# COMP 322: Fundamentals of Parallel Programming

## Lecture 35: Algorithms based on Parallel Prefix (Scan) operations, cont.

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu



# Parallelizing Prefix Sum (Lecture 13)

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] \\ &= (A[0] + A[1] + A[2] + A[3]) + (A[4] + A[5]) + A[6] \end{aligned}$$

Approach:

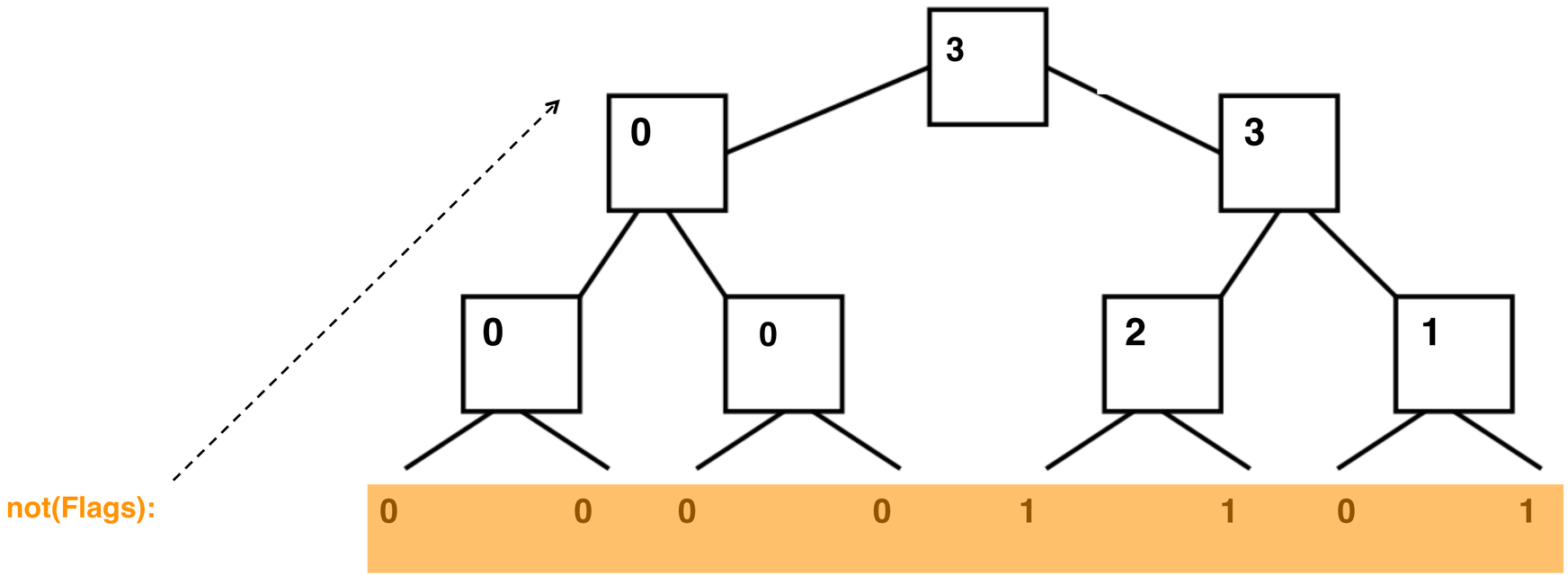
- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep



# Parallel Pre-scan Sum: Upward Sweep

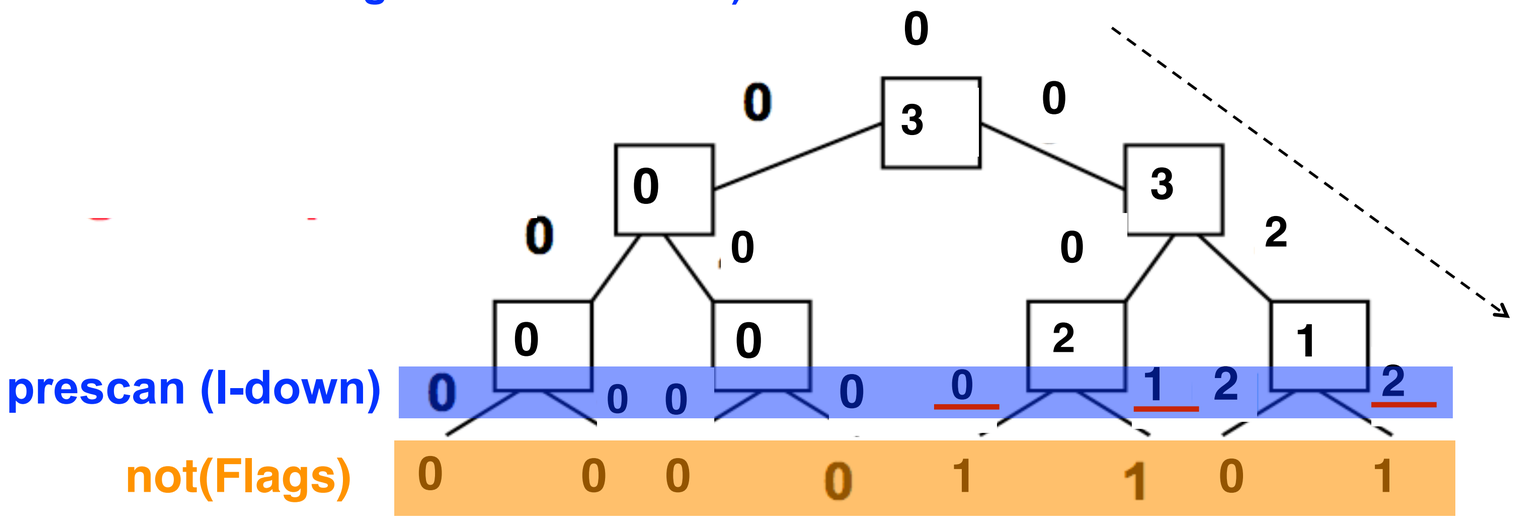
**Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way**

1. **Receive values from left and right children**
2. **Compute left+right and store in box**
3. **Send left+right value to parent**



# Parallel Pre-scan Sum: Downward Sweep

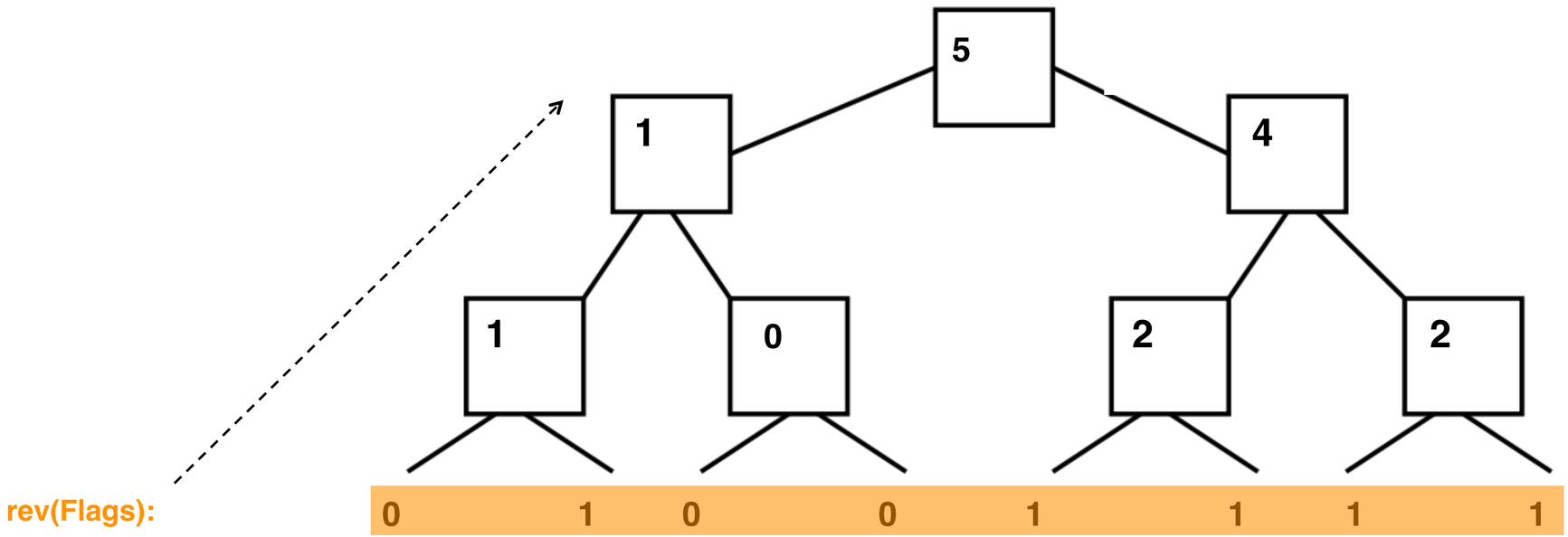
1. **Receive value from parent (root receives 0)**
2. **Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)**
3. **Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)**



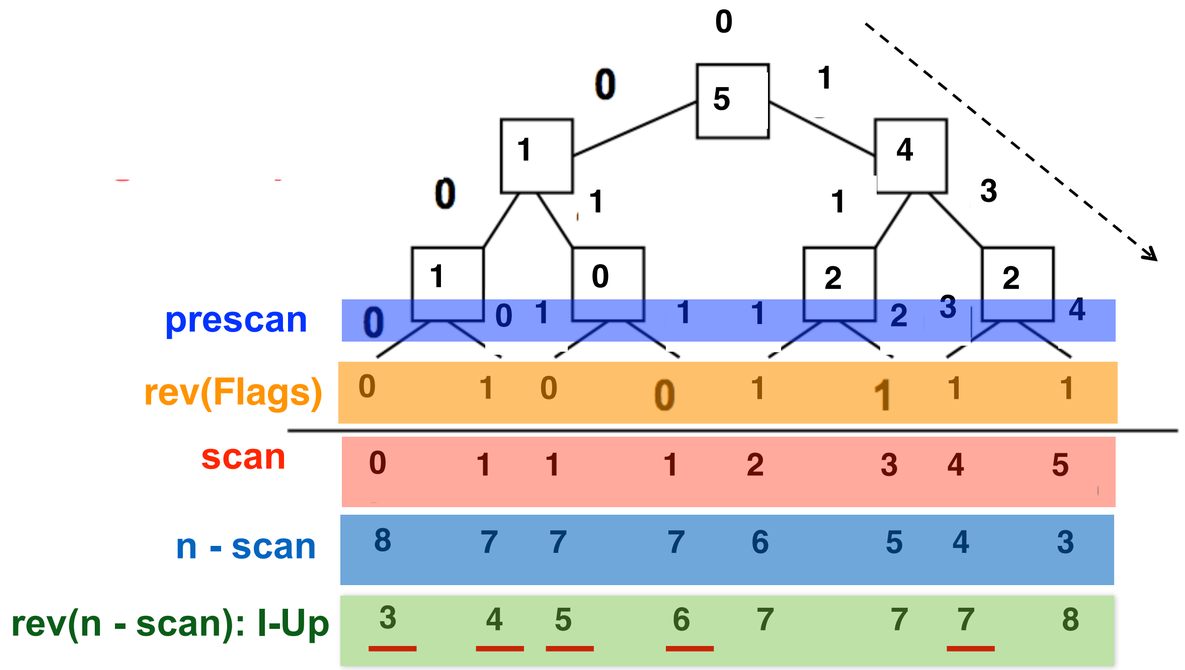
# Parallel Pre-scan Sum: Upward Sweep

**Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way**

1. **Receive values from left and right children**
2. **Compute left+right and store in box**
3. **Send left+right value to parent**



# Parallel Scan Sum: Downward Sweep



# Worksheet #34: Parallelizing the Split step in Radix Sort

**The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the bottom of a vector, and packs the keys with a 1 to the top of the same vector. It maintains the order within both groups.**

**The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation (complete I-down) can be performed in parallel.**

```
                                [101 111 011 001 100 010 111 010]
1.A =                           [5 7 3 1 4 2 7 2]
2.A⟨0⟩ =                         [1 1 1 1 0 0 1 0] //lowest bit
3.A←split(A,A⟨0⟩) =             [4 2 2 5 7 3 1 7]
4.A⟨1⟩ =                         [0 1 1 0 1 1 0 1] // middle bit
5.A←split(A,A⟨1⟩) =             [4 5 1 2 2 7 3 7]
6.A⟨2⟩ =                         [1 1 0 0 0 1 0 1] // highest bit
7.A←split(A,A⟨2⟩) =             [1 2 2 3 4 5 7 7]
```

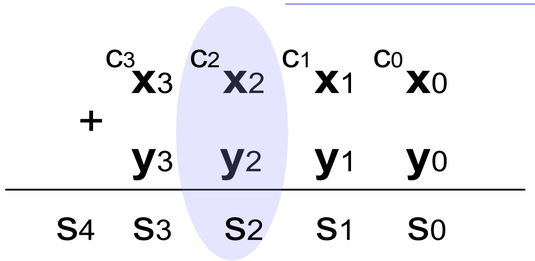
```
procedure split(A, Flags)
  I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum
  I-up   rev(n - scan(+, rev(Flags)) // rev = reverse
  in parallel for each index i
    if (Flags[i])
      Index[i] ← I-up[i]
    else
      Index[i] ← I-down[i]
  result ← permute(A, Index)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | = | [ 5 | 7 | 3 | 1 | 4 | 2 | 7 | 2 ] |
| Flags | = | [ 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 ] |
| I-down | = | [ 0 | 0 | 0 | 0 | 0 | 1 | 2 | 2 ] |
| I-up | = | [ 3 | 4 | 5 | 6 | 6 | 6 | 7 | 7 ] |
| Index | = | [ 3 | 4 | 5 | 6 | 0 | 1 | 7 | 2 ] |
| permute(A, Index) | = | [ 4 | 2 | 2 | 5 | 7 | 3 | 1 | 7 ] |

FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.



# Binary Addition




This is the pen and paper addition of two 4-bit binary numbers **x** and **y**.
**c** represents the generated carries.
**s** represents the produced sum bits.

A **stage** of the addition is the set of **x** and **y** bits being used to produce the appropriate sum and carry bits. For example the highlighted bits $x_2$, $y_2$ constitute **stage 2** which generates carry $c_2$ and sum $s_2$ .

Each stage *i* adds bits $a_i$, $b_i$, $c_{i-1}$ and produces bits $s_i$, $c_i$
The following hold:

| $a_i$ | $b_i$ | $c_i$ | Comment: | Formal definition: |
|---|---|---|---|---|
| 0 | 0 | 0 | The stage “kills” an incoming carry. | “Kill” bit: $k_i = \overline{x_i + y_i}$ |
| 0 | 1 | $c_{i-1}$ | The stage “propagates” an incoming carry | “Propagate” bit: $p_i = x_i \oplus y_i$ |
| 1 | 0 | $c_{i-1}$ | The stage “propagates” an incoming carry | |
| 1 | 1 | 1 | The stage “generates” a carry out | “Generate” bit: $g_i = x_i \bullet y_i$ |

# Binary Addition

| $a_i$ | $b_i$ | $c_i$ | Comment: | Formal definition: |
|---|---|---|---|---|
| 0 | 0 | 0 | The stage “kills” an incoming carry. | “Kill” bit: $k_i = \overline{x_i + y_i}$ |
| 0 | 1 | $c_{i-1}$ | The stage “propagates” an incoming carry | “Propagate” bit: $p_i = x_i \oplus y_i$ |
| 1 | 0 | $c_{i-1}$ | The stage “propagates” an incoming carry | |
| 1 | 1 | 1 | The stage “generates” a carry out | “Generate” bit: $g_i = x_i \bullet y_i$ |

The carry $c_i$ generated by a stage *i* is given by the equation:

$$c_i = g_i + p_i \cdot c_{i-1} = x_i \cdot y_i + \left(x_i \oplus y_i\right) \cdot c_{i-1}$$

This equation can be simplified to:

$$c_i = x_i \cdot y_i + \left(x_i + y_i\right) \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

The “$a_i$” term in the equation being the “alive” bit.
The later form of the equation uses an OR gate instead of an XOR which is a more efficient gate when implemented in CMOS technology. Note that:

$$a_i = \overline{k_i}$$

Where $k_i$ is the “kill” bit defined in the table above.

# Binary addition as a prefix sum problem.

- We define a new operator: “ ∘ ”
- Input is a vector of pairs of ‘propagate’ and ‘generate’ bits:

$$(g_n, p_n)(g_{n-1}, p_{n-1})\ldots(g_0, p_0)$$

- Output is a new vector of pairs:

$$(G_n, P_n)(G_{n-1}, P_{n-1})\ldots(G_0, P_0)$$

- Each pair of the output vector is calculated by the following definition:

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$

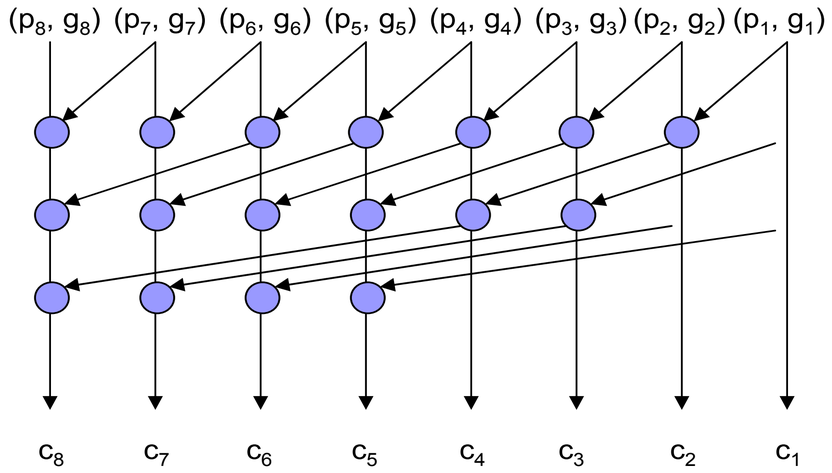
$$Where:$$

$$(G_0, P_0) = (g_0, p_0)$$

$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y)$$

$$with \quad +, \cdot \quad being\ the\ OR, AND\ operations$$

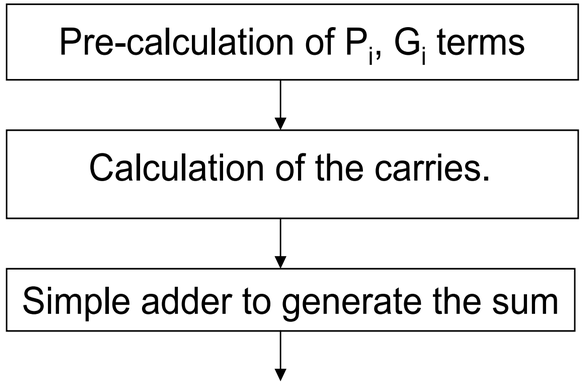
# 1973: Kogge-Stone adder




- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

# Summary

- A parallel prefix adder can be seen as a 3-stage process:




- There exist various architectures for the carry calculation part.
- Trade-offs in these architectures involve the
  - area of the adder
  - its depth
  - the fan-out of the nodes
  - the overall wiring network.

# Announcements & Reminders

- Hw 4 - entire written + programming (Checkpoint #2) is due Wednesday, Apr 28th at 11:59pm
- Lab 8 extension until Tuesday, Apr 27th at 12pm (noon)
- No lab this week



# Worksheet #35: Creating a Circuit for Parallel Prefix Sums

Assume that you have a full adder cell, ⊕, that can be used as a building block for circuits (no need to worry about carry's). Create a circuit that generates the prefix sums for 1, … 6, by adding at most 5 more cells to the sketch shown below, while ensuring that the CPL is at most 3 cells long. Assume that you can duplicate any value (fan-out) to whatever degree you like without any penalty.