EECS151/251A Discussion 11

Christopher Yarp

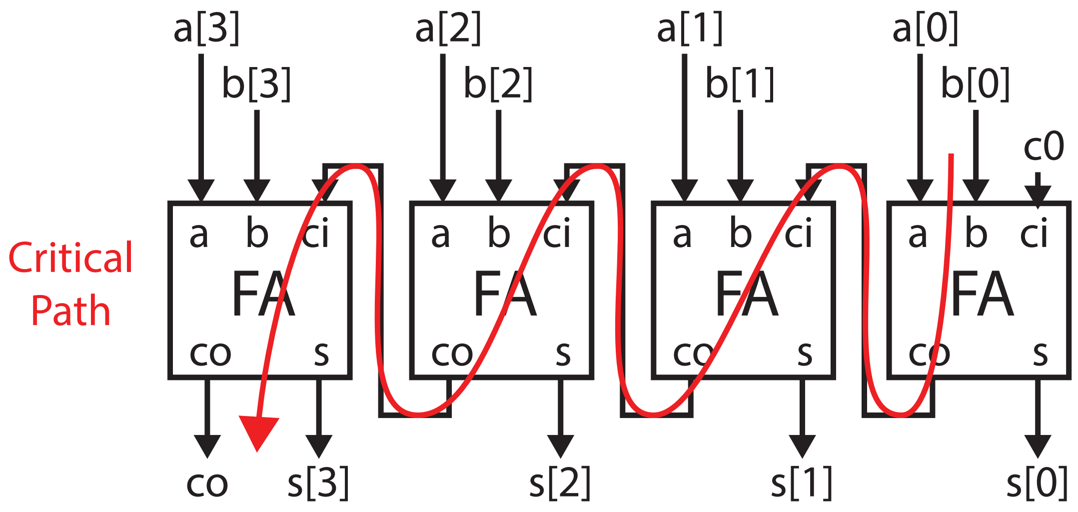
Apr. 19, 2019

Plan for Today

- Adders
- Multipliers
- Pipelining
- Questions

Adders

- Earlier in the term, we discussed the Carry Ripple Adder
  - Replicates how we add by hand
    - Compute sum[i] and carry_out[i] based on A[i], B[i], and carry_in[i]
    - carry_in[i] = carry_out[i-1], carry_in[0] = 0 if unsigned.
- Primary Downside: Long Critical Path
  - The carry ripple results in a critical path that goes through each FA
  - Grows linearly with the number of bits added
- How do we make adders faster?
  - Cut the critical path!
  - How?
    - **<u>Change how we work with carries!</u>**



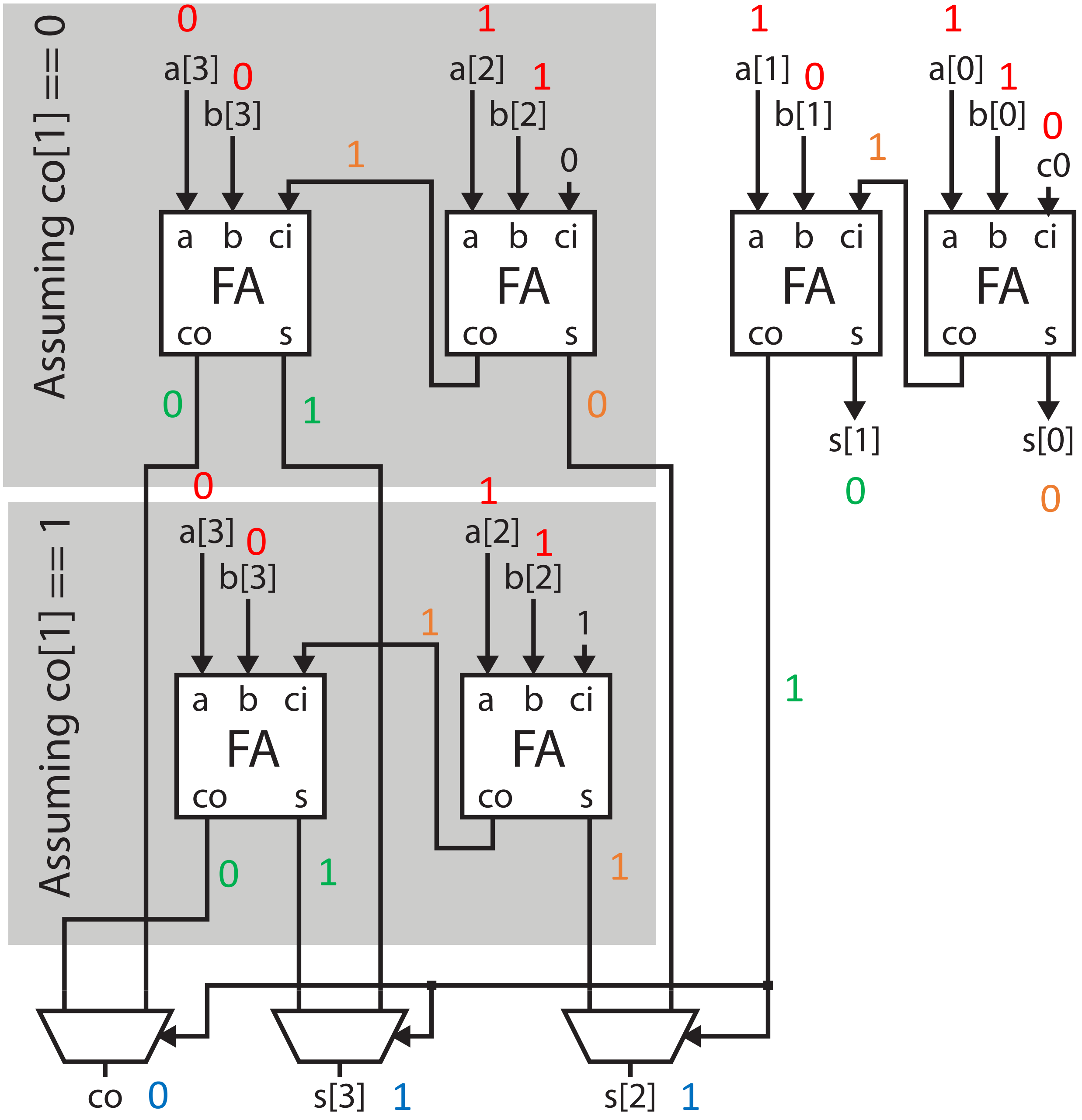
Carry Select Adder

- 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:

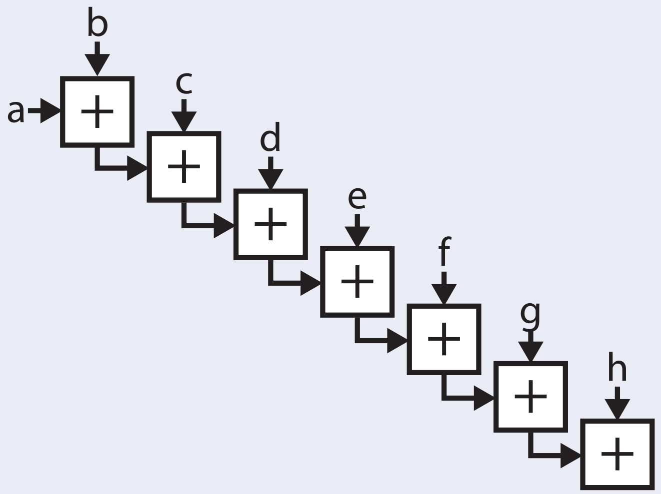
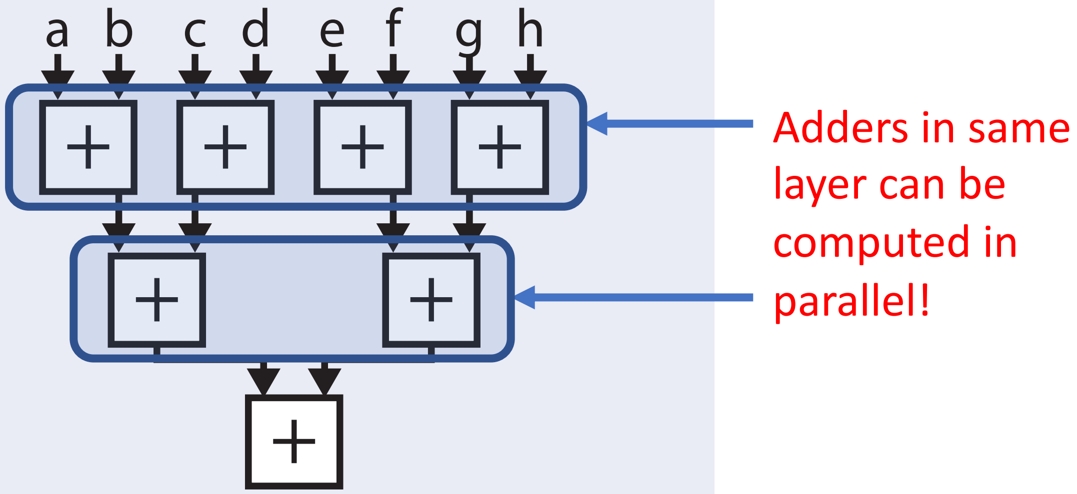
Example:

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



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

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

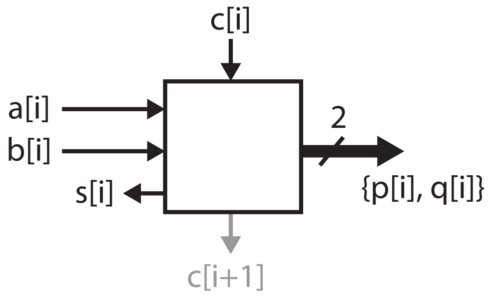
Carry Look-ahead Adder

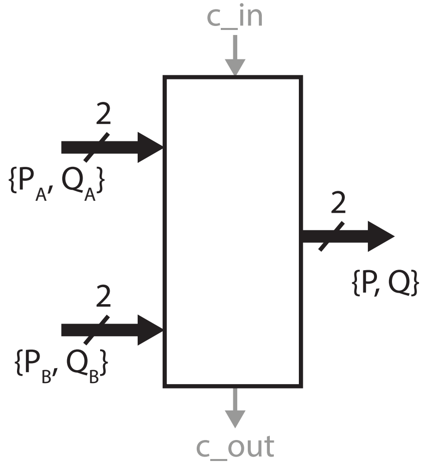
- The smaller blocks are modified full adders.
  - Can calculate g and p immediately
  - Must wait for carry-in to compute sum bit
  - Some FAs are required to create a carry-out
  - $g_i = a_i \cdot b_i$
  - $p_i = a_i \oplus b_i$

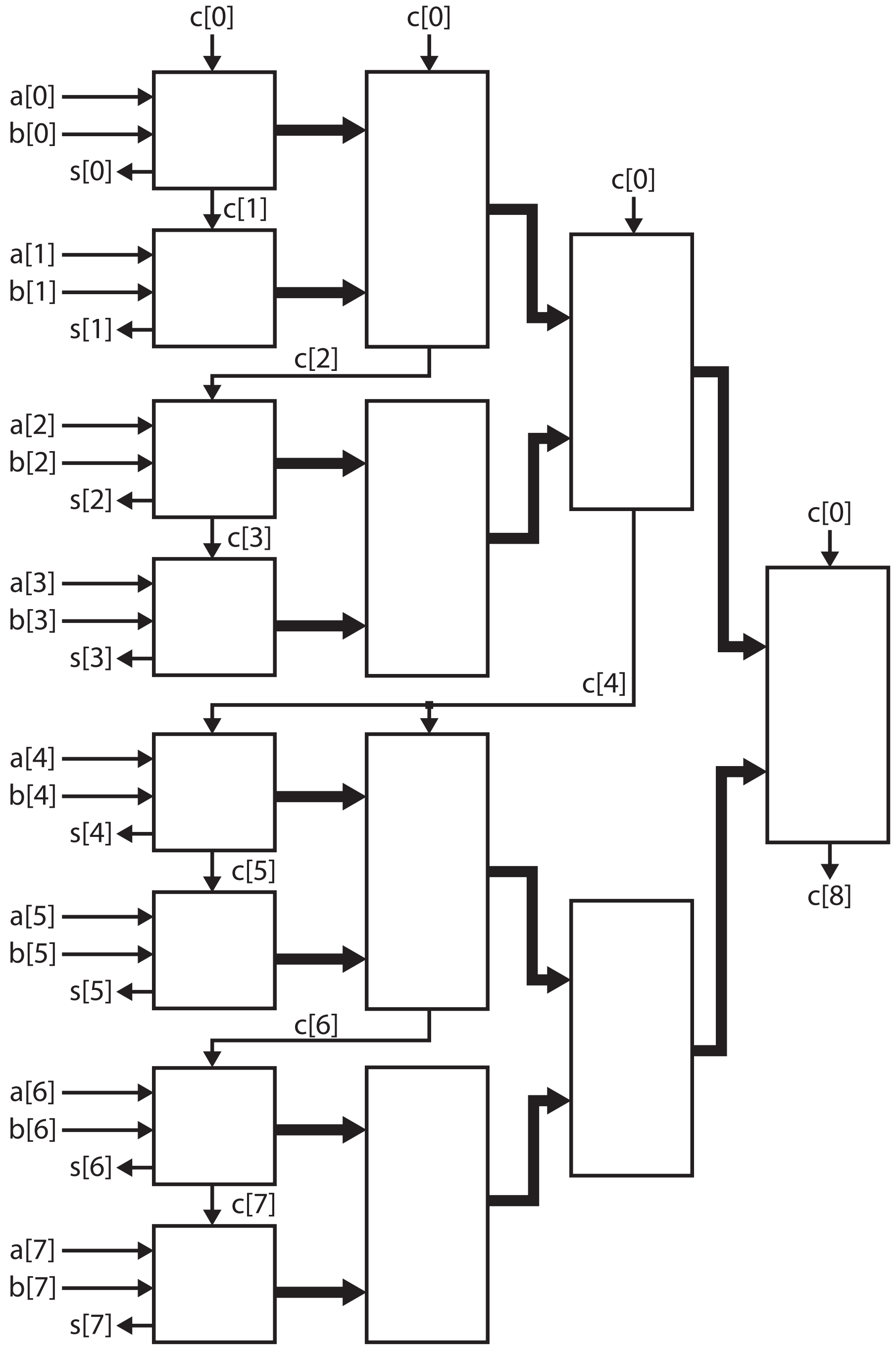
  No Dependence on carry-in

  - $s_i = p_i \oplus c_i$
  - $c_{i+1} = g_i + p_i \cdot c_i$

  Depend on carry-in







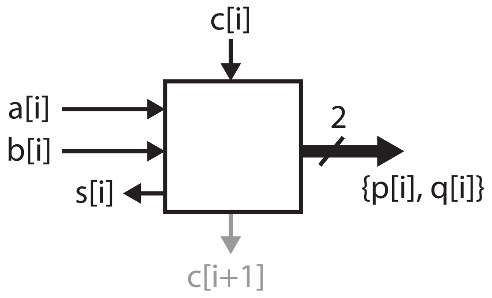
Carry Look-ahead Adder

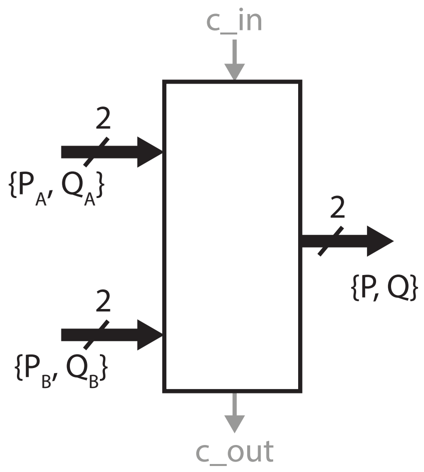
- The larger blocks compute P & G for higher levels of the hierarchy.
  - P & G can be computed without carry-in
  - Carry-in is required to generate carry-out
  - $P = P_A P_B$
  - $G = G_B + G_A P_B$

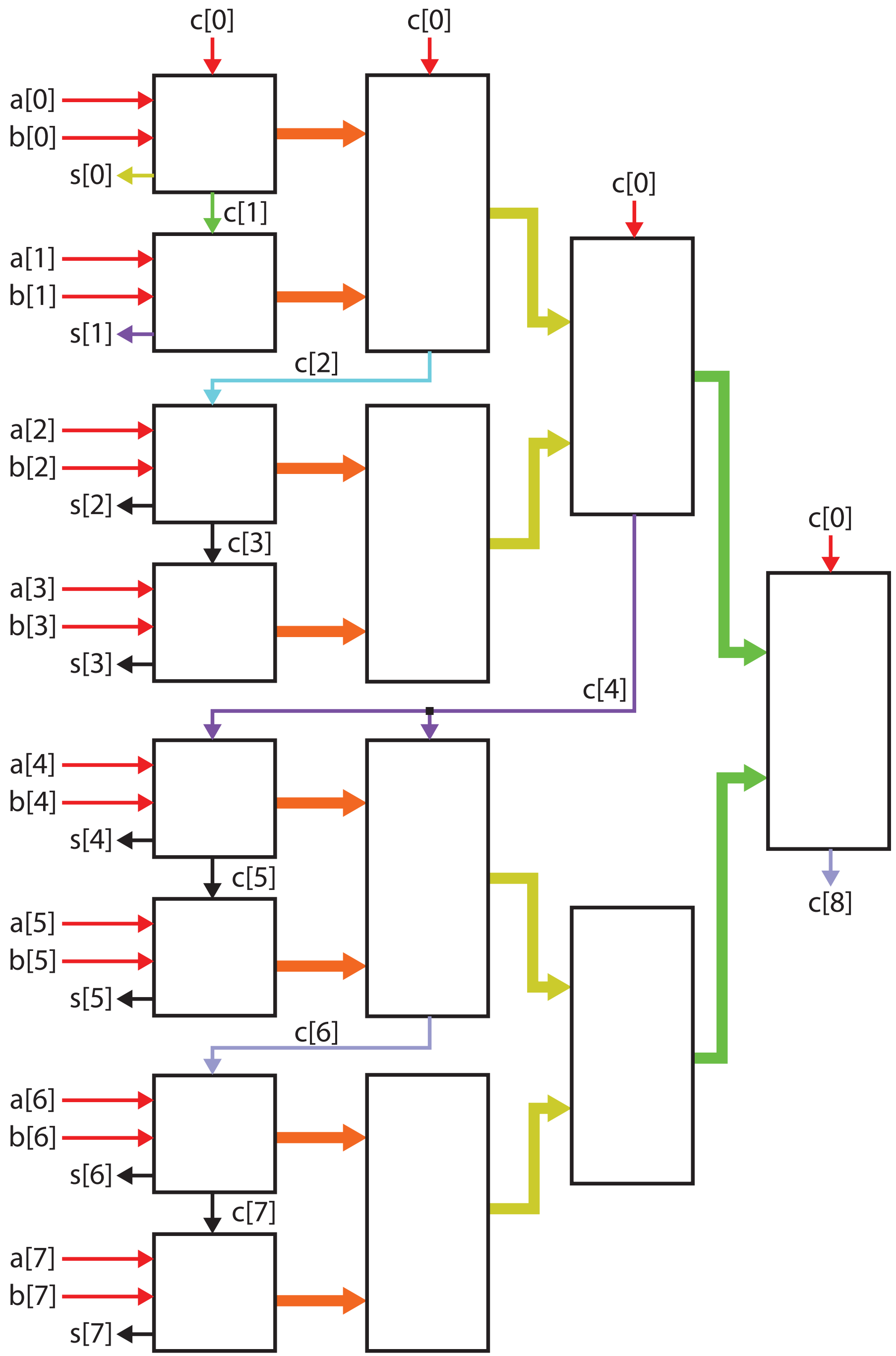
  No Dependence on carry-in

  - $C_{out} = G + C_{in} P$

  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)
  - $c_1 = g_0 + p_0 \cdot c_0 = g_0$
  - $c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1(g_0) = g_1 + p_1 g_0$
  - $c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2(g_1 + p_1 g_0) = g_2 + p_2 g_1 + p_2 p_1 g_0$
  - $c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0) = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

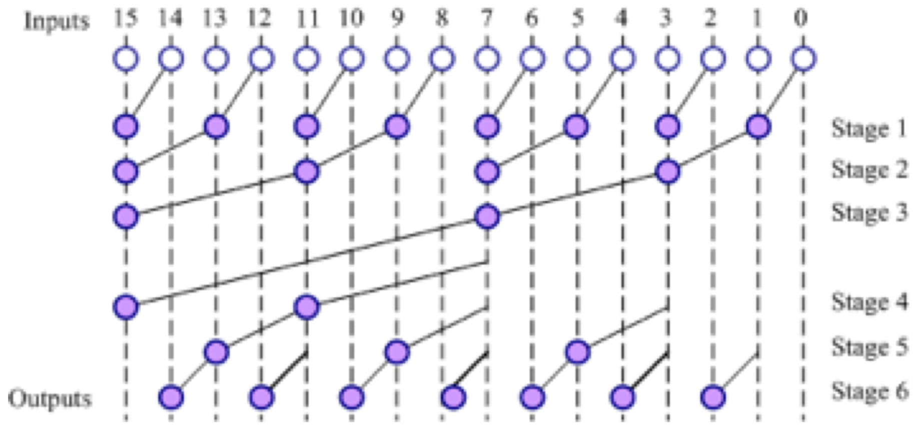
- Computing the caries involves ANDs and ORs of individual p and g signals
  - These p and g signals can all be computed in parallel since they do not depend on carry-ins
- These operations are associative!
  - We can change the order in which they are evaluated
  - Allows us to compute them in a tree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

Parallel Prefix Trees

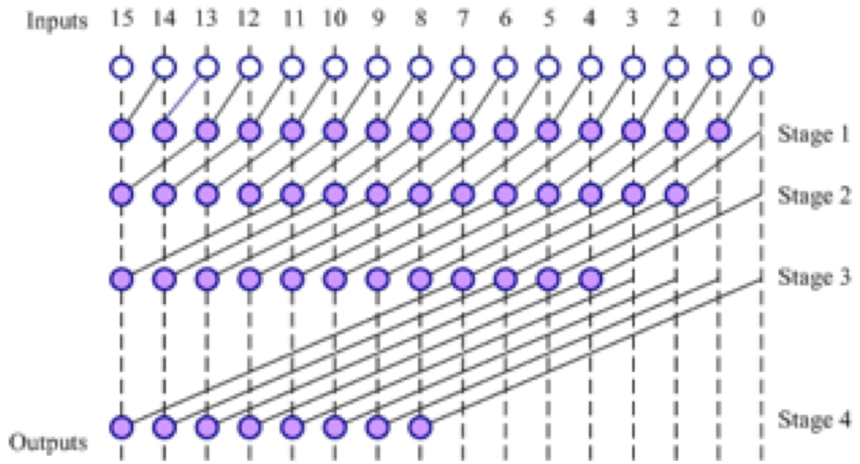
- Similar to a reduction tree except that you want to keep the intermediate values
  - Intermediate values are re-used when computing
- In our case, we *could* use a reduction tree to compute the last carry.
  - This would be of limited use to us because we need all of the intermediate carry bits that 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

Multipliers

- Remember, the mechanics of multiplication in binary are generally the same as decimal multiplication (signed multiply requires a slight tweak).
- 2 Steps to Multiplication:
  - Generation of partial products
  - Adding partial products
- Making faster multipliers mostly involves changing how we deal with generating and adding the partial products

Unsigned Multiplication Example

```
  4'b0011 (3)
* 4'b0110 (6)
```

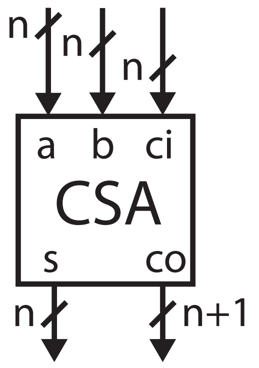
- Partial Products can be generated in parallel
- Let's try to improve the addition of the partial products

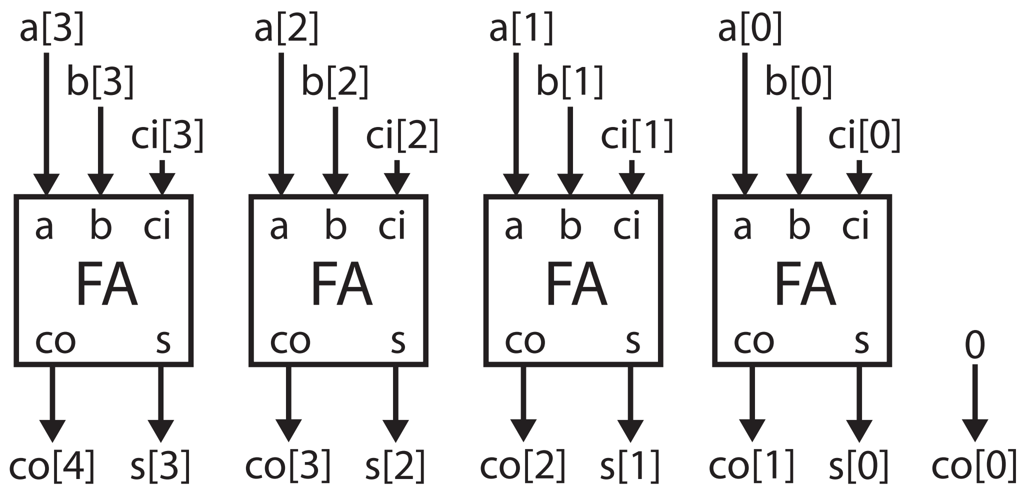
```
  4'b0011 (3)
* 4'b0110 (6)
  -------
     0000
    0011
   0011
+ 0000
---------
 00010010 (18)
```

Partial Products

Carry Save Addition

- When we generate a carry in a given column of an addition, we add it to the 2 values in the next column.
  - This addition may in turn generate its own carry
- If adding carries is just like another addition, can we delay adding the carry bits until later?
  - Yes, so long as we remember what the carry bits need to be added
- This is the basis of the carry save adder:
  - Takes in a, b, and carry_in (multi-bit)
  - Produces a sum and carry_out (multi-bit)



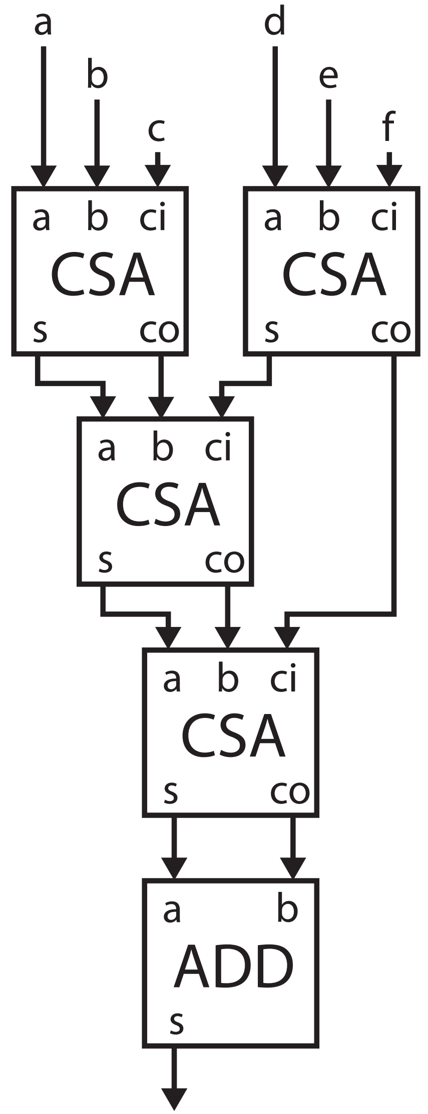


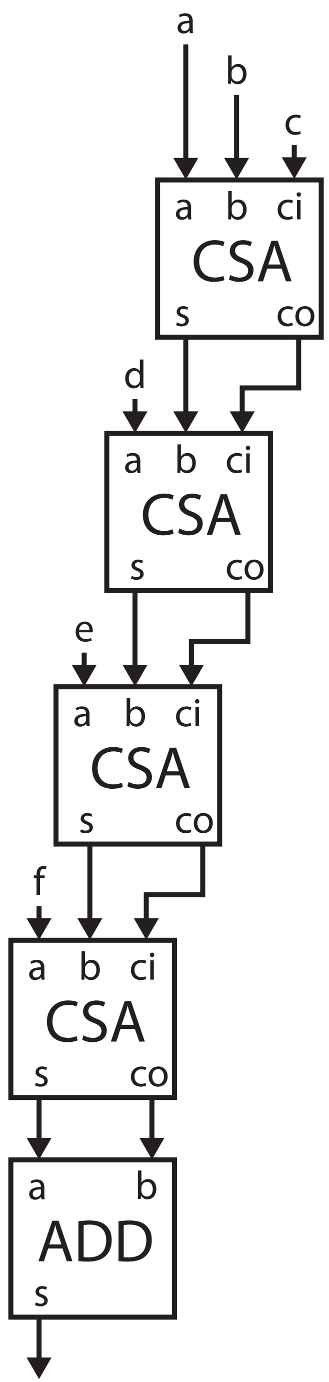
Using Carry Save Addition

- Using Carry Save Addition Allows us to create a multi-input adder that is:
  - Relatively fast: Carry Save Adders do not have a carry ripple
  - Relatively small: do not need the logic to handle the carry logic to create a fast adder
- However, still need a standard adder at the end to add the final carry-out and sum.
  - This is one of the fast adders such as the Carry Lookahead or Parallel Prefix Adders
  - Good news! We only need one of them.

Using Carry Save Addition

- Because addition is associative, it actually does not matter what order the carry bits are added back into the sum
  - Can use a tree structure

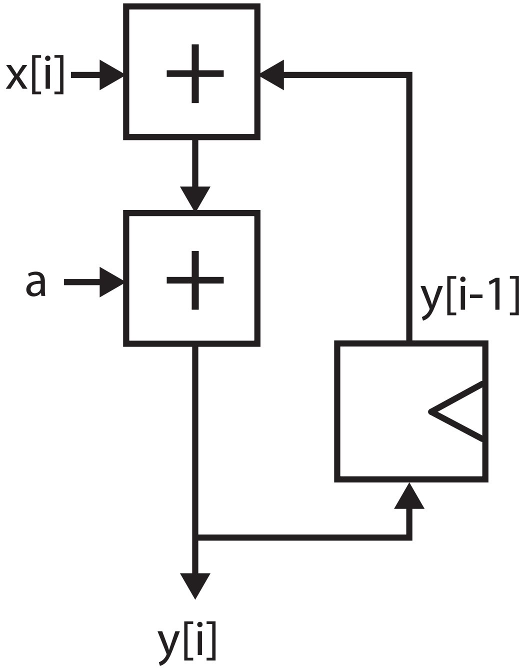




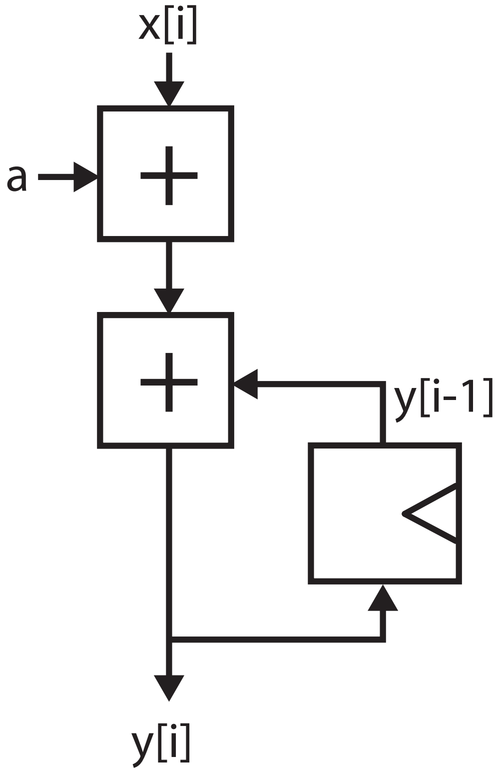
Quick Note on Pipelining With Feedback

- Pipelining in the presence of feedback is problematic due to the dependence
- However, if the feedback loop includes operators that are associative and commutative, we may be able to make the feedback loop shorter.
  - Tightening the feedback path pushes some logic outside of the loop
  - Logic outside of the feedback loop (feed forward) can usually be pipelined relatively easily.

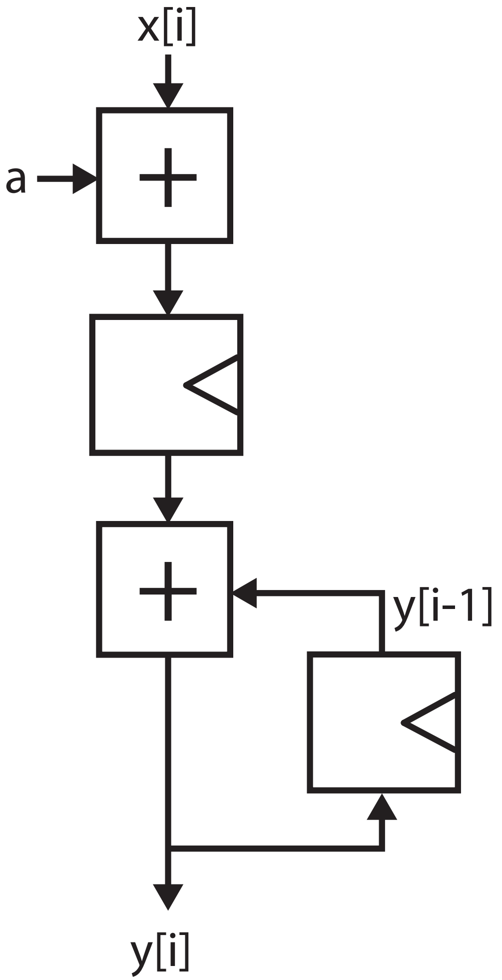
Example from Lecture



Orig: y[i] = (y[i-1]+x[i])+a



Reorg: y[i] = y[i-1]+(x[i]+a)



Feed Forward Section Pipelined