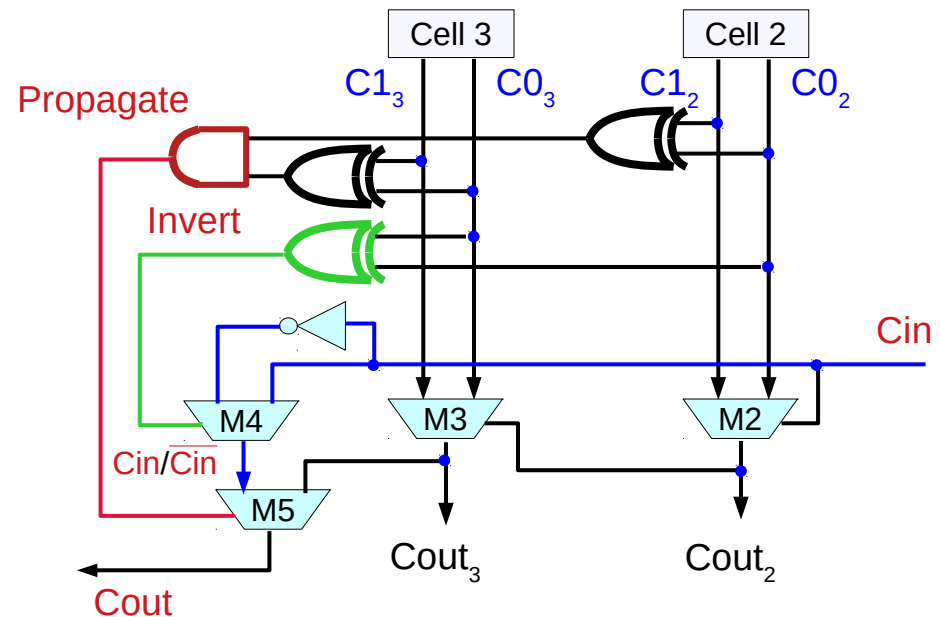
- **normal propagate**
- **inverse propagate**

by **AND**ing together the **XOR** of each stage's **C1** and **C0** signal

If so, we know that the **Cout** function

- **Cin**
- **Cin bar**.

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



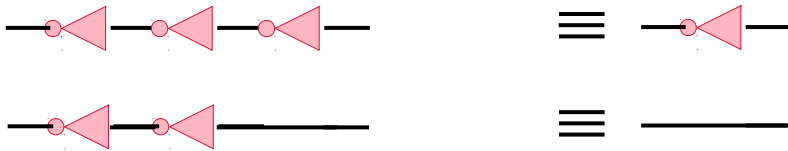
https://en.wikipedia.org/wiki/Carry-lookahead_adder

Variable Block

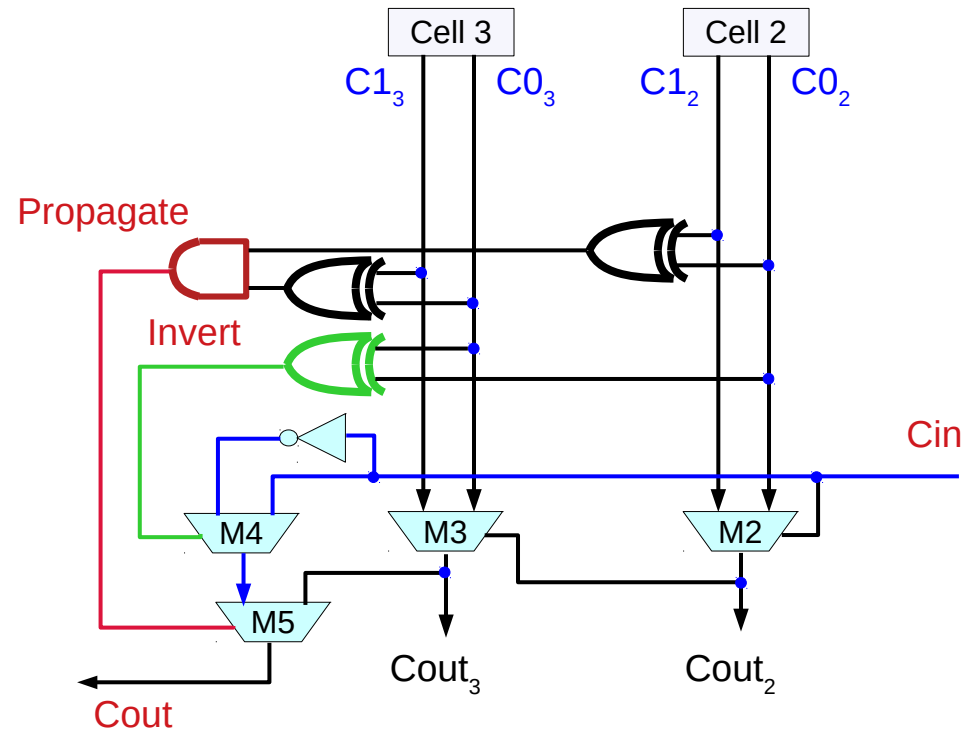
Invert is used only when **Propagate** is true

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



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



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

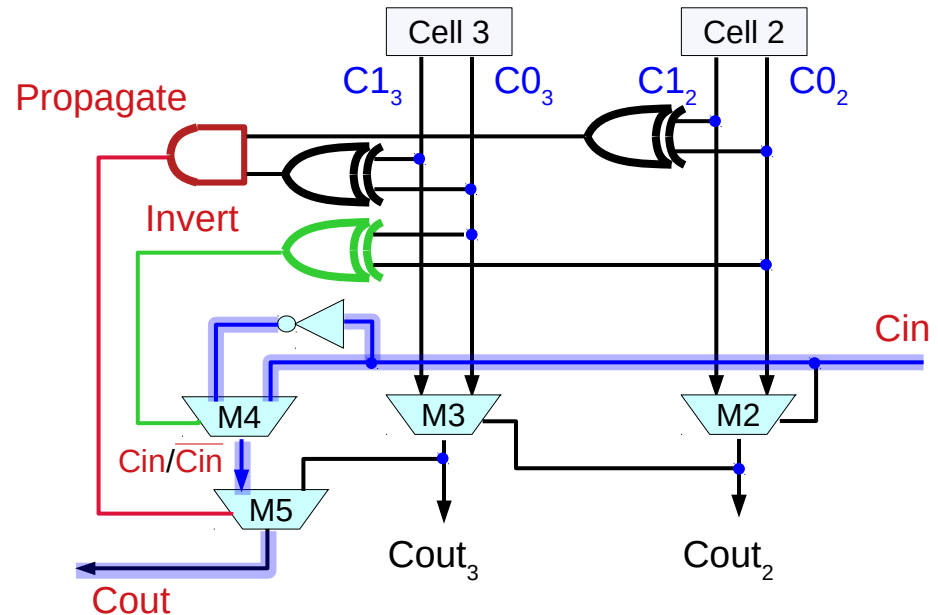
Variable Block

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 $C_{in}/\overline{C_{in}}$

No propagate ripple carry



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

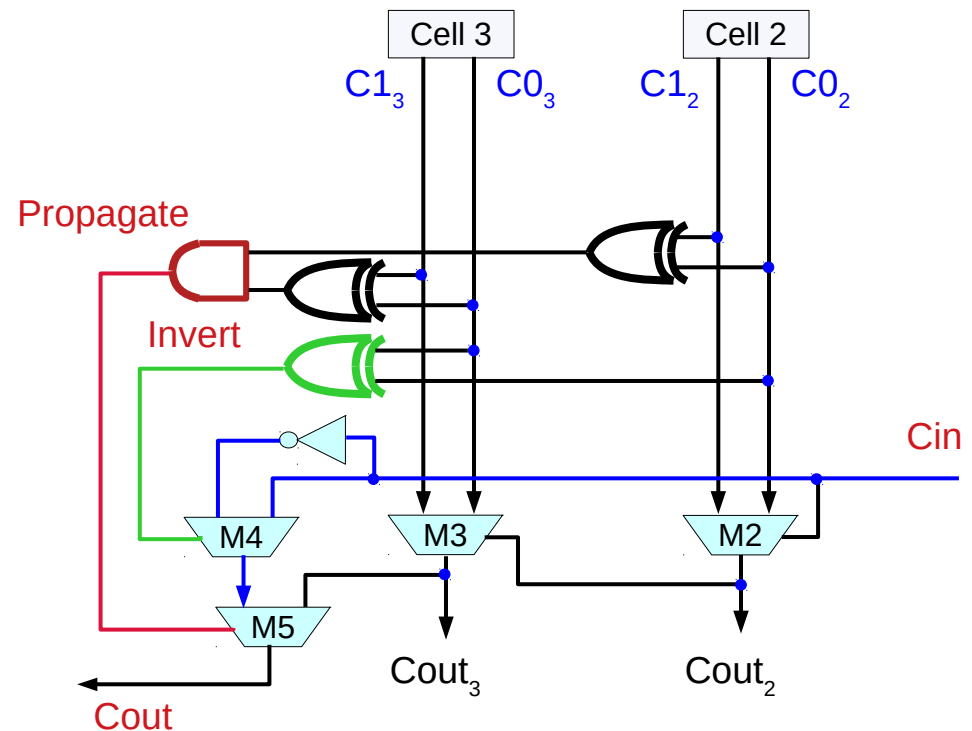
Variable Block

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,

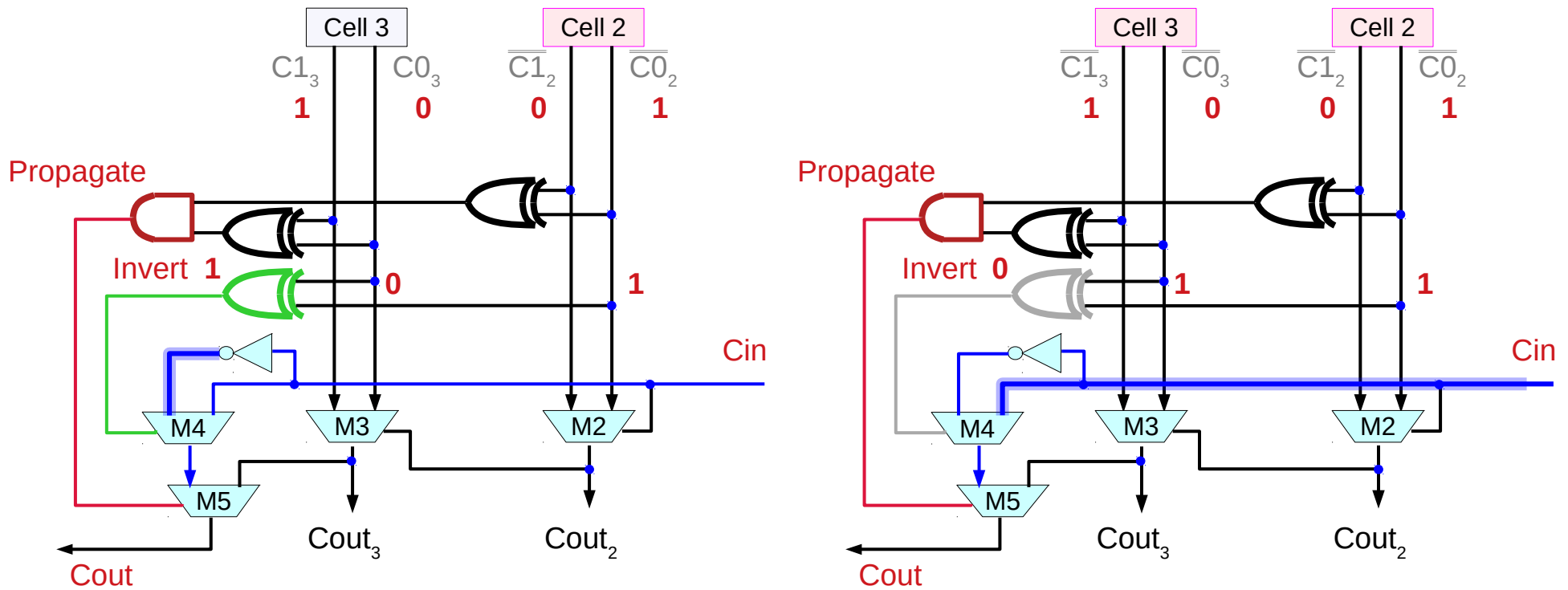
if it is in **generate** mode **inversion signal** will be ignored anyway

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

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
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

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

Variable Block

the organization of the blocks in the **variable block carry** structure bears some similarity to the **carry select** structure

the early stages of the structure grow in length, with short blocks for the low order bits, building in length further in the chain in order to equalize the arrival time of the carry from the block with that of the previous block

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

Variable Block

however, unlike the carry select structure, the **variable block** adder must also worry about the delay from the **Cin** input through the block's **ripple chain**

Thus, after the carry chain passes the midpoint of the logic, the blocks begin decreasing in length.

This balances the path delays in the system and improves performance

The division of the overall structure into blocks depends on the details of the logic structure and the length of the entire computation

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

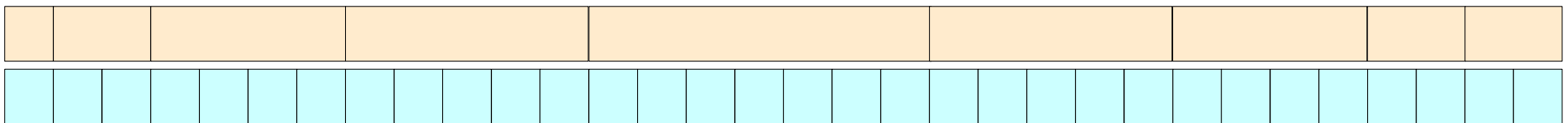
Variable Block

We use a block length from low order to high order cells of 2, 2, 4, 5, 7, 5, 4, 2, 1 for a normal 32 bit structure

$$8+17+7$$

The first and last block in each adder is a simple ripple carry chain, while all other blocks use the variable block structure.

Delay values of the variable block carry chain relative to other carry chains



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

Variable Block

We use a block length from low order to high order cells of 2, 2, 4, 5, 7, 5, 4, 2, 1 for a normal 32 bit structure

The first and last block in each adder is a simple ripple carry chain, while all other blocks use the variable block structure.

Delay values of the variable block carry chain relative to other carry chains

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

Propagate

Propagate

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

P **P**

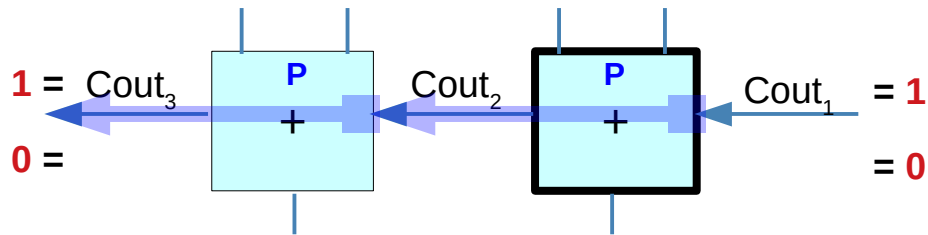
$$C1 = X+Y$$

$$C0 = X \cdot Y$$

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

P **G**

G



P	G'	= P
G	G'	= 0
P'G'	G	= 0

Invert

Invert

$$(C1_3 \oplus C1_2)$$

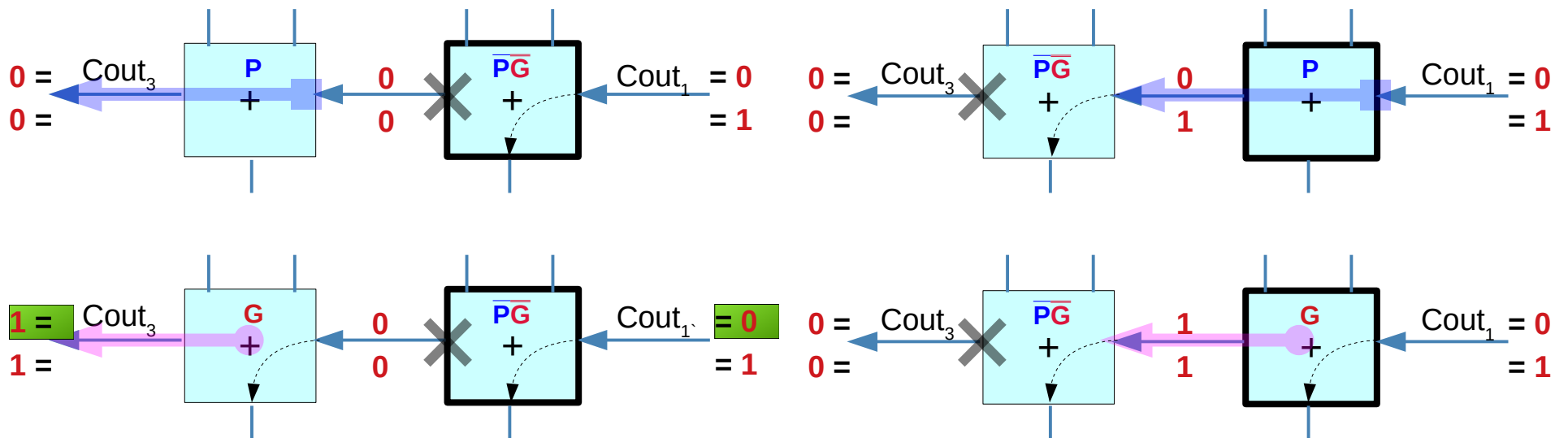
$$\begin{array}{cc} P & P \\ G & G \end{array}$$

$$\begin{array}{l} C1 = X+Y \\ C0 = X \cdot Y \end{array}$$

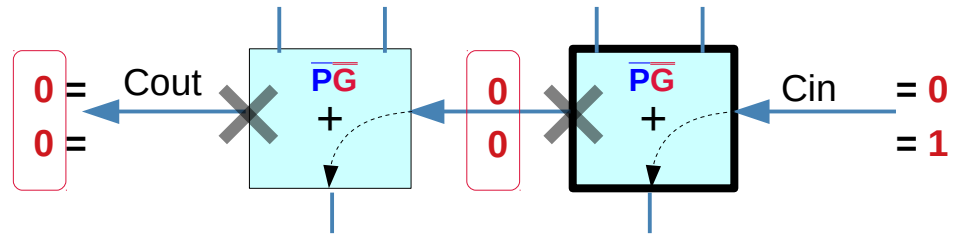
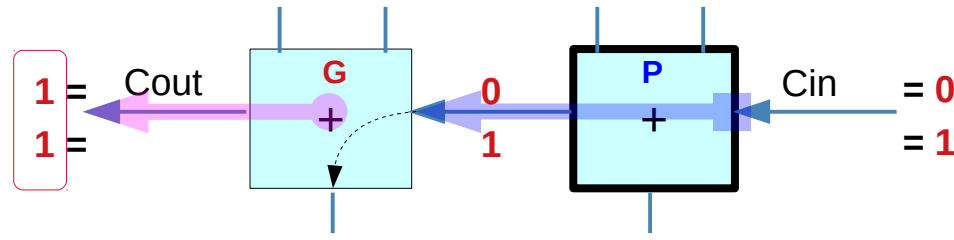
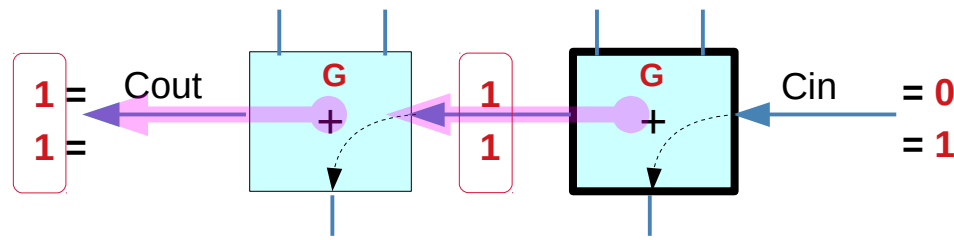
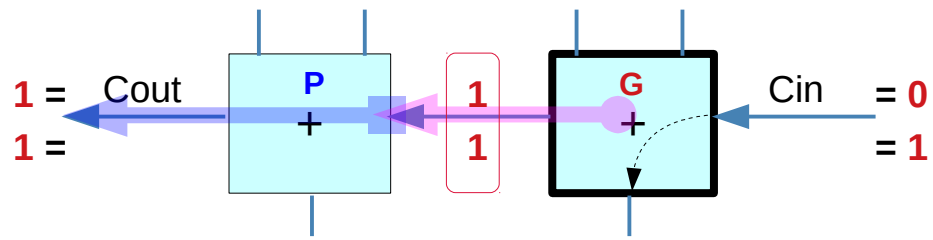
$$C1 \oplus C0 = X \oplus Y = P$$

$$\begin{array}{cc} P & G \\ G & \end{array}$$

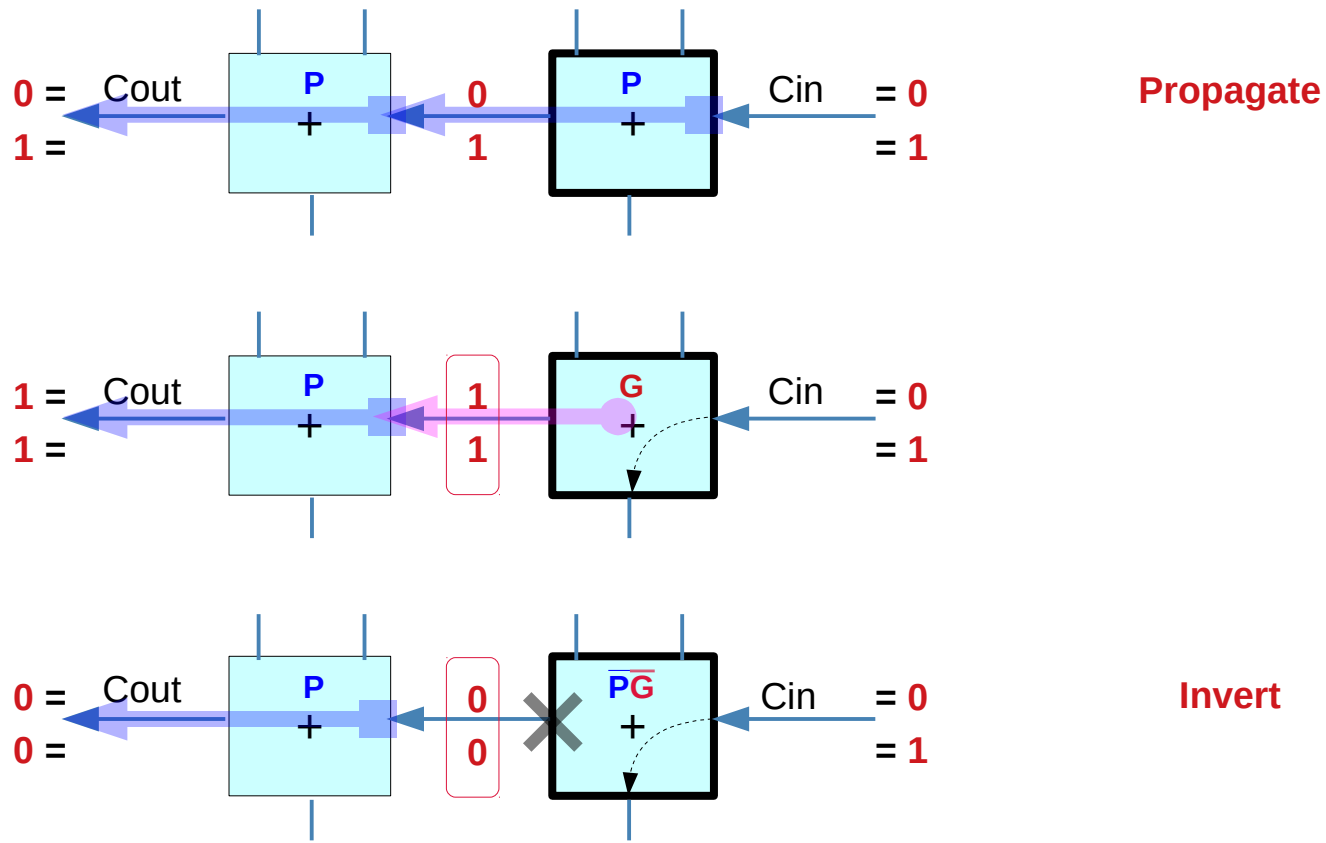
$$\begin{array}{cc} P & P'G' \\ G & P'G' \\ \hline P'G' & P \\ P'G' & G \end{array}$$



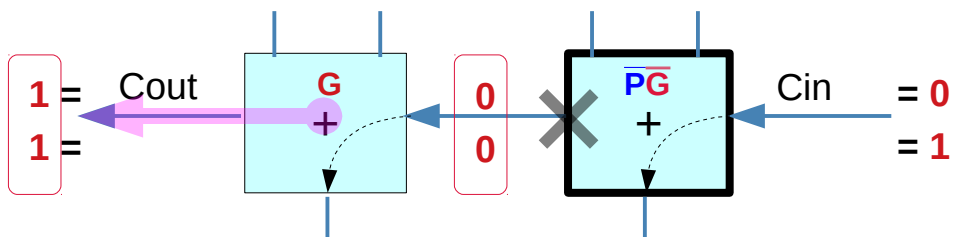
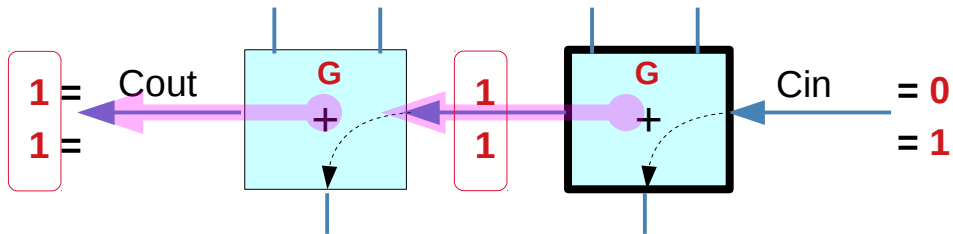
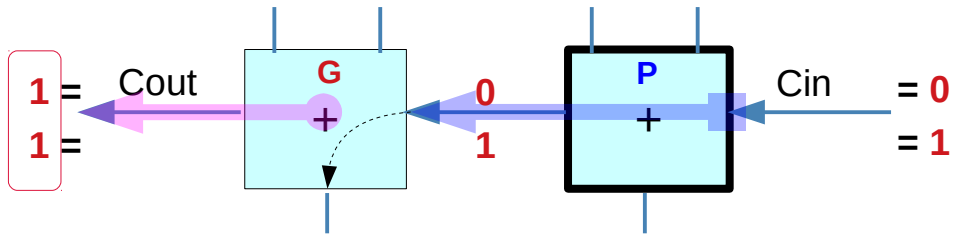
Variable Block



Variable Block

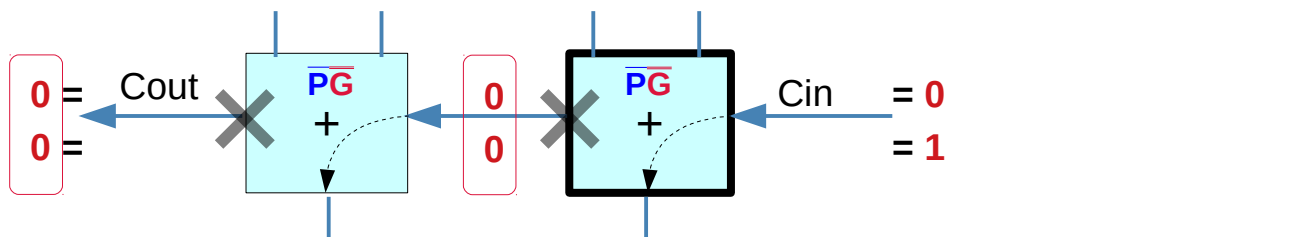
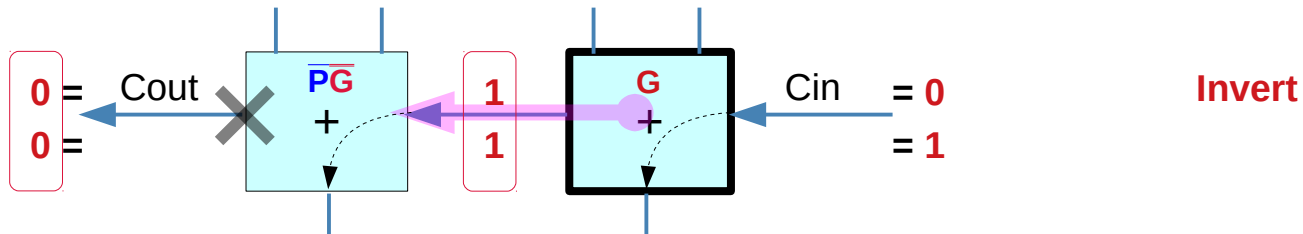
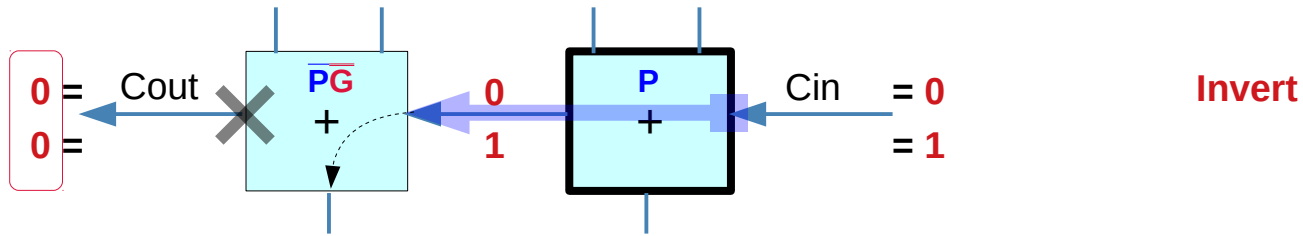


Variable Block

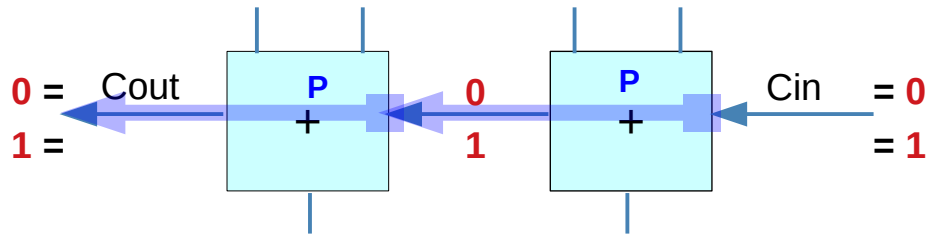


Invert

Variable Block



Variable Block



Propagate

