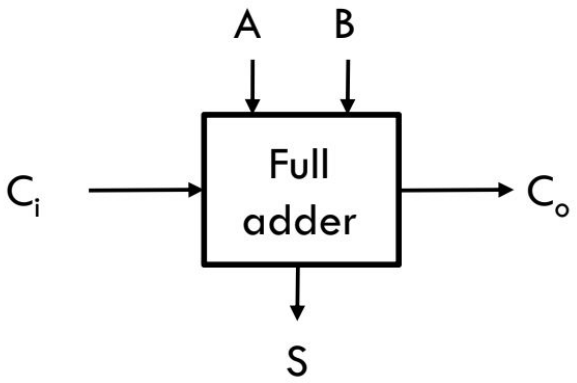
Discussion 11

Adders, Midterm 2

Sp19 Discussion 11 (adders): http://inst.eecs.berkeley.edu/~eecs151/sp19/files/discussion11.pdf
Sp19 Discussion 12 (multipliers): http://inst.eecs.berkeley.edu/~eecs151/sp19/files/discussion12.pdf

Full Adder



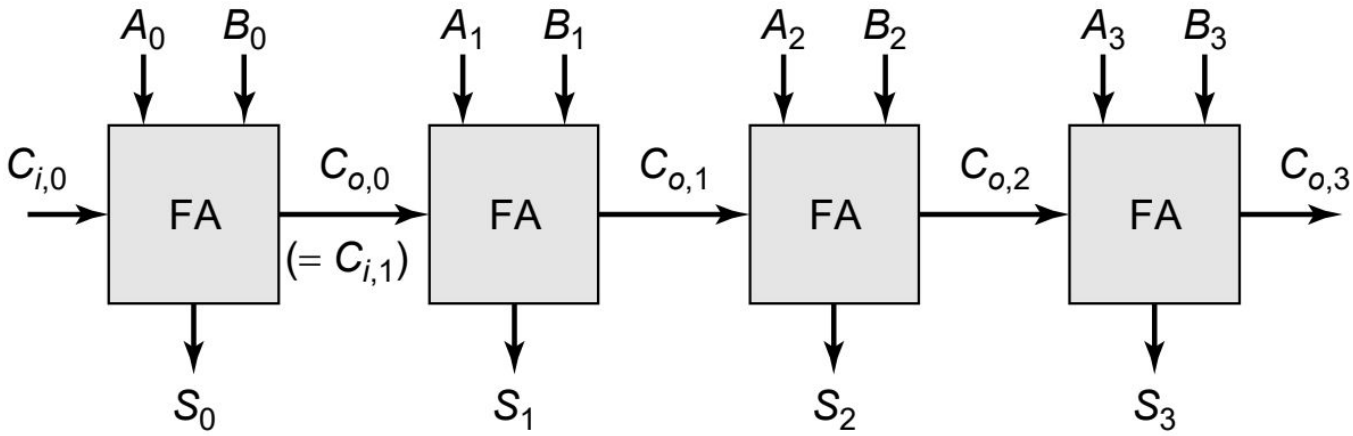
$$S = A \oplus B \oplus C_i$$

$$S = A\,\overline{B}\,\overline{C_i} + \overline{A}\,B\,\overline{C_i} + \overline{A}\,\overline{B}\,C_i + A\,B\,C_i$$

$$C_o = A\,B + B\,C_i + A\,C_i$$

- A full adder implements a single-bit adder with carry in
- A half adder doesn't have a carry in, but still has a carry out
- The full adder is the primitive used in many adder topologies
- The path from Ci -> Co is often on the critical path

Ripple Carry Adder



- For a N-bit ripple carry adder
  - Delay = (N-1) * t_{FA,Ci -> Co) + t_{FA, Ci -> S) = O(N)
  - Area = N * A_{FA} = O(N)
- Delay is dominated by the carry chain

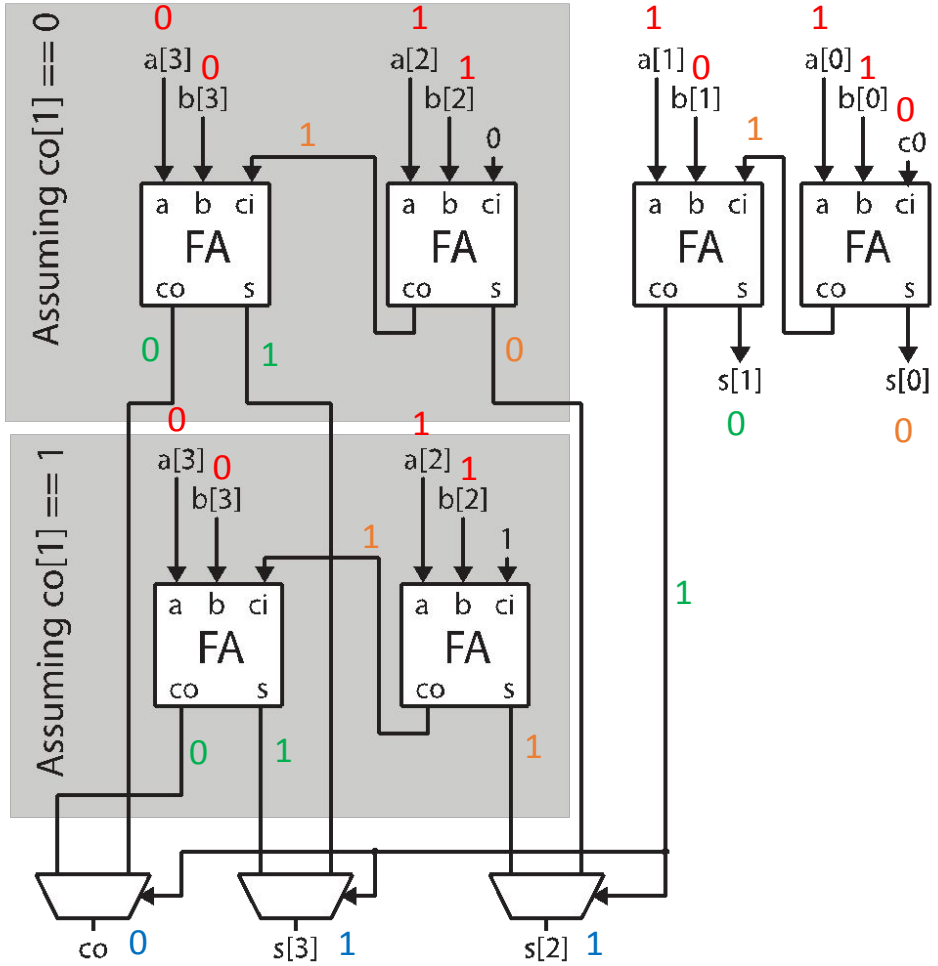
Carry Select Adder

- One way to reduce critical path is to cut the adder into 2 parts, severing the carry chain.
  - Problem: The LSB side of the adder will work as expected but the MSB side still depends on the value of the carry!
  - Solution: There are 2 possibilities for the carry-in to the MSB adder, 0 and 1. Calculate the result of **BOTH** cases and pick the correct one
    - Allows the MSB computations to occur in parallel with the LSB calculation with a small delay to select the correct value
  - Downside: Replicated logic, wasted effort (energy) on result that is not used

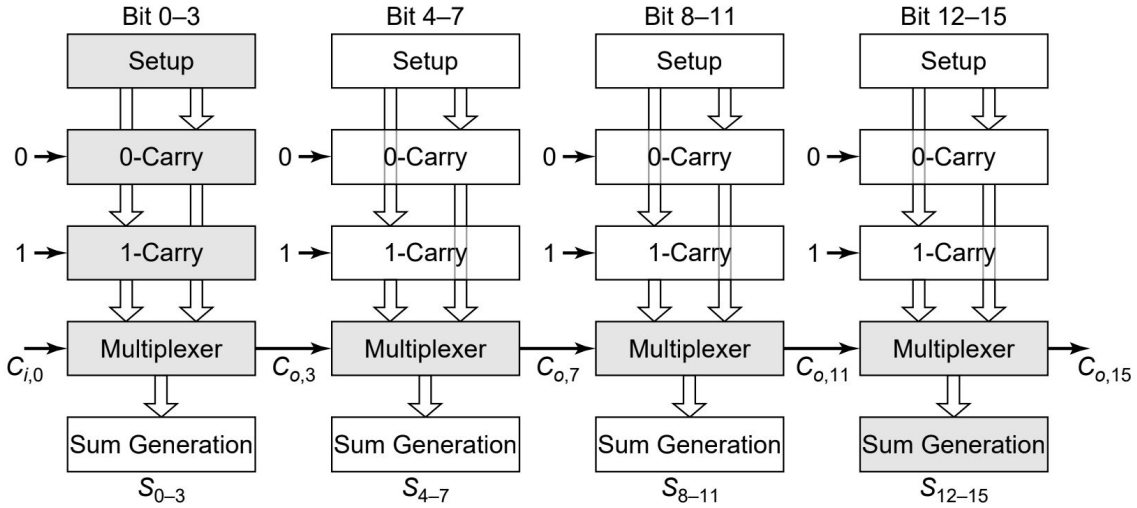
Carry Select Example:

Example:

```
     4'b0111 (7)
   + 4'b0101 (5)
-------------
    5'b01100 (12)
```

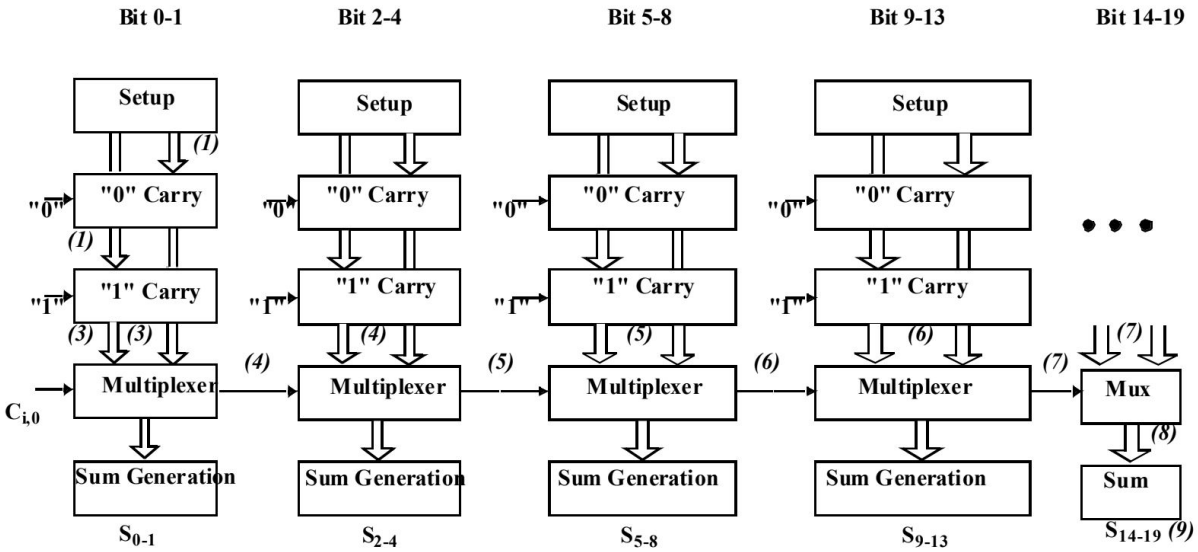


Linear Carry-Select



- For a N-bit linear carry-select adder with M sections:
- Delay = (N/M) * t_{FA,Ci -> Co) + M * t_{mux} + t_{FA,Ci -> S}
  - Take a partial wrt M, M_{opt} = sqrt(N)
- Area = 2 * N * A_{FA} + M * A_{mux}

Non-Linear Carry-Select Adder

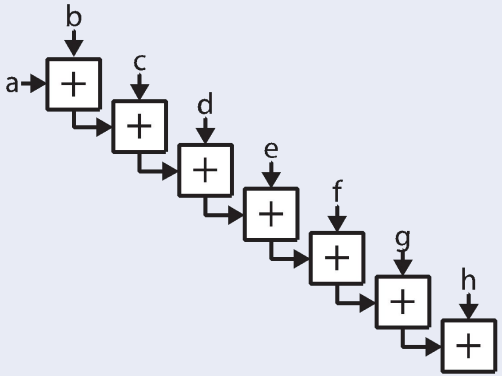


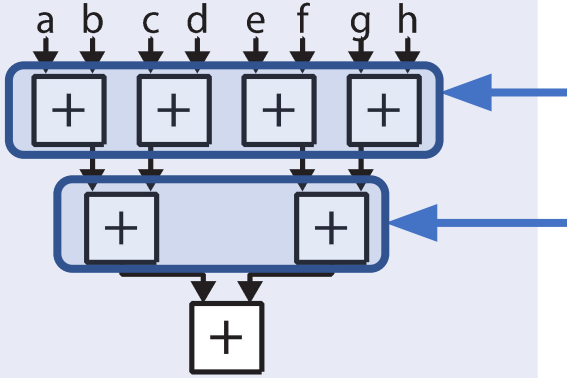
- A linear carry-select adder has the same number of FAs per stage
- This is non-optimal because all FA chains complete in parallel at the same time, and thus arrive at the muxes before the control signal arrives
- The goal is the equalize the delays of each stage
- t_{stage0, setup->Co) = t_{stage1, setup->mux} = … = t_{stageM, setup->mux}
- Solution depends on mux vs FA delay, solve numerically

Quick Aside: Associativity

- An operator, #, is associative if the following is true: (a # b) # c = a # (b # c)
- Addition*, multiplication, AND, OR, XOR, are associative
- This allows us to compute them in a tree structure
  - Ex. Compute: a+b+c+d+e+f+g+h

((((((a+b) + c) + d) + e) + f) + g) + h	((a+b) + (c+d)) + ((e+f) + (g+h))
Delay: 7 Additions HW Requirements: 7 Adders	Delay: 3 Additions HW Requirements: 7 Adders





See: Weisstein, Eric W. "Associative." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/Associative.html

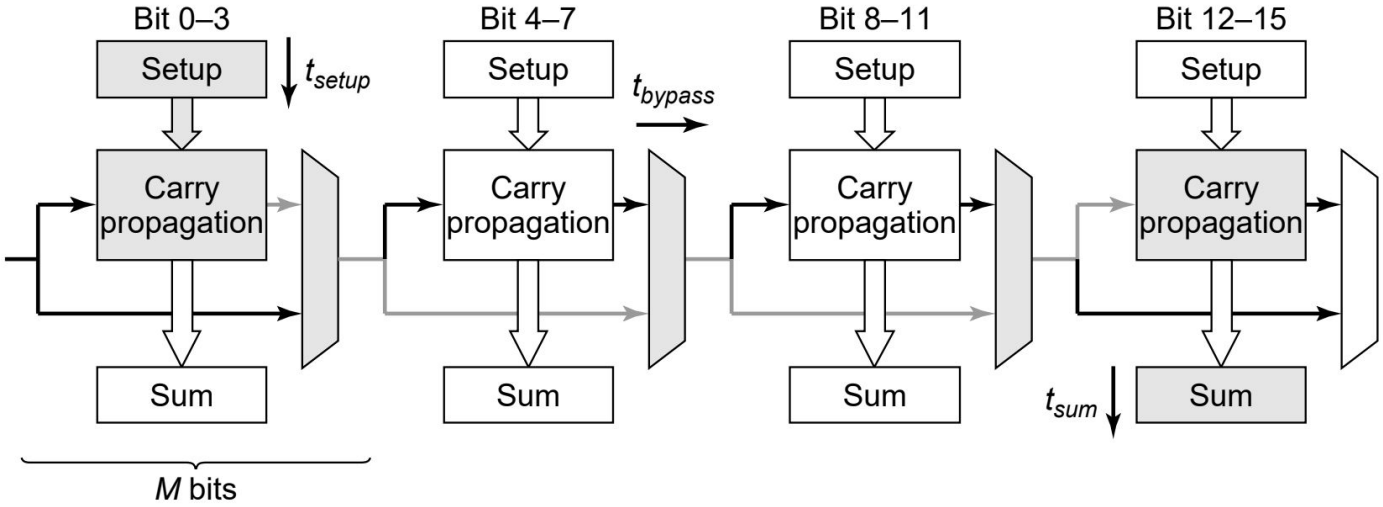
* Addition of floating point numbers is generally not considered associative

Propagate/Generate

- We want to make our adder topology associative so we can avoid the carry propagation being in the critical path. Define p, g signals:
- Propagate (p) indicates when Co = Ci
- Generate (g) indicates when Co = 1
- p = A + B (= A XOR B)
  - Propagate can be high along with generate, as its redundant
- g = AB
- Let's use propagate to implement a carry-bypass adder

A	B	C_i	S	C_o	*Carry status*
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

Carry-Bypass (Carry-Skip) Adder



- Muxes are controlled by block propagate (P = p1*p2*...*pM)
  - P can be computed using a tree (log critical path)
- Critical path starts at LSB carry chain, then goes through carry bypass muxes, and the final carry chain to create the sum bits
- Why doesn't the critical path go through every carry block? If block propagate is true, then none of the carry paths matter. If it's false, then the carry in doesn't matter! (false path)

Carry Lookahead Adder

- The carry logic, as we have presented it, is not associative
  - We need to compute the bits in order from LSB to MSB, since each FA needs the carry-out of the previous stage
- This is a problem as it limits us to a linear chain of FAs, preventing us from doing work in parallel
- Solution: Make the carry logic associative through re-defining the FAs

Redefining FAs: Carry Generate and Propagate

- Each FA Now Generates 2 New Signals
  - g (Generate): True if the adder is guaranteed to **generate a carry**, regardless of the value of the carry-in
    - If both operands have a 1 in this position, it is guaranteed that a carry will be generated
    - $g_i = a_i \cdot b_i$
  - p (Propagate): True if the carry-out of this stage will equal the carry-in (**propagate carry-in**)
    - If exactly one of the inputs is true, the carry-out will equal the carry-in
    - $p_i = a_i \oplus b_i$

Redefining FAs: Carry Generate and Propagate

- The sum and carry-out of a FA can now be defined in terms of these new signals
  - The sum is true if:
    - A single input is true and the carry-in is false
    - The inputs are both 0 or both 1, and the carry-in is true
    - $s_i = p_i \oplus c_i$, where c_i is the carry-in for this digit
  - The carry-out is true if:
    - Carry generate is true
    - Propagate is true and the carry-in is true
    - $c_{i+1} = g_i + p_i \cdot c_i$

What good did that do?

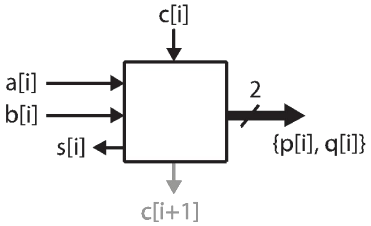
- Note that the sum and carry-out bits in each FA still depend on the values of the carry-in.
  - This means that we still need the compute the carry-in value for each bit position and have logic to generate the sum
- However, the p and g values can all be computed simultaneously
  - There is **no dependence** on carry-in when computing p and g!
- We leverage this property in the carry lookahead adder by grouping together adders and creating P and G signals for the entire group
  - P represents if the entire group will propagate a carry signal
  - G represents if the entire groups generates a carry signal
- The P and G signals can be processed in a tree structure

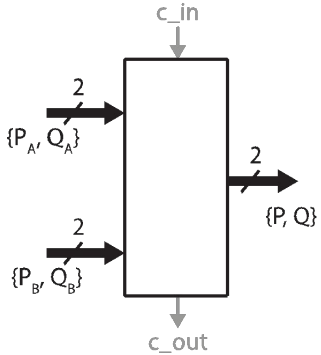
Carry Look-ahead Adder

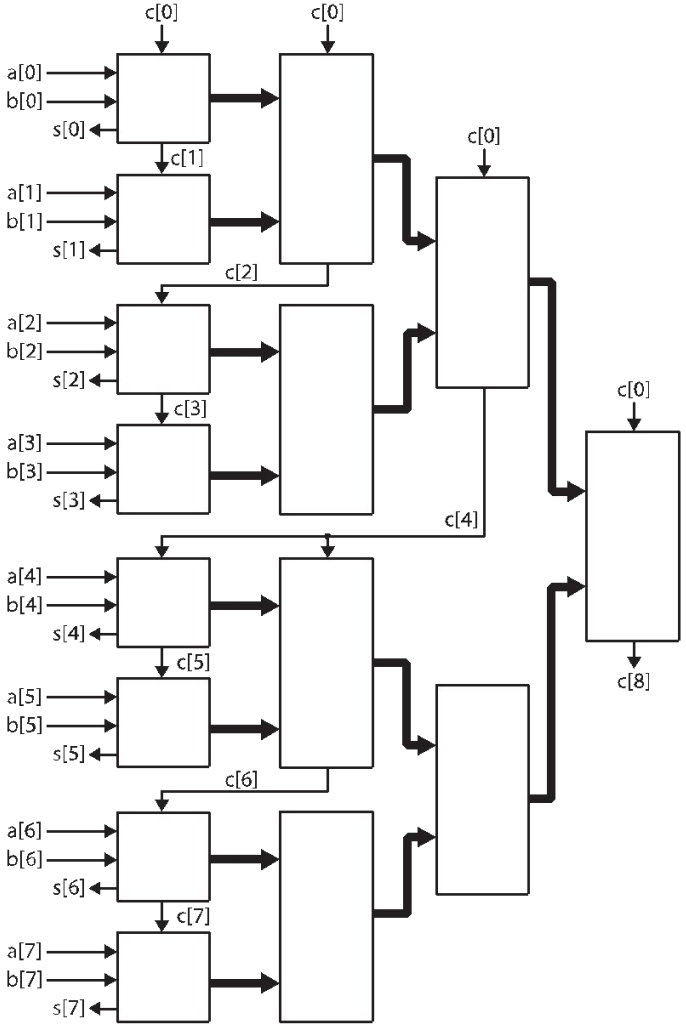
- The smaller blocks are modified full adders.
  - Can calculate g and p immediately
  - Must wait for carry-in to compute sum bit
  - Some FAs are required to create a carry-out
  - $g_i = a_i \cdot b_i$
  - $p_i = a_i \oplus b_i$

  No Dependence on carry-in
  - $s_i = p_i \oplus c_i$
  - $c_{i+1} = g_i + p_i \cdot c_i$

  Depend on carry-in





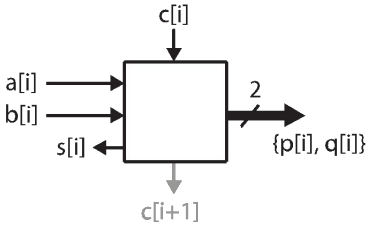


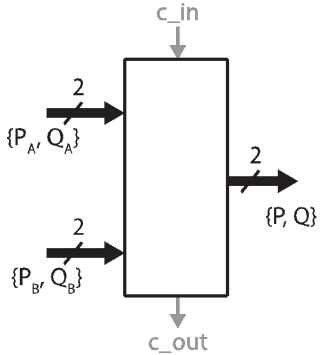
Carry Look-ahead Adder

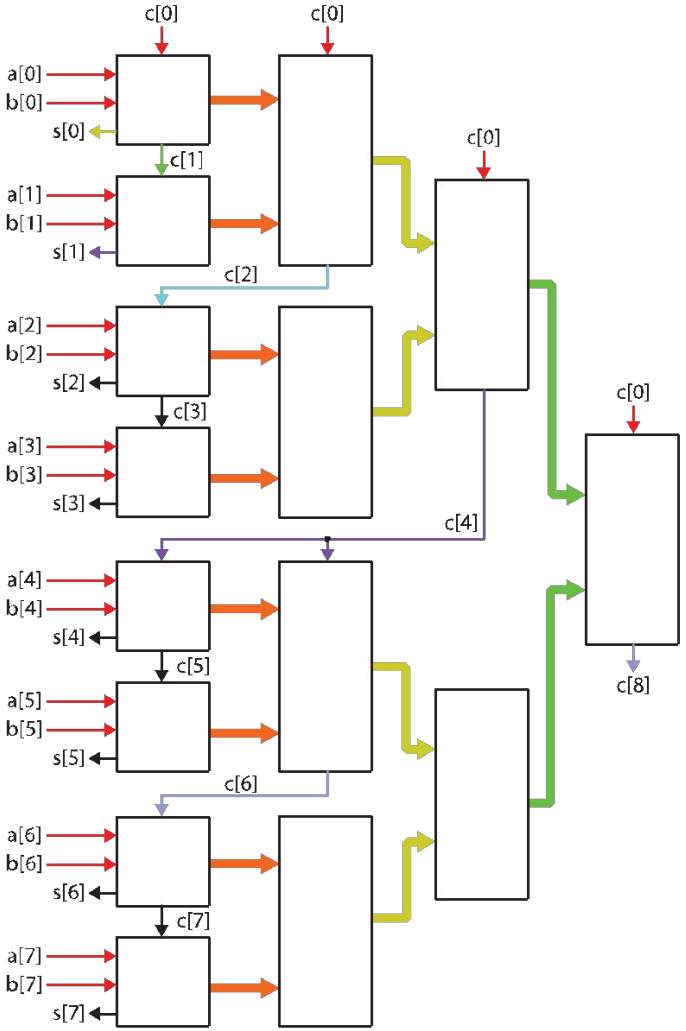
- The larger blocks compute P & G for higher levels of the hierarchy.
  - P & G can be computed without carry-in
  - Carry-in is required to generate carry-out
  - $P = P_A P_B$
  - $G = G_B + G_A P_B$

  No Dependence on carry-in
  - $C_{out} = G + C_{in} P$

  Depend on carry-in







Parallel Prefix Adder

- One disadvantage of the carry lookahead adder as described in the lecture slides is that the carry-out bit still ripple through the groups in the first layer
- An alternative is to compute the carry bits directly without any grouping
  - However, we don’t want to fall back to a carry ripple solution.
  - Trick: unroll the expression for the carry bit

Unrolling the Carry-in

- Recall: $c_{i+1} = g_i + p_i \cdot c_i$
- Let's compute the caries using unrolling
  - $c_0 = 0$ (unsigned)
  - $c_1 = g_0 + p_0 \cdot c_0 = g_0$
  - $c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1(g_0) = g_1 + p_1 g_0$
  - $c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2(g_1 + p_1 g_0) = g_2 + p_2 g_1 + p_2 p_1 g_0$
  - $c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0) = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

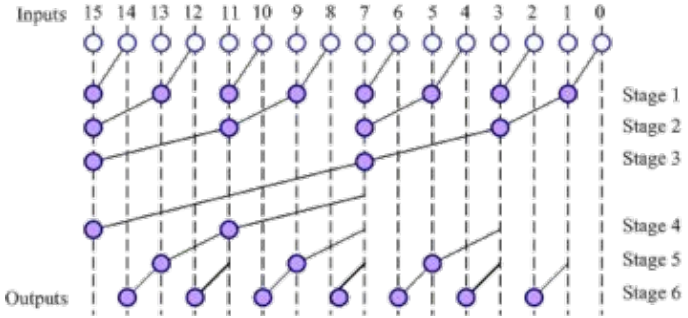
- Computing the caries involves ANDs and ORs of individual p and g signals
  - These p and g signals can all be computed in parallel since they do not depend on carry-ins
- These operations are associative!
  - We can change the order in which they are evaluated
  - Allows us to compute them in a tree (parallel computation)!

Parallel Prefix Trees

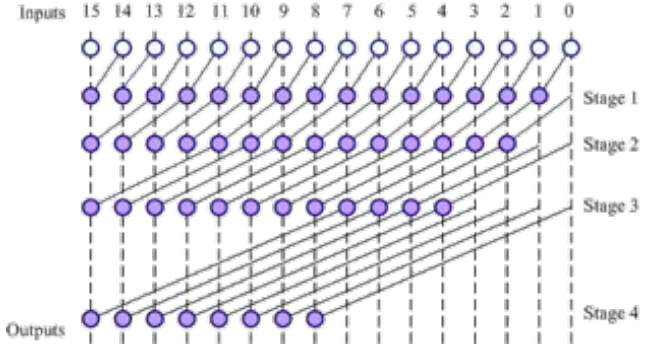
- Similar to a reduction tree except that you want to keep the intermediate values
  - Intermediate values are re-used when computing
- In our case, we *could* use a reduction tree to compute the last carry.
  - This would be of limited use to us because we need all of the intermediate carry bits that would be computed as part of the reduction tree
- Parallel prefix trees give us these intermediate values!
  - Work on operators that are associative

Different Parallel Prefix Trees

- There is a tradeoff in parallel prefix trees in how intermediate values are computed/reused
- Note that both of these graphs produce the same outputs (the partial results)



Brent-Kung (Diagram from Lecture Slides)
Most reuse (minimal logic) but with a longer critical path



Kogge-Stone (Diagram from Lecture Slides)
Requires more resources but has a shorter critical path.

Parallel Prefix Adder

- The Parallel Prefix Tree Described Above is for computing the **carry bits**
- We **still need full adders** to produce the p & g signals and to calculate the final sum
  - Modified **full adders feed the parallel prefix tree** with p and g values
  - **Full adders receive the carry in from the parallel prefix tree** to compute the sum bit

Midterm 2

1) *It's all logical…* (16 points, 20 minutes)

a) The following function F that implements an OR-AND-Invert OAI21 gate.

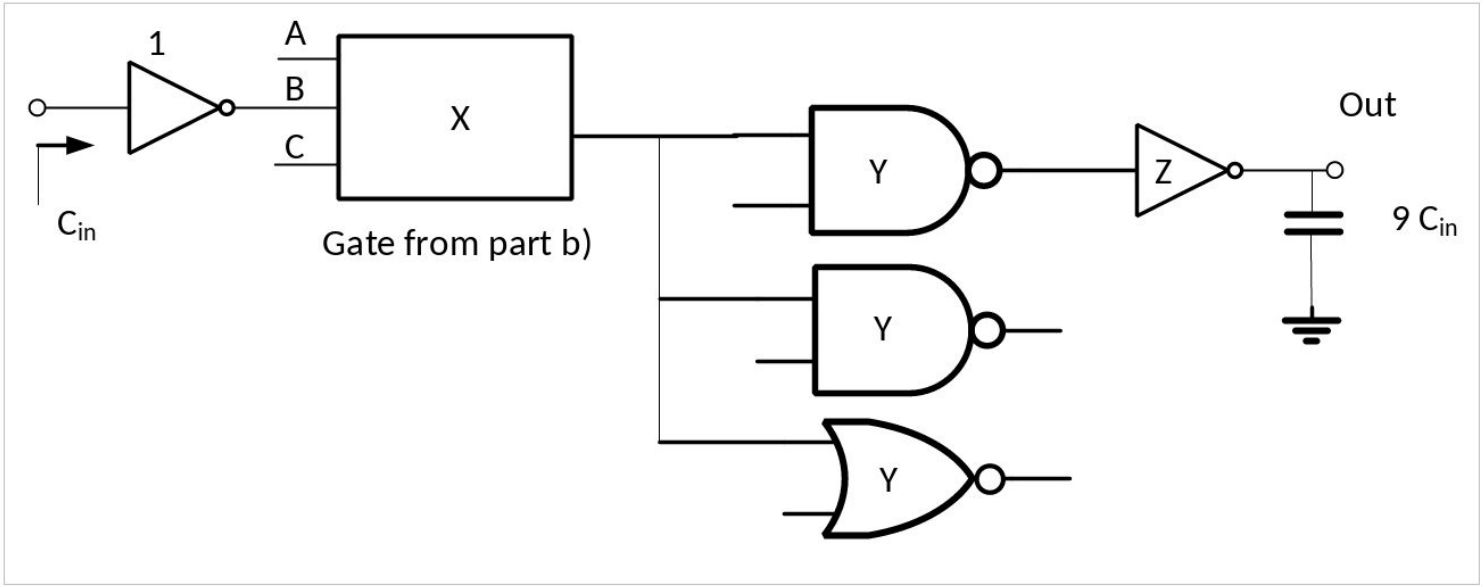
$$F = \overline{(A+B)C}$$

Implement the function F as standard, complementary CMOS logic gate. Size your transistors to match the pull-up/pull-down resistances to those of a unit inverter. You can assume NMOS and PMOS devices have equal strength in this technology.

b) Find the logical effort for the input B for the gate from part a).

Midterm 2

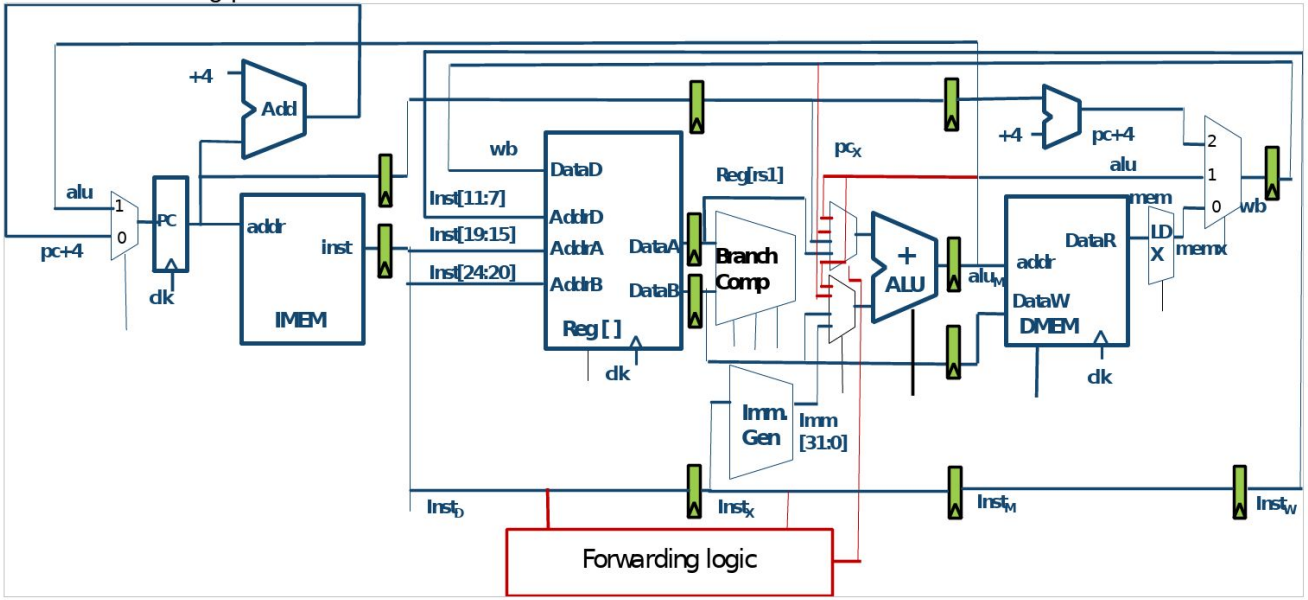
c) The gate from part a) is used in the following circuit. Determine the gate sizes, X, Y, Z, that minimize the delay in the path below. If you're not confident about your answer to part c, assume the logical efforts of the new gate is 3. Gates sized as '1' have the equivalent driving resistance and input capacitance equal to a unit-sized inverter.



Midterm 2

2) *RISC-V Pipelining and Hazards…* (24 points, 26 minutes)

Consider the 5-stage pipeline presented in lecture with combinational-read IMEM and DMEM and the forwarding paths as drawn.



Fill the pipeline diagrams for the following assembly program. Indicate when NOPs are injected into the pipeline by writing 'NOP'. Assume you can write to and read from the same address in the register file (Reg [ ])

Case 1: The forwarding paths in the diagram are unused and all hazard resolution is accomplished with stalling.

Case 2: The forwarding mux in the diagram is driven correctly by the forwarding logic.

```
1           addi x1, x0, 0xFF
2           andi x2, x1, 0xF
3           bge x1, x2, label
4           xori x2, x2, 1
5           ori x3, x2, x1
6    label: lw x5, 0(x6)
7           sw x5, 4(x6)
```

Midterm 2

b) Answer true or false for the following statements, about the 5-stage pipeline in this problem.

i) A mispredicted branch instruction causes 2 instructions to be killed

◯ True ◯ False

ii) Jumps (**`jal`**, **`jalr`**) cause 3 NOPs to be injected into the pipeline

◯ True ◯ False

iii) Using forwarding paths could increase CPI (clocks per instruction) over stalling-only baseline

◯ True ◯ False

iv) Immediate generation, in general, can proceed in parallel with register file reads

◯ True ◯ False

v) Making the IMEM and the DMEM synchronous read would yield a 8-stage pipeline

◯ True ◯ False

vi) If the immediate generation was moved to the F stage, and you had an additional adder, you can make **`jal`** inject no NOPs

◯ True ◯ False

Midterm 2

3) *Energy and Performance* (10 points, 12 minutes)

a) We would like to examine some properties of a single-cycle RISC-V datapath. The datapath presents a total load capacitance of 5pF to the supply, operates at 500 MHz, and has an activity factor of $\alpha_{[0 \rightarrow 1]} = 0.1$. V_{DD} = 1V. The CPI is 1.

$$CPI = \frac{Clocks}{Instruction}$$

$$\frac{Seconds}{Program} = \frac{\frac{Instructions}{Program} * Cycles}{\frac{Instruction}{Cycle}} * Seconds$$

a) What is the dynamic power consumption?

P = _____.

b) What is the average energy per instruction?

E_{inst} = _____.

Midterm 2

c) How much energy is drawn from the supply to run a program with 1000 executed instructions?

E_{1000} = _____.

d) What is the energy-delay product for the case in c) ?

EDP = _____.

e) You are able to pipeline this design so a 5-stage pipeline operates at 1.5GHz, at the same supply V_{DD} = 1V, has the same activity factor, and has a CPI of 3. The load capacitance has increased by 50% compared to the non-pipelined baseline. Calculate the energy-delay product when running the same program as in c)

$EDP_{5\text{-stage}}$ = _____.

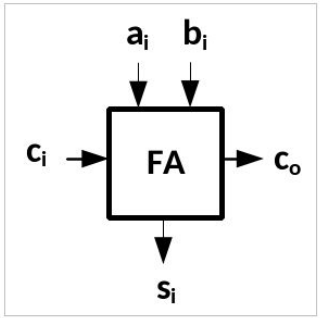
Midterm 2

4) *Delays and Adders* (20 points, 22 minutes)

You are designing a datapath in a brand-new CMOS FinFET technology that has only four CMOS gates: 2-input NAND, 2-input NOR, 2-input XOR, and an OAI21 gate ($Y = \overline{(A+B)C}$). Gate capacitance equals drain capacitance per unit area ($\gamma = 1$), and the four gates come in only one size each, i.e., you do not need to size gates in this problem.

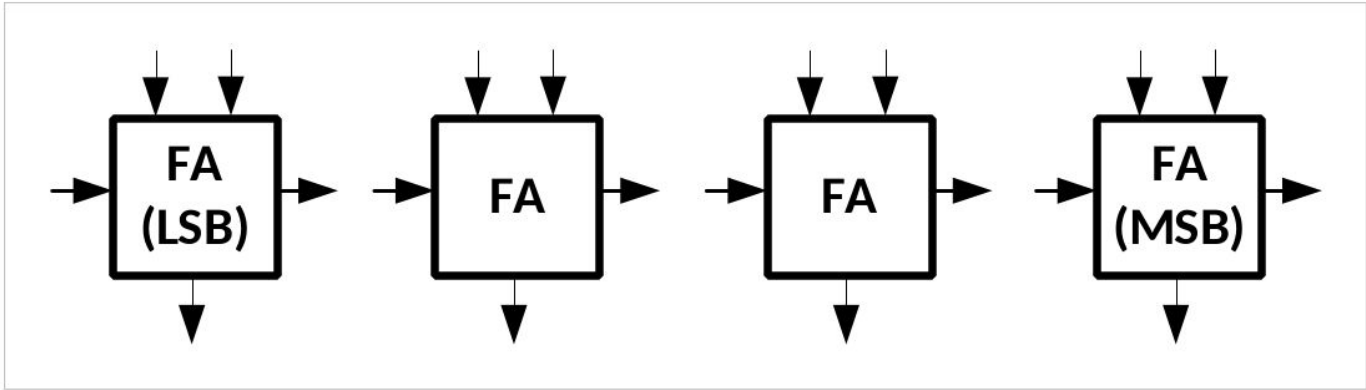
a) If both the 2-input NAND gate and the 2-input NOR gate have a delay dependence on a fanout, f, given as $t_{NAND2} = t_{NOR2} = 2ns(4 + 3f)$, what is the delay dependence on the fanout of the input C of the OAI21 gate (C is in the critical path)?

b) Design a fast full-adder using the four types of gates and calculate the ci->sum and ci->cout delay. Note that using an OAI21 gate could potentially simplify the logic for C_out. Assume the capacitance driven by the sum and the carry bits is equal to the capacitance of inputs a_i, b_i and $t_{XOR2} = 2ns\ (8 + 8f)$.



Midterm 2

c) Complete the drawing and label all inputs and outputs for an 4-bit ripple-carry adder in figure below by using full-adder (FA) cells from part b). Inputs are a[3:0] and b[3:0] and there is no carry-in to the least-significant bit.



Midterm 2

d) Find the critical path delay for the circuit in part c), if the capacitance driven by the sum and the carry bits is equal to the input a_i, b_i capacitance. If you are not confident in your answer in part b), you can express the delay in terms of $t_{FA,Ci \rightarrow Co}$, and $t_{FA,Ci \rightarrow S}$