
Lecture 14:

Datapath Functional Units Adders

Mark Horowitz

Computer Systems Laboratory

Stanford University

horowitz@stanford.edu

Overview

Reading

W&E 8.2.1 - Adders

References

Hennessy and Patterson “Computer Architecture a Quantitative Approach”
Appendix A (written by David Goldberg)

Introduction

Somewhat surprisingly datapaths often contain many simple cells – latches to stage data (especially in a pipelined machine) and tristate drivers to drive the bus lines. But these cells are pretty simple. Since these units implement the dataflow portion of an algorithm, they tend to operate on numbers. This lecture will explore some methods of building datapath functional units that add. After talking about adders, we turn to two related topics, ALUs and counters

Numbers

Numbers are represented by n-bit binary strings.

- n is the datapath width

Numbers are represented in two forms:

- Fixed Point (Integer)

There are two forms here, signed and unsigned

Unsigned - Numbers range from 0 to $2^n - 1$

Signed - Numbers are in two's complement form

Range from -2^{n-1} to $2^{n-1} - 1$

-1 is 11...11 (it is 0 -1)

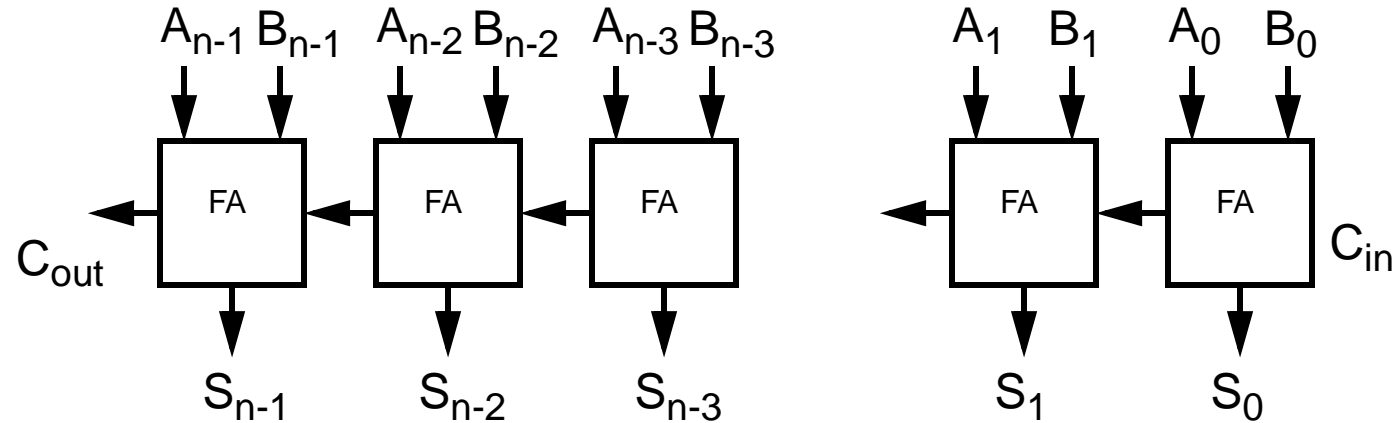
- Floating Point

Even more complicated, contains an exponent field and a mantissa, which give it even more dynamic range.

We will stick to integers

Addition

The MSB output can depend on the LSB bits because of the carry:



A full adder (FA) has three inputs and two outputs

$$\text{Sum} = A \text{ XOR } B \text{ XOR } C_{in} = \bar{A} \bar{B} C_{in} + \bar{A} B \bar{C}_{in} + A \bar{B} \bar{C}_{in} + A B C_{in}$$

$$C_{Out} = AB + BC_{in} + C_{in}A$$

In this organization the critical path is from C_{in} to C_{out}

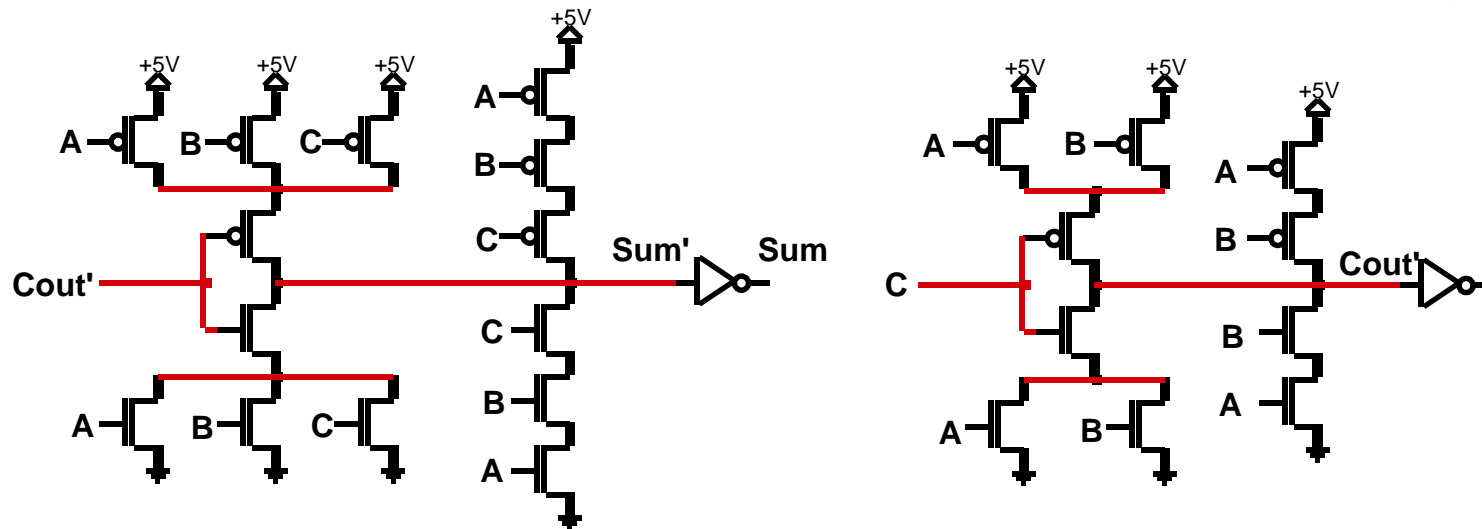
Full Adder Implementation

Use two complex gates, and two inverters:

$$C_{out} = A B + C (A + B)$$

$$Sum = \overline{C_{out}} (A + B + C) + A B C$$

Notice that the implementation of the two gates is not standard -- the pMOS devices are not the dual of the nMOS, they have the same topology. This is VERY unusual. The reason for this is that XOR and Majority functions are self duals -- complementing the inputs and the output leaves the function unchanged, $XNOR(x_b, y_b) = XOR(x, y)$. This is not in general true. The designer of this gate took advantage of this fact to reduce the maximum number of series devices in each gate.



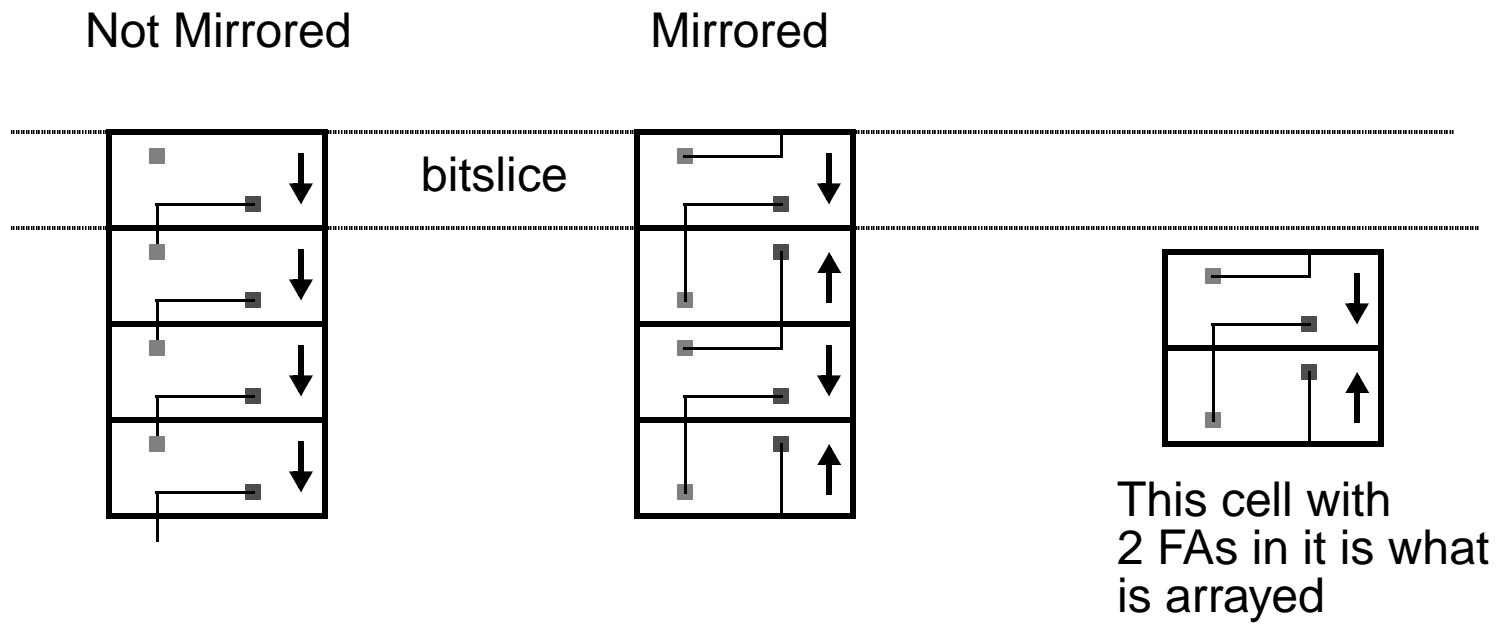
Critical path goes through two gates per stage.

Datapath Layout of Adder

Base cell is a full adder

Layout complexity depends if the bitcells are mirrored

- They usually are to save area, which makes the carry chain hard



- Need to plan if the cells will be mirrored

Faster Ripple Adder

The critical path through each bit goes through two 'gates'

- Complex gate to generate $\overline{C_{out}}$, and then an Inv to generate C_{out}

Inverter serves to functions:

- Complements the output (generates C_{out} in its true form)
- Isolates the load capacitance of the next bit from the output of the gate with series devices. (Remember this inverter should be sized so the complex gate and the inverter has similar delays)

Could let each stage invert, since Sum and Carry functions are self duals:

$$\text{Sum} = A \text{ XOR } B \text{ XOR } C_{in} = \sim(\overline{A} \text{ XOR } \overline{B} \text{ XOR } \overline{C}_{in})$$

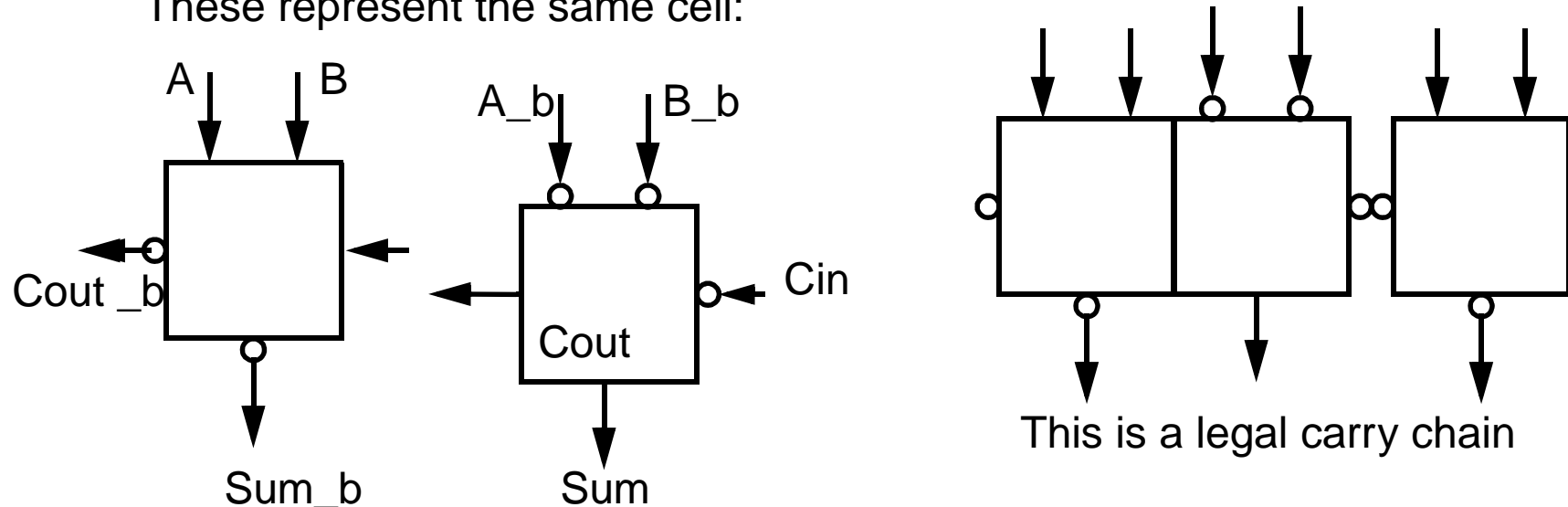
$$C_{Out} = AB + BC_{in} + C_{in}A = \sim(\overline{A} \overline{B} + \overline{B} \overline{C}_{in} + \overline{C}_{in} \overline{A})$$

So if I use the same hardware I get the true values out if I input the complements, and get the complements if I input the true inputs.

Inverting Full Adder Ripple

Assume that you removed the inverters in the previous full adder

These represent the same cell:



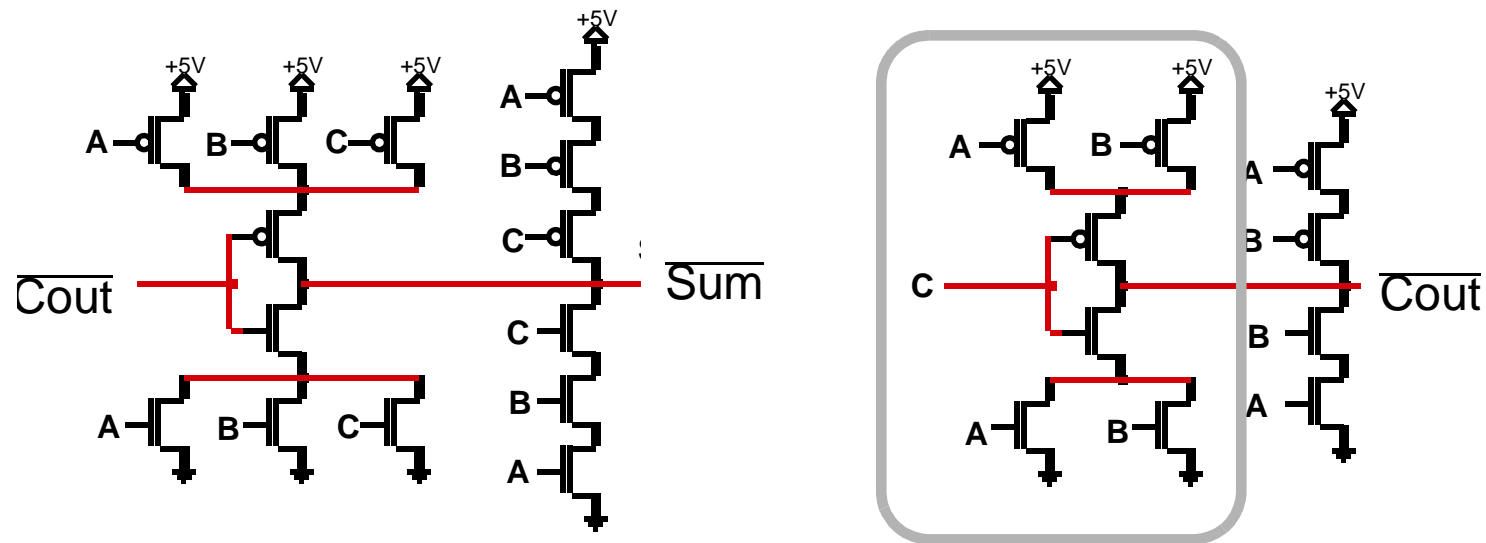
CAUTION:

For this to be faster than the version with the inverter, the dominant load on $\overline{\text{Cout}}$, must be just from the two transistors in the following Cout gate. Leads to interesting transistor sizing.

Adder Sizing

The transistors in the grey oval are on the critical path. The rest are not

- To reduce the loading on the critical signal, the other transistors should be small (at least much smaller than the transistors on the critical path)
- Since C will be late, the transistors connected to A,B should be large



- The adder layout in Weste does this.

Faster Adders

We have looked at reducing the number of gates in the critical path of a ripple adder, and to try to make those gates fast. But there are much better methods of speeding up adders. These take a more global view of the problem, and try to reduce the length of carry chain by doing things in parallel.

We will briefly look a number techniques for speeding up adders. The first is still a ripple adder, but the 'ripple' is done with pass transistors. Then we will look at some techniques of using parallelism to reduce the adder delay.

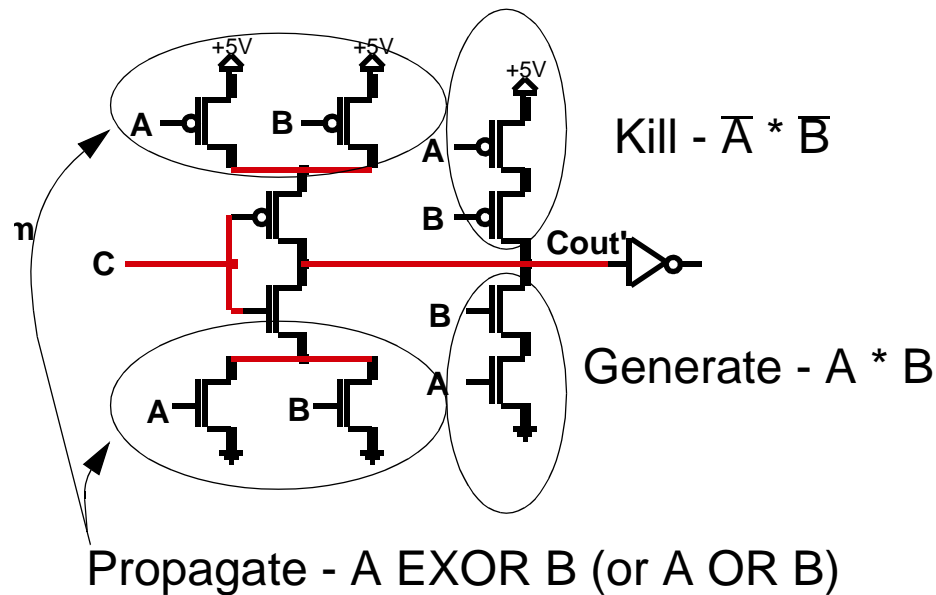
These techniques include:

- Switch logic carry chains
- Carry bypass (carry skip)
- Carry look-ahead
- Carry select (conditional sum)

Carry Chain

This is the critical path of the adder so we will focus our attention here.

Look at carry gate:

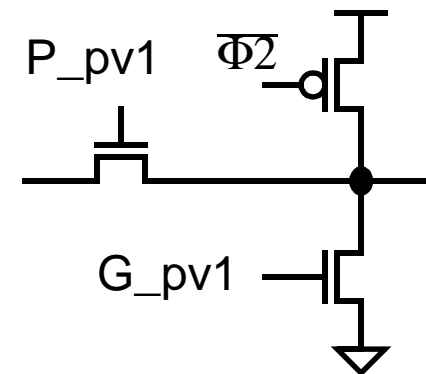
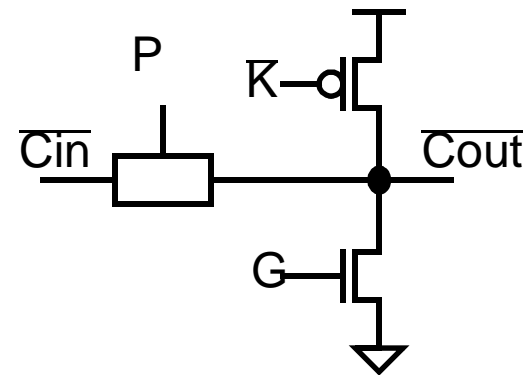


Switch-Logic Carry Chains

Switch-Logic:

- Implement propagate with pass gate
- Implement kill with a pull down transistor
- Implement generate with a pull up transistor

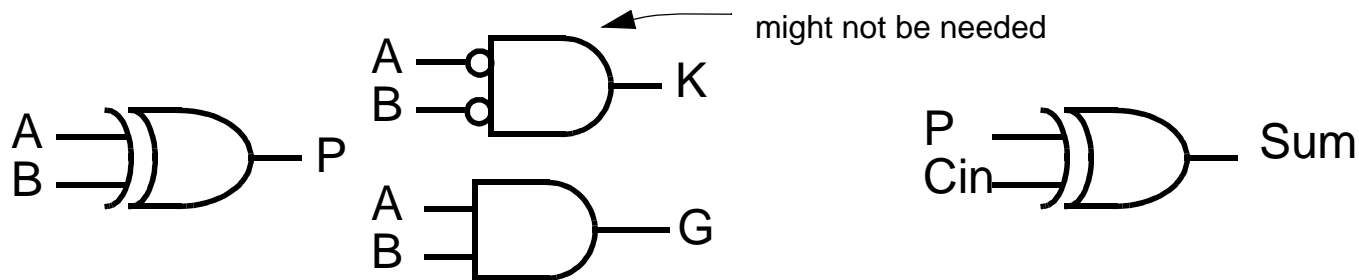
To reduce the logic needed, and the capacitance on the carry chain use precharge switch logic. Precharge the output high, and pull it low if needed. The inputs to the gate can be outputs from other domino gates (Carry is a monotonic function of P , C , K^1)



1. Need to be careful, since we will use an inverter to buffer the output before we use it. That is the reason that the switch logic is generating C_b and not C . Switching G and K will generate C directly.

Adders Using Carry Chains

The carry chain is only part of the adder. You need to generate the P, G signals that the adder needs and to generate the sum at the end. In addition to the carry chain, each bit cell needs the following gates:

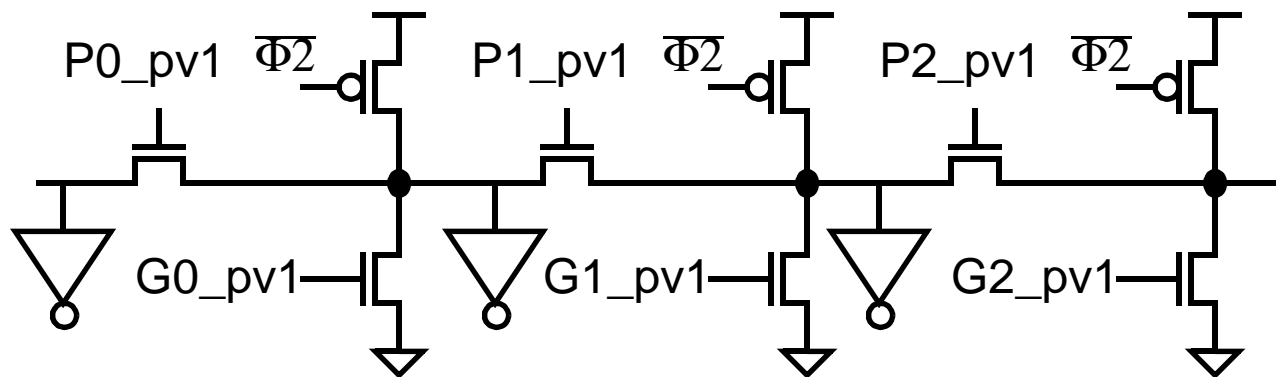


- The gates that generate P, G, K can be precharge gates, since the inputs are usually stable signals. This means that P, G, K can be domino $_v$ signals, and can drive the domino carry chain
- The final EXOR must be a static gate since it is not a monotonic function of its inputs, and its inputs will be $_v$ signals.

Timing of Manchester Carry Chains

The good news is there is not a gate between stages.

The bad news is that the number of series transistors increases with the number of stages, so the delay will grow like n^2

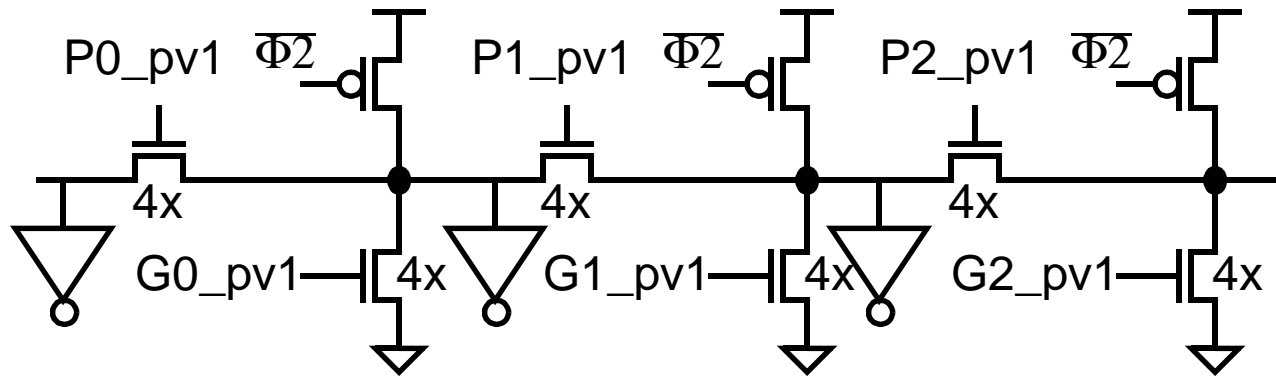


- Capacitance per stage (assuming all 4:2 devices, no diff sharing)
 $3 n_{diff} + p_{diff} + C_g + inv + \text{bit-width of wire} = 12fF + 4fF + 4fF + 8fF + 8fF (30\mu)$
 $= 36fF$
- Resistance per-stage is 6.5K, so the delay is approximately $.12ns * n^2$, $(RCn^2/2)$ where n is the number of stages directly tied together.

Sizing Manchester Carry Chains

Critical path is through the pass chain. Try to reduce this delay:

- Make P and G transistors 4x larger, and share diffusion¹



- Capacitance per stage:

$$2ndiff (16\lambda) + pdiff + Cg + inv + \text{bit-width of wire} = 32fF + 4fF + 16fF + 8fF + 8fF \\ (30\mu) = 68fF$$

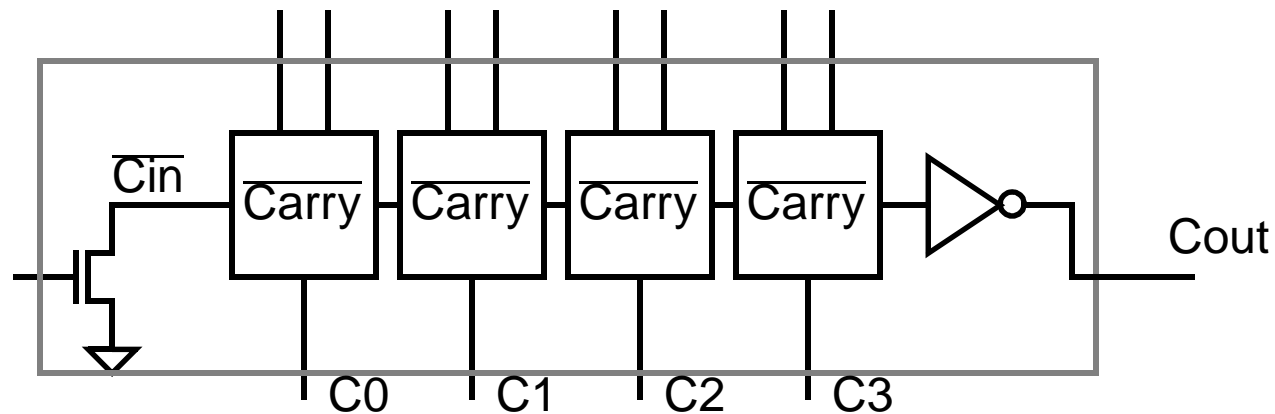
- Resistance per-stage is 1.6K, delay is $0.054ns * n^2$.

1. Make G larger since it does not hurt (diffusion is shared, and since it will be important in faster adders)

Manchester Carry Chains

To limit the effect of the n^2 term, break carry chain into sections.

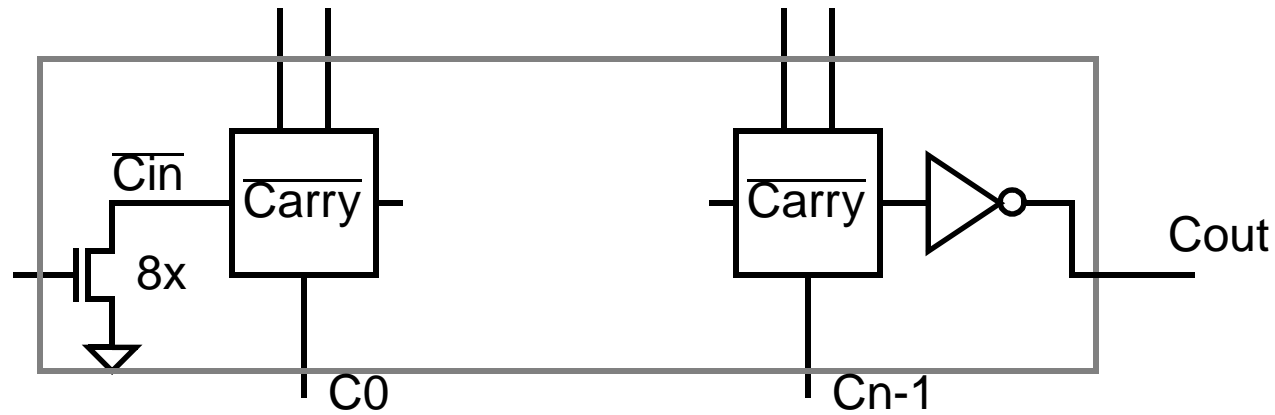
- Each section is about 4 stages long (3 stages might be better)
- Between sections the carry is buffered.



- The buffering makes the delay linear with the number of bits
- But the carry stills needs to flow through all the carry chains.

Timing of Buffered Carry Chains

What is the 'right' number of stages?



Assume first transistor is $8x$ min, and final inverter is minimum

Delay is the inverter delay (C_{out} rising) plus the delay of the chain including the resistance of the initial $8x$ transistor.

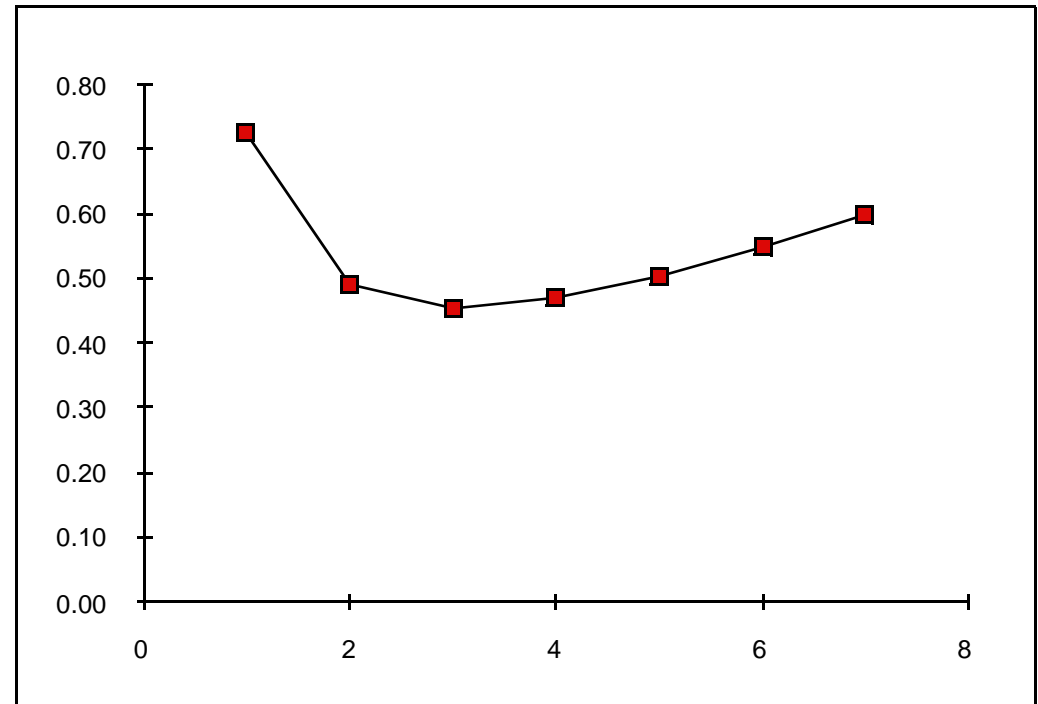
$$\text{Inverter delay} = 13K_{pMOS} * (8fF_{diff} + 32fF_{gate}) = 0.5ns$$

$$\text{Chain} = 0.8K * (68fF * n) + 0.054 * n^2 ns = 0.054n(n+1)ns^1$$

1. I have taken some short cuts here, but suggest you go through the long way if your are confused

Timing of Carry Chains

Stages	Total Delay	Delay per bit
1	0.61	0.61
2	0.82	0.42
3	1.15	0.38
4	1.58	0.39
5	2.12	0.42
6	2.77	0.46

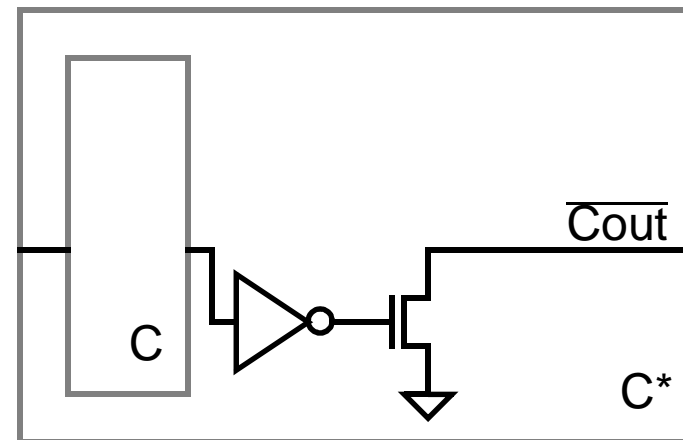
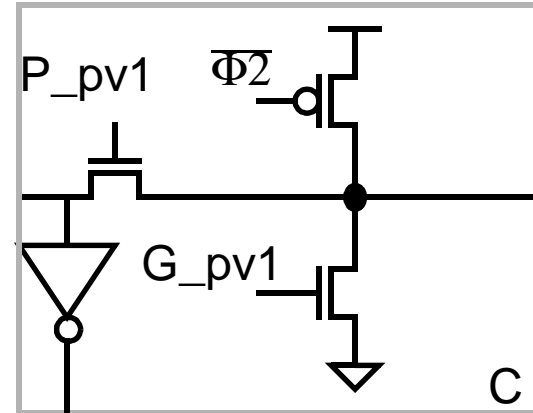


So for these sizing, the optimal number in a stage is around 4, and the average delay per bit is around 0.4 ns. This is not optimally sized (pMOS in final inverter should be larger) but it is probably close.

Layout of Carry Chain

Layout of a Manchester adder is not too bad, even with groups:

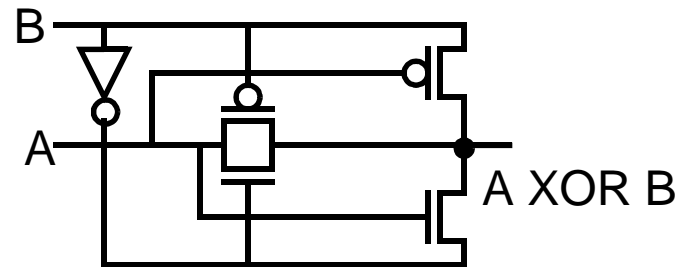
P,G gen ↑	C ↑	XOR ↑
P,G gen ↓	C ↑	XOR ↓
P,G gen ↑	C ↑	XOR ↑
P,G gen ↓	C* ↑	XOR ↓
P,G gen ↑	C ↑	XOR ↑
P,G gen ↓	C ↑	XOR ↓
P,G gen ↑	C ↑	XOR ↑
P,G gen ↓	C* ↑	XOR ↓



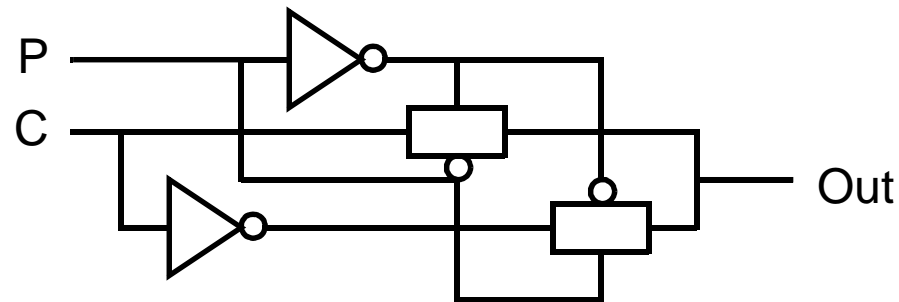
Final XOR

The final XOR of the adder needs to be a static gate.

- While this XOR works in silicon it gives IRSIM problems, so we won't use it:



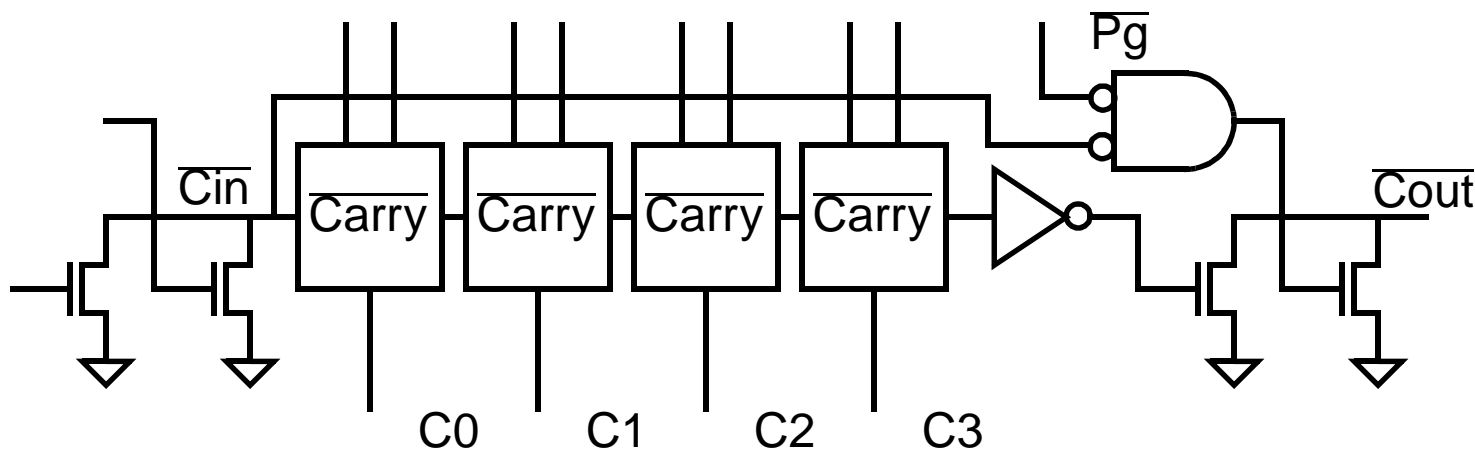
- This other version is a little safer



Carry Bypass Adders

Since we have divided the bits in the word into a number of groups.

- For each group check to see if all the P are true
- If so, then bypass the C_{in} to C_{out} of that group
- Otherwise, do the normal thing.



Why Carry Bypass is Faster

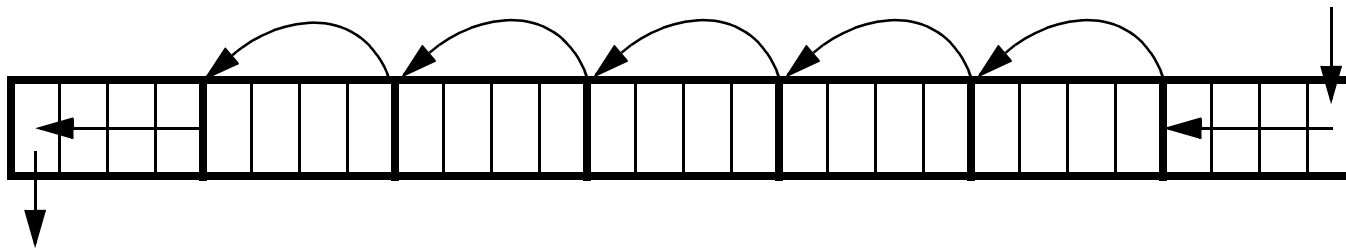
All groups can calculate P_g at the same time (in parallel)

Worse-case is when carry needs to propagate through all bits

- Since we precomputed P_g , that path is now much shorter

Hop around groups, rather than through them

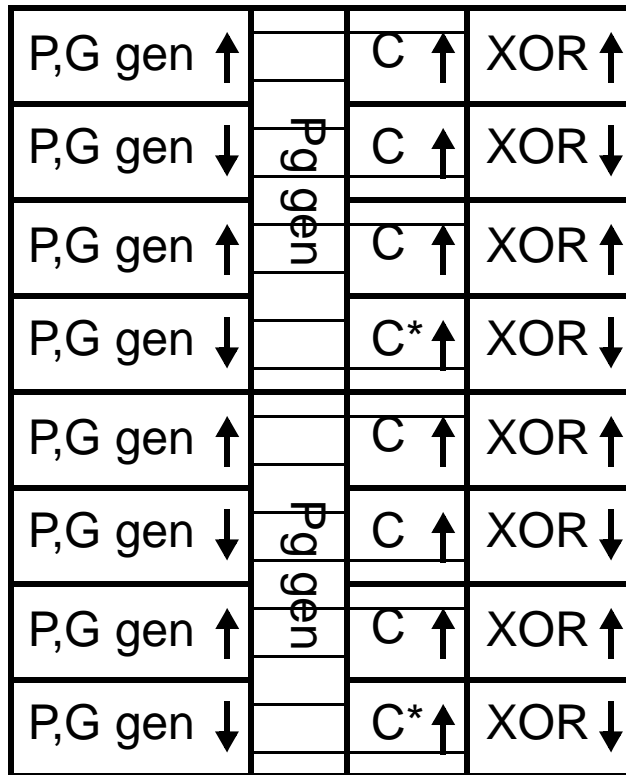
- Critical path is now through one local carry chain, then through a number of bypass gates, then back into a final local carry chain.



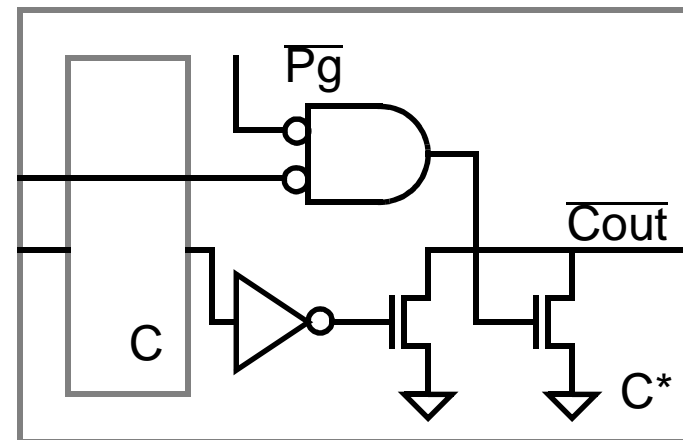
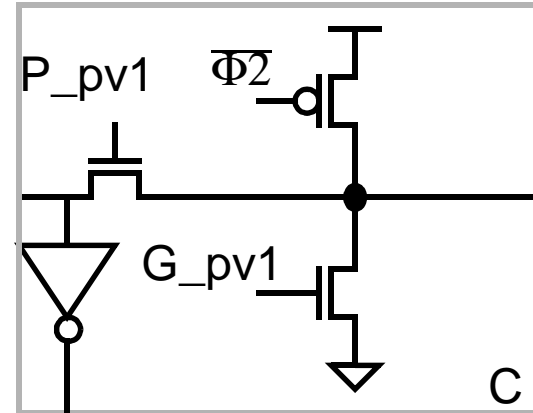
- This improvement did not cost much hardware.

Layout of Carry Chain

Layout of a bypass adder is almost the same, C^* gets a more stuff:



Also have a few more wires to route. You need to generate Pg (a 4 input NAND gate in the PG gen section, and you need to route C_{in_b} to C^*

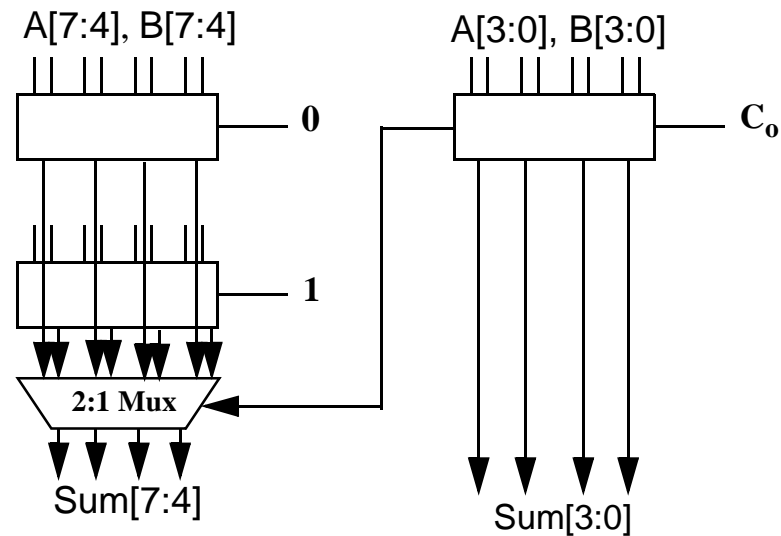


Building Faster Adders

By using more parallelism, one can build even faster adders

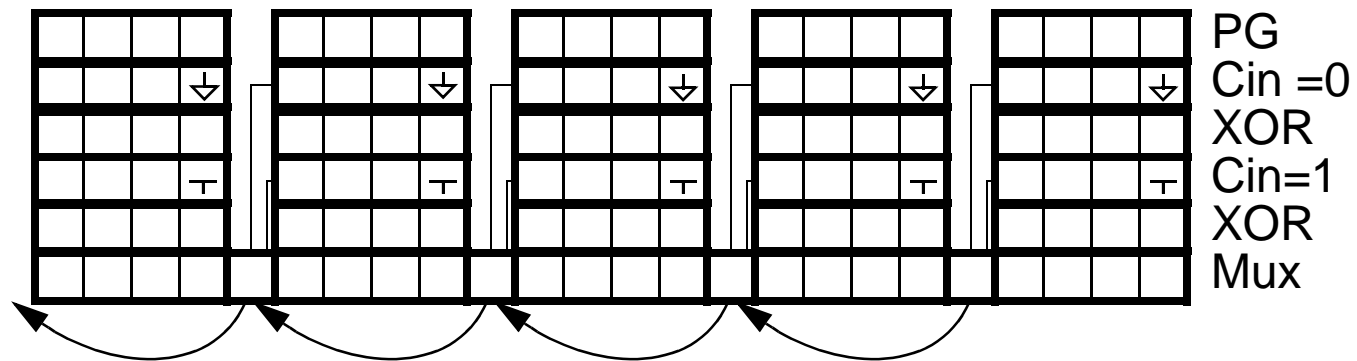
While waiting for the carry input, why not calculate both possible answers (answer if C_{in} is 0 and answer if C_{in} is 1)

When C_{in} is known, it is only a Mux delay to get C_{out} and all the Sums for the group.



Carry Select Adder

A larger adder would look something like this:



Notice that the PG logic can be shared with both carry chains

Critical path is first carry chain and then n mux delay

What is the optimal block size for a carry select adder? (Hint they are not all the same)

+ Even Faster Adders

These adders do more of the calculation in parallel, by bypassing the bypass. This leads to a tree like structure, so these adders are often called tree adders. For these adders the add time grows $O(\ln n)$ where n is the number of bits.

These adders often build trees to combine both P and G over larger and larger groups. The reason that this works, is that both functions can be computed hierarchically.

- Since P is just the AND function, it can be computed in any order

$$\begin{aligned} P_{15-0} &= P_{15} P_{14} P_{13} P_{12} \dots P_0 \\ &= P_{15-12} P_{11-8} P_{7-4} P_{3-0} \end{aligned}$$

- Generate for a group (G_g)

$$\begin{aligned} G_{15-0} &= G_{15} + P_{15} G_{14} + P_{15} P_{14} G_{13} + P_{15} P_{14} P_{13} G_{12} \dots \\ &= G_{15-12} + P_{15-12} G_{11-8} + P_{15-12} P_{11-8} G_{7-4} + P_{15-12} P_{11-8} P_{7-4} G_{4-0} \end{aligned}$$

+ Tree Adders

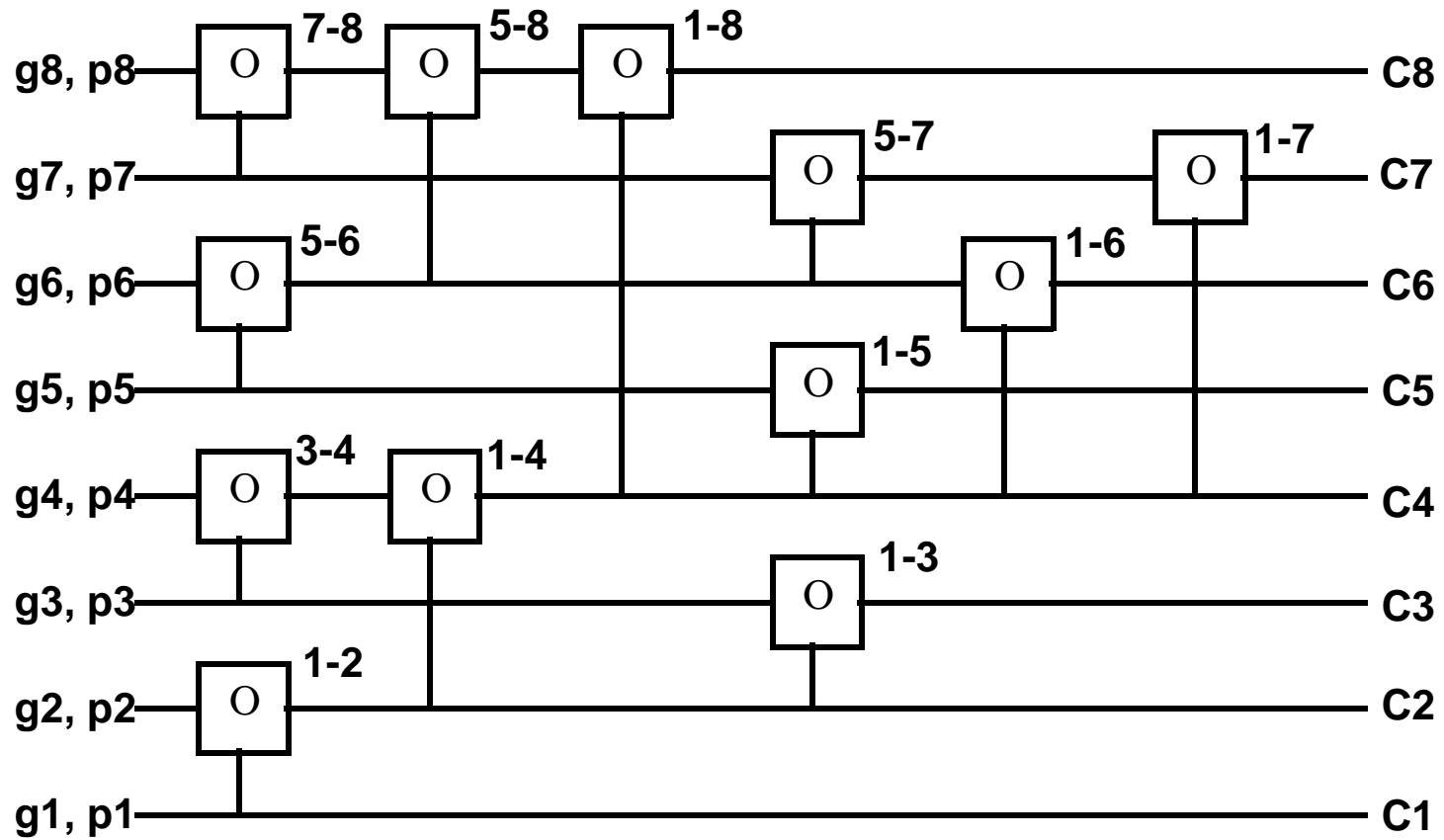
Since we can compute P and G in any order, we can compute them mostly in parallel by using a tree structure.

- First compute all the two bit groups (for a binary tree), then use these outputs to compute all 4 bit groups, then 8 bit groups, etc
- At each stage you do the same function:
 - $P_g = P_{\text{left}} * P_{\text{right}}$; $G_g = G_{\text{left}} + P_{\text{left}} * G_{\text{right}}$
(less significant bits are on the right)
- Initially, the inputs are P,G from the bits

Then the inputs are the outputs from the previous level in the tree

Once you go up the tree, you need to go back down the tree to generate the outputs for all the bits.

+ Tree Adder



Adder Design

There are many adder designs

- Simple ones have $(2)N$ gates in critical path
- Can remove gates by using switches, but still have linear delay
- More complex ones add in $\ln(N)$ gates

These adders have larger wire loads, and higher fanouts, but still can be made very fast (2-3ns 64 bits)

Book and references have more complete description of adders

Added complexity (and area) not worth it for small adders!

- Even though adders are not completely regular, they work nicely in a datapath layout

+ Aside - ALU Design

Once you have an adder, making an ALU is very simple

Two approaches:

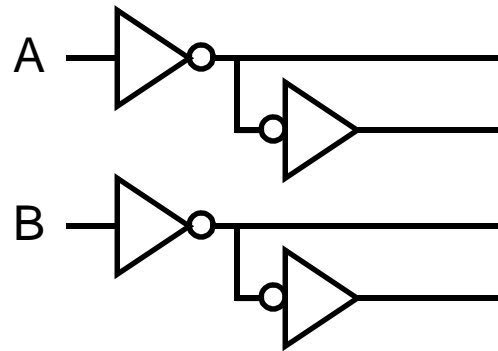
Build a separate logic unit and mux together the outputs. This is probably the fastest solution, since you don't slow down the add critical path, but it will take more area.

Merge the two designs together by changing the definition of P and G. Since the output (Sum) is $P \text{ XOR } C_{in}$, if $G = 0$, and $C_{in}(\text{to adder}) = 0$ then Sum will equal P. Can do logical operations by using a general function box for the P function.

The first is probably the preferable solution, but I will show the second, because it is a little more clever (and the programmable P function unit is a perfect LU for the first solution)

+ Input Buffer

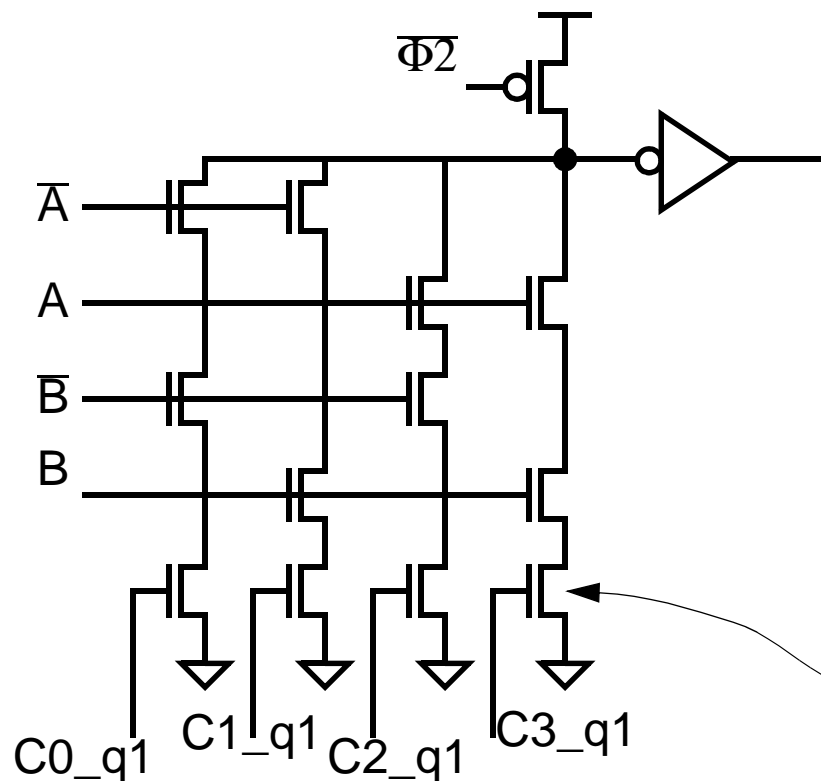
- If the input buses are `_s1`



- Buffer the input to reduce loading on the bus. If this is not needed, then one of the inverters can be removed
- If the buses are not stable, then a pass transistor can be added to the inputs of the first inverter to make them latches

+ P Function Block

The block that generates the signal called P must be able to generate any Boolean function of two variable. This is easy -- just use a mux. To reduce control lines, I will use a precharge mux.



By setting the right values on the control lines, this block can generate any logic function

Exor = 0 1 1 0

And = 0 0 0 1

Or = 0 1 1 1

These transistors can be shared for all the bit slices.

+ G Function Block

This is similar to the P function block, but it does not need to be as complex. If we only wanted to do addition and logic functions, then it would only need to generate the functions (AND, 0). But we want to be able to do subtraction too.

- $A - B = A + \bar{B} + 1$, where \bar{B} is the ones complement of B, which is just the complement of each bit.
- Since after the P, and G function block, no other part of the adder uses A,B, we can get subtract by redefining P and G, and setting Cin to be 1
- If we didn't do this, we would need to add an explicit mux to invert one of the inputs to adder in the case of subtraction.

- For addition:

$$P = A \bar{B} + B \bar{A}; \quad G = A B$$

- For subtraction:

$$P = A B + \bar{B} \bar{A}; \quad G = A \bar{B}$$

+ Rest of ALU

Is basically the same as an adder:

- Need a fast carry chain
- Final static XOR gate
- Latch to hold the value (since the output of the ALU is `_v1`)
- Bus driver to drive the output of the latch on bus when the ALU result is needed

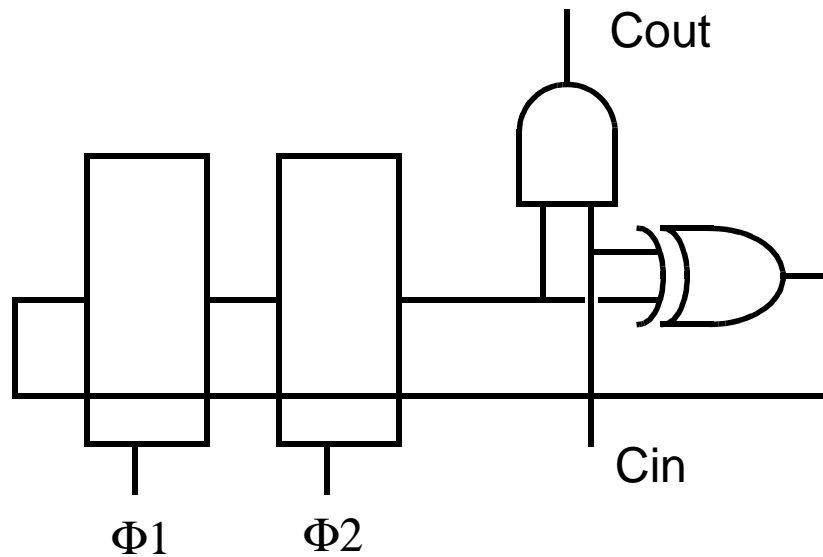
Counters

Often you need to build a counter in your design

- Counter must obey two phase clocking

Ripple counters are out

- A synchronous counter is



- It is just an incrementer and a register

Counters

Incrementers:

- Are just adders, where one input is 0, and $C_{in}=1$
 - $B=0$ implies $P = A$, $G = 0$
 - So P,G logic is simple (does not exist), but still have carry chain

Can (and must) use any of the fast carry techniques to create fast counters

Also need a way to test counters:

- Need a reset, to get counter to known state
- But also probably don't want to wait when it counts to check the carry chain. For large counters you need to add a way to load the counter and read out its state.