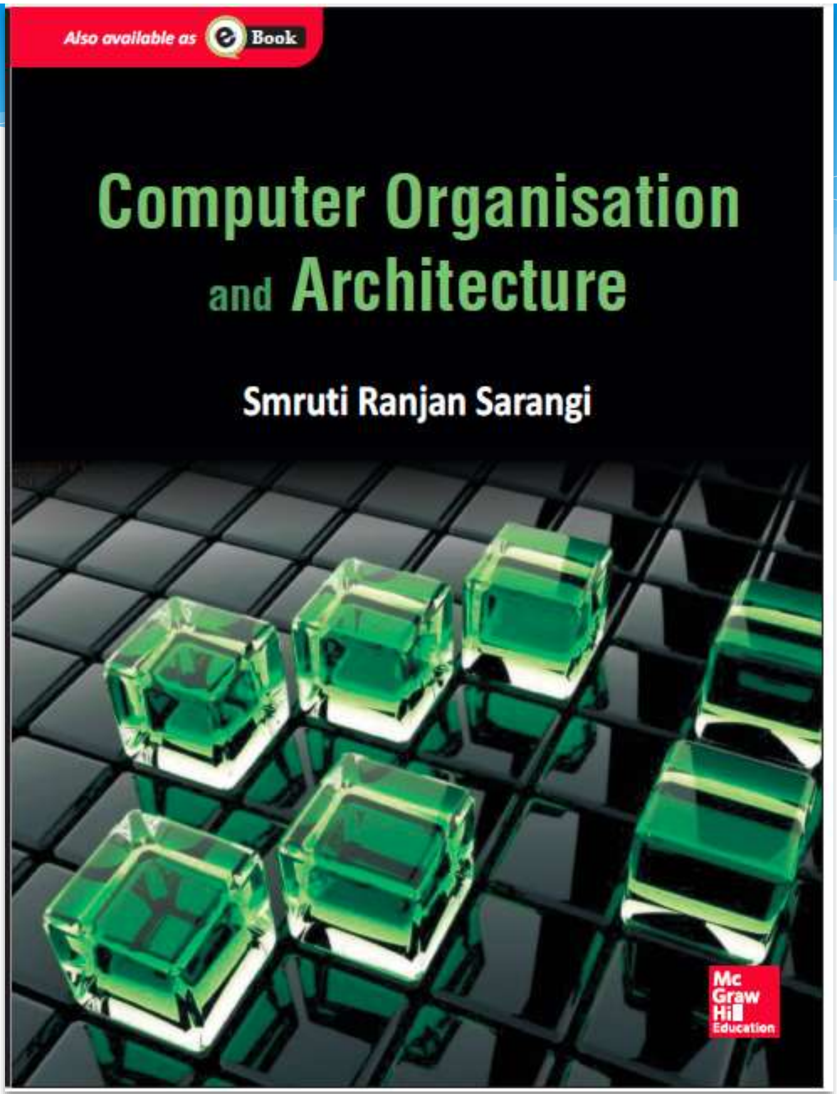
PowerPoint Slides

Computer Organisation and Architecture

Smruti Ranjan Sarangi,
IIT Delhi

Chapter 7 Computer Arithmetic



These slides are meant to be used along with the book: Computer Organisation and Architecture, Smruti Ranjan Sarangi, McGrawHill 2015
Visit: http://www.cse.iitd.ernet.in/~srsarangi/archbooksoft.html

Outline

* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

Adding Two 1 bit Numbers

- Let us add two 1 bit numbers – a and b
  - 0 + 0 = 00
  - 1 + 0 = 01
  - 0 + 1 = 01
  - 1 + 1 = 10
- The lsb of the result is known, as the sum, and the msb is known as the carry

Sum and Carry

$$\begin{array}{r} a \\ + \quad b \\ \hline \end{array}$$

carry sum

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

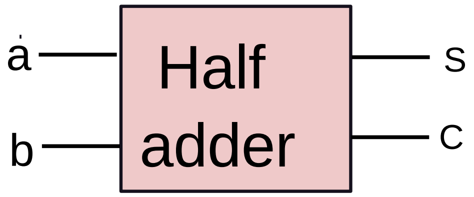
Truth Table

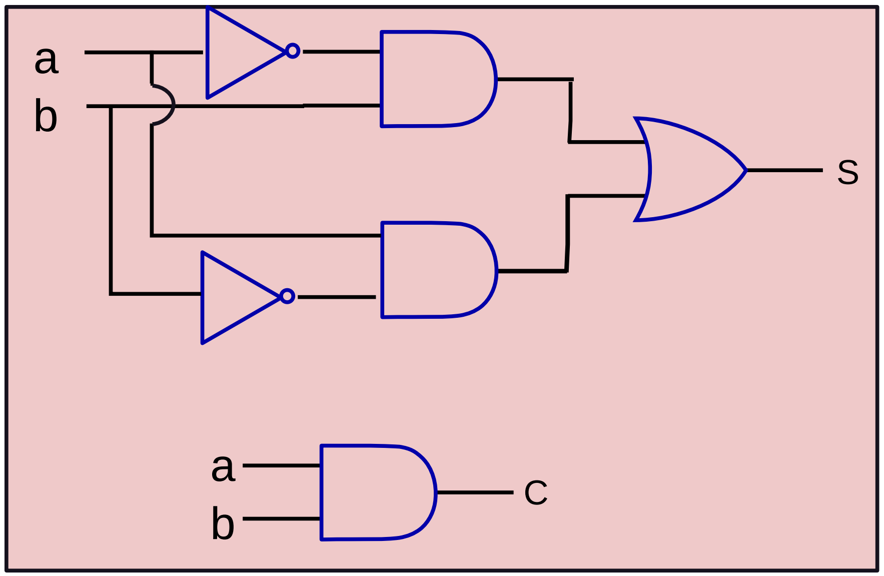
$$S = a \oplus b = \bar{a}.b + a.\bar{b}$$

$$c = a.b$$

Half Adder

* Adds two 1 bit numbers to produce a 2 bit result





Full Adder

Add three 1 bit numbers to produce a 2 bit output

$$a + b + c_{in} = 2 * c_{out} + s$$

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

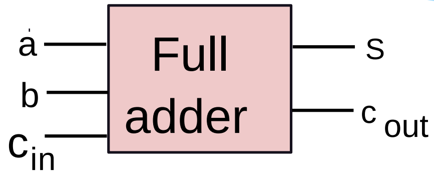
Mc Graw Hill Education

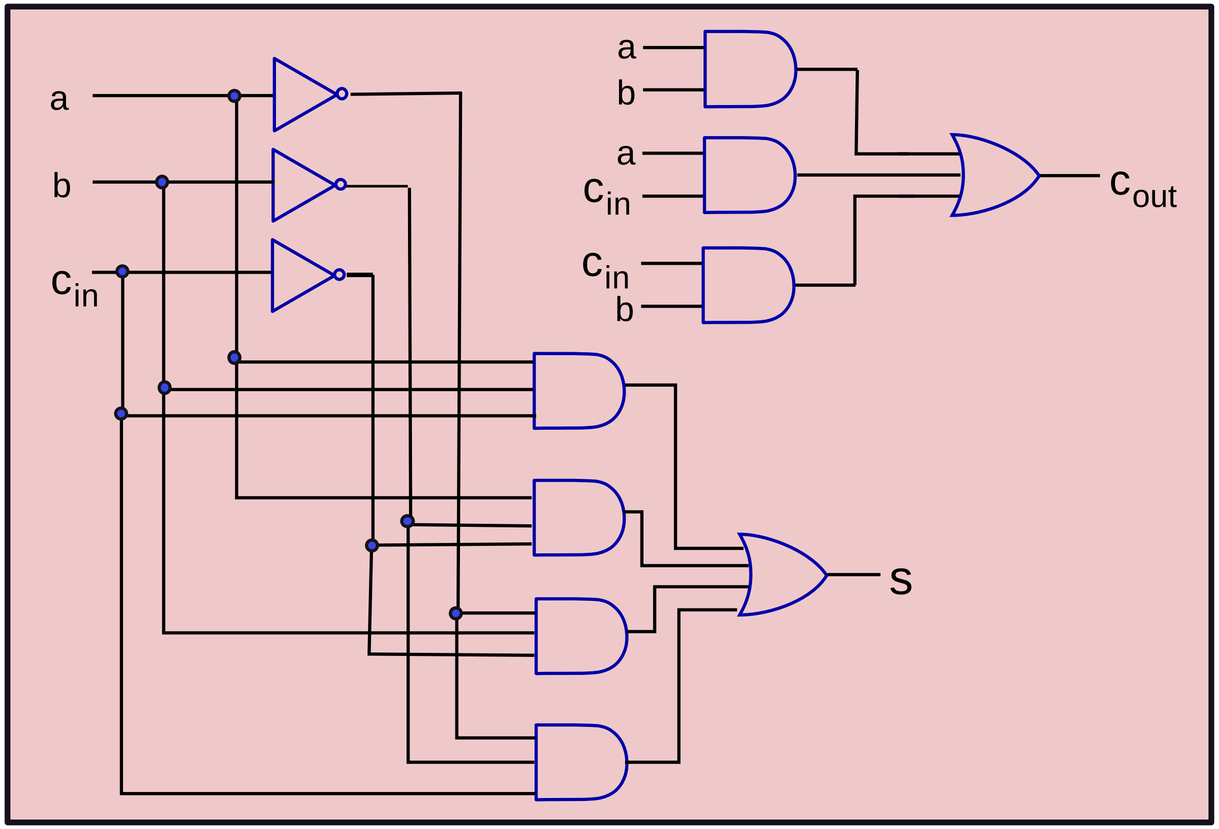
Equations for the Full Adder

$$s = a \oplus b \oplus c_{in}$$
$$= (a.\bar{b} + \bar{a}.\mathrm{b}) \oplus c_{in}$$
$$= (a.\bar{b} + \bar{a}.\mathrm{b}).\overline{c_{in}} + \overline{a.\bar{b} + \bar{a}.\mathrm{b}}.c_{in}$$
$$= a.\bar{b}.\overline{c_{in}} + \bar{a}.\mathrm{b}.\overline{c_{in}} + \overline{(a.\bar{b})}.\overline{(\bar{a}.b)}.c_{in}$$
$$= a.\bar{b}.\overline{c_{in}} + \bar{a}.\mathrm{b}.\overline{c_{in}} + (\bar{a} + b).(a + \bar{b}).c_{in}$$
$$= a.\bar{b}.\overline{c_{in}} + \bar{a}.\mathrm{b}.\overline{c_{in}} + \bar{a}.\bar{b}.c_{in} + a.b.c_{in}$$

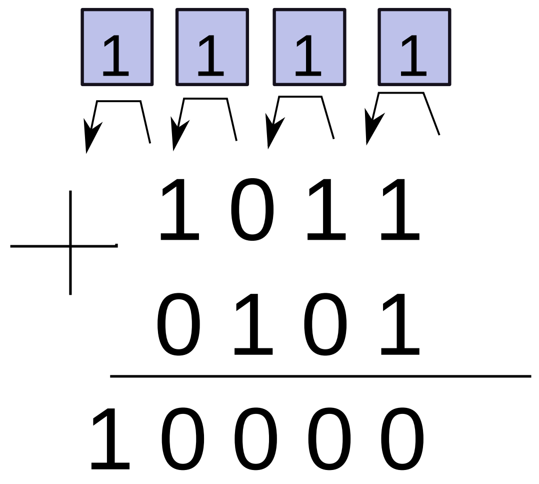
$$c_{out} = a.b + a.c_{in} + b.c_{in}$$

Circuit for the Full Adder





Addition of two *n* bit numbers

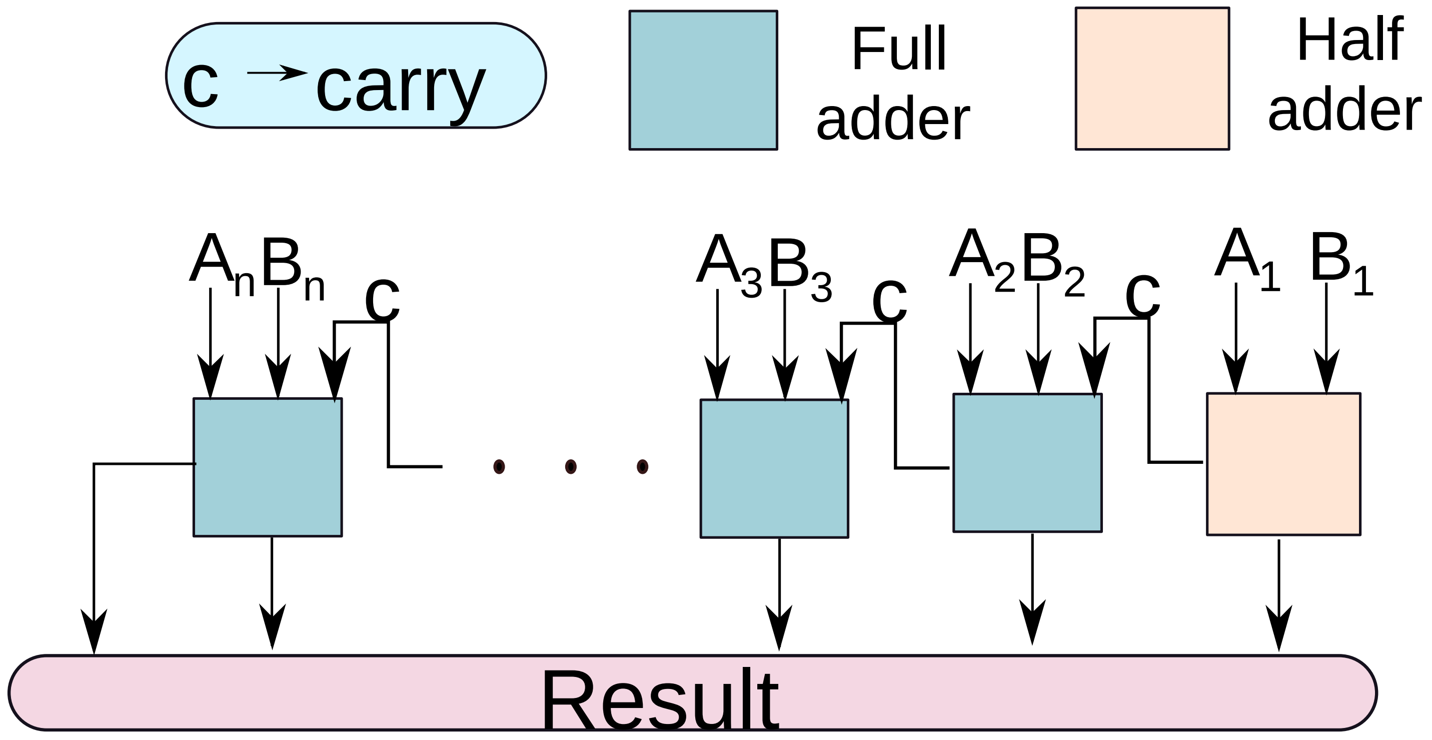


- We start from the lsb
- Add the corresponding pair of bits and the carry in
- Produce a sum bit and a carry out

Observations

- We keep adding pairs of bits, and proceed from the lsb to the msb
- If a carry is generated, we add it to the next pair of bits
- At the last step, if a carry is generated, then it becomes the msb of the result
- The carry effectively ripples through the bits

Ripple Carry Adder



Operation of the Ripple Carry Adder

- Problem : Add A + B
- Number the bits : A_1 to A_n and B_1 to B_n
  - lsb → A_1 and B_1
  - msb → A_n and B_n
- Use a half adder to add A_1 and B_1
- Send the carry(c) to a full adder that adds : $A_2 + B_2 + c$
- Proceed in a similar manner till the msb

McGraw Hill Education

How long does the Ripple Carry Adder take ?

- Time :
  - Time of half adder : t_h
  - Time of full adder : t_f
  - Total Time : $t_h + (n-1)t_f$

McGraw Hill Education

Asymptotic Time Complexity

- Most of the time, we are primarily interested in the order of the function
- For example : we are only interested in the n^2 term in $(2n^2 + 3n + 4)$
- We do not care about the constants, and terms with smaller exponents
  - $3n$ and 4
- We can thus say that :
  - $2n^2 + 3n + 4$ is order of (n^2)

The O notation

- Formally :
  - We say that: f(n) = O(g(n))
  - if, $|f(n)| \le c|g(n)|$, for all $n > n_0$. Here *c* is a positive constant.
- In simple terms:
  - Beyond a certain *n* , g(n) is greater-than-equal to a certain constant times f(n)
    - *For example, beyond 15, $(n^2 + 10n + 16) \le 2n^2$*

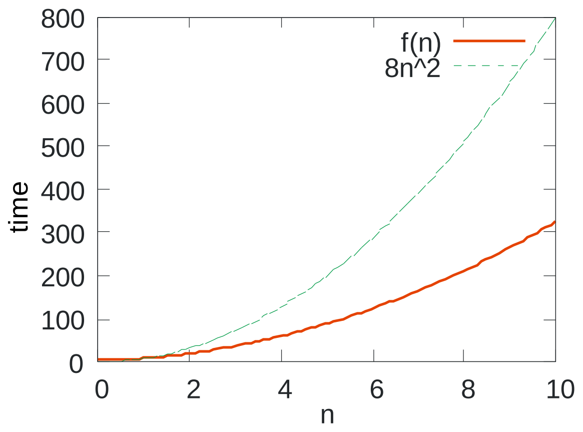
Example of the big O Notation

$f(n) = 3n^2 + 2n + 3$. *Find its asymptotic time complexity.*

Answer:

$$\begin{aligned} f(n) &= 3n^2 + 2n + 3 \\ &\leq 3n^2 + 2n^2 + 3n^2 \quad (n > 1) \\ &\leq 8(n^2) \end{aligned}$$

Hence, $f(n) = O(n^2)$.



$8n^2$ *is a strict upper bound on* $f(n)$ *as shown in the figure.*

Big O Notation - II

Example:
$f(n) = 0.00001n^{100} + 10000n^{99} + 234344$. Find its asymptotic time complexity.

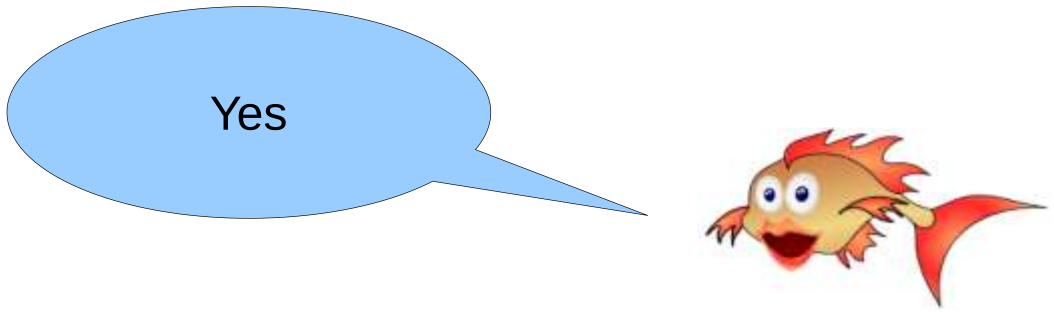
Answer: $f(n) = O(n^{100})$

- We shall use the asymptotic time complexity metric (big O notation) to characterize the time taken by different adders

Ripple Carry Adders and Beyond

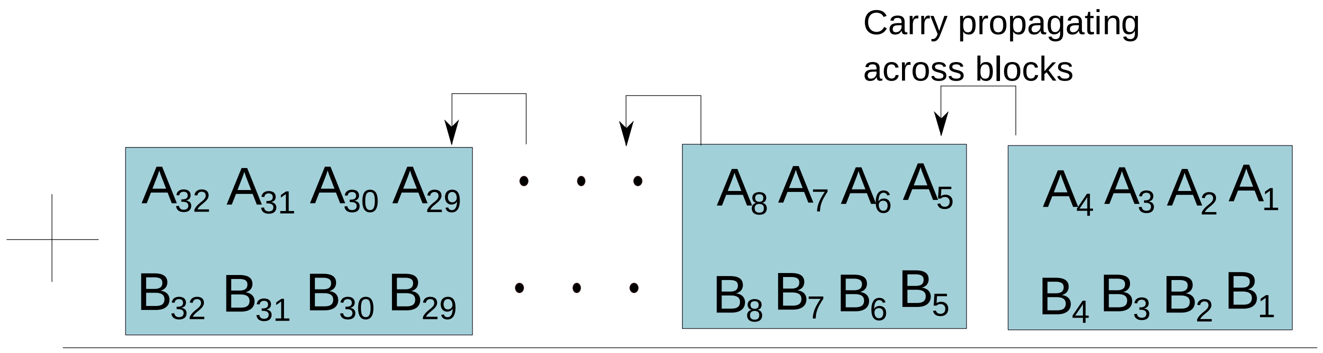
- Time complexity of a ripple carry adder :
  - O(n)

- Can we do better than O(n) ?



Carry Select Adder O(√n) time

- Group bits into blocks of size (k)
- If we are adding two 32 bit numbers A and B, and k = 4, then the blocks are :


Carry propagating across blocks
$A_{32}\ A_{31}\ A_{30}\ A_{29}$
$B_{32}\ B_{31}\ B_{30}\ B_{29}$
$A_8\ A_7\ A_6\ A_5$
$B_8\ B_7\ B_6\ B_5$
$A_4\ A_3\ A_2\ A_1$
$B_4\ B_3\ B_2\ B_1$
+

- Produce the result of each block with a small ripple carry adder

Carry Select Adder - II

- In this case, the carry propagates across blocks
- Time complexity is O(n)

Idea :

- Add the numbers in each block in parallel
- Stage I : For each block, produce two results
  - Assuming an input carry of 0
  - Assuming an input carry of 1

Carry Select Adder – Stage II

- For each block we have two results available
- Result → (k sum bits), and 1 carry out bit
- Stage II
  - Start at the least significant block
    - The input carry is 0
    - Choose the appropriate result from stage I
  - We now know the input carry for the second block
    - Choose the appropriate result
    - Result contains the input carry for the third block

Carry Select Adder – Stage II

- Given the result of the second block
  - Compute the carry in for the third block
  - Choose the appropriate result
- Proceed till the last block
- At the last block (most significant positions)
  - Choose the correct result
  - The carry out value, is equal to the carry out of the entire computation.

How much time did we take ?

- Our block size is k
  - Stage I takes k units of time
- There are n/k blocks
  - Stage II takes (n/k) units of time
- Total time : (k + n/k)

$$\frac{\partial\,(k+n/k)}{\partial k} = 0$$

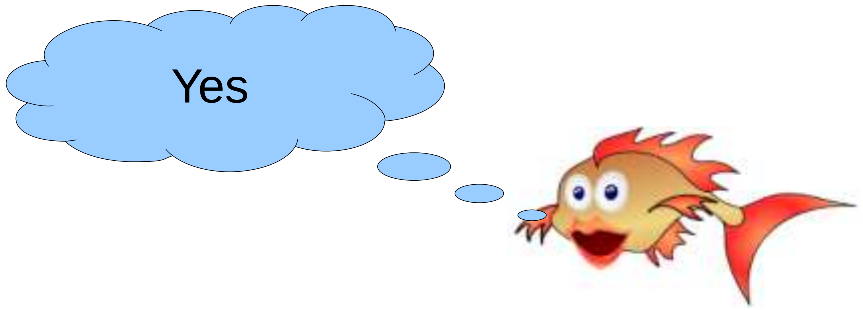
$$\Longrightarrow 1 - \frac{n}{k^2} = 0$$

$$\Longrightarrow k = \sqrt{n}$$

Time Complexity of the Carry Select Adder

- T = O(√n + √n) = O(√n)
- Thus, we have a √n time adder

Can we do better ?



Carry Lookahead Adder (O(log n))

- The main problem in addition is the carry
- If we have a mechanism to compute the carry quickly, we are done
- Let us thus focus on computing the carry without actually performing an addition

Generate and Propagate Functions

- Let us consider two corresponding bits of A and B
  - A_i and B_i
- Generate function : A new carry is generated (C_{out} = 1)
- Propagate function : $C_{out} = C_{in}$
- Generate and Propagate Functions are :

$$g_i = A_i . B_i$$
$$p_i = A_i \oplus B_i$$

Using the G and P Functions

- If we have the generate and propagate values for a bit pair, we can determine the carry out

$$C_{out} = g_i + p_i . C_{in}$$

Example

Example:

Let A_i = 0, B_i = 1. Let the input carry be C_{in}. Compute g_i, p_i, and C_{out}.

Answer:

$$g_i = A_i . B_i = 0.1 = 0$$
$$p_i = A_i \oplus B_i = 0 \oplus 1 = 1$$

$$C_{out} = g_i + p_i . C_{in} = C_{in}$$

G and P for Multi-bit Systems

- $C_{out}^{i} \rightarrow$ output carry for ith bit pair
- $C_{in}^{i} \rightarrow$ input carry for ith bit pair
- $g_i \rightarrow$ generate value for ith bit pair
- $p_i \rightarrow$ propagate value for ith bit pair

G and P for Multibit Systems - II

$$\boxed{C_{out}^{1} = g_1 + p_1.C_{in}^{1}}$$

$$\begin{aligned} C_{out}^{2} &= g_2 + p_2.C_{out}^{1} \\ &= g_2 + p_2.\left(g_1 + p_1.C_{in}^{1}\right) \\ &= (g_2 + p_2.g_1) + p_2.p_1.C_{in}^{1} \end{aligned}$$

$$\begin{aligned} C_{out}^{3} &= g_3 + p_3.C_{out}^{2} \\ &= g_3 + p_3.\left((g_2 + p_2.g_1) + p_2.p_1.C_{in}^{1}\right) \\ &= (g_3 + p_3.g_2 + p_3.p_2.g_1) + p_3.p_2.p_1.C_{in}^{1} \end{aligned}$$

G and P for multibit Systems - III

$$C_{out}^{4} = g_4 + p_4 . C_{out}^{3}$$

$$= g_4 + p_4 . \left((g_3 + p_3 . g_2 + p_3 . p_2 . g_1) + p_3 . p_2 . p_1 . C_{in}^{1} \right)$$

$$= (g_4 + p_4 . g_3 + p_4 . p_3 . g_2 + p_4 . p_3 . p_2 . g_1) + p_4 . p_3 . p_2 . p_1 . C_{in}^{1}$$

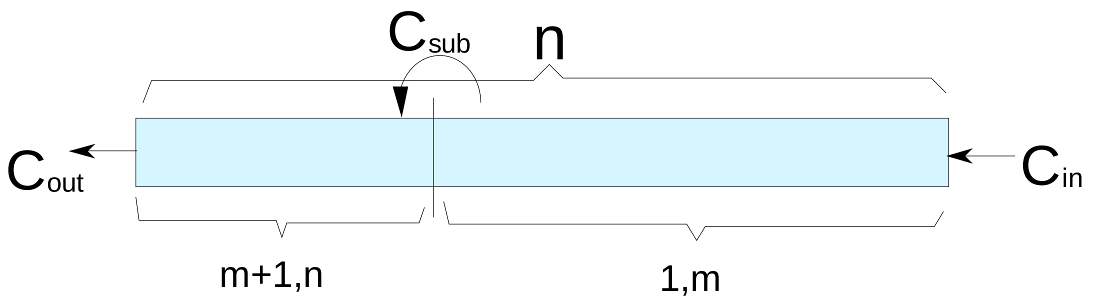
Mc Graw Hill Education

Patterns

1 bit	$C_{out}^{1} = \underbrace{g_1}_{G_1} + \underbrace{p_1}_{P_1} . C_{in}^{1}$
2 bit	$C_{out}^{2} = \underbrace{g_2 + p_2 . g_1}_{G_2} + \underbrace{p_2 . p_1}_{P_2} . C_{in}^{1}$
3 bit	$C_{out}^{3} = \underbrace{g_3 + p_3 . g_2 + p_3 . p_2 . g_1}_{G_3} + \underbrace{p_3 . p_2 . p_1}_{P_3} . C_{in}^{1}$
4 bit	$C_{out}^{4} = \underbrace{g_4 + p_4 . g_3 + p_4 . p_3 . g_2 + p_4 . p_3 . p_2 . g_1}_{G_4} + \underbrace{p_4 . p_3 . p_2 . p_1}_{P_4} . C_{in}^{1}$
n bit	$C_{out}^{n} = G_n + P_n . C_{in}^{1}$

Computing G and P Quickly

- Let us divide a block of *n* bits into two parts



- Let the carry out and carry in be : C_{out} and C_{in}
- We want to find the relationship between
  - $G_{1,n}$, $P_{1,n}$ and ($G_{m+1,n}$, $G_{1,m}$, $P_{m+1,n}$, $P_{1,m}$)

Computing G and P Quickly - II

$$C_{out} = G_{m+1,n} + P_{m+1,n}.C_{sub}$$
$$= G_{m+1,n}$$
$$+ P_{m+1,n}.\left(G_{1,m} + P_{1,m}.C_{in}\right)$$
$$= \underbrace{G_{m+1,n} + P_{m+1,n}.G_{1,m}}_{G_{1,n}} + \underbrace{P_{m+1,n}.P_{1,m}}_{P_{1,n}} .C_{in}$$

$$C_{out} = G_{1,n} + P_{1,n}.C_{in}$$

$$\boxed{G_{1,n} = G_{m+1,n} + P_{m+1,n}.G_{1,m}}$$

$$\boxed{P_{1,n} = P_{m+1,n}.P_{1,m}}$$

Insight into Computing G and P quickly

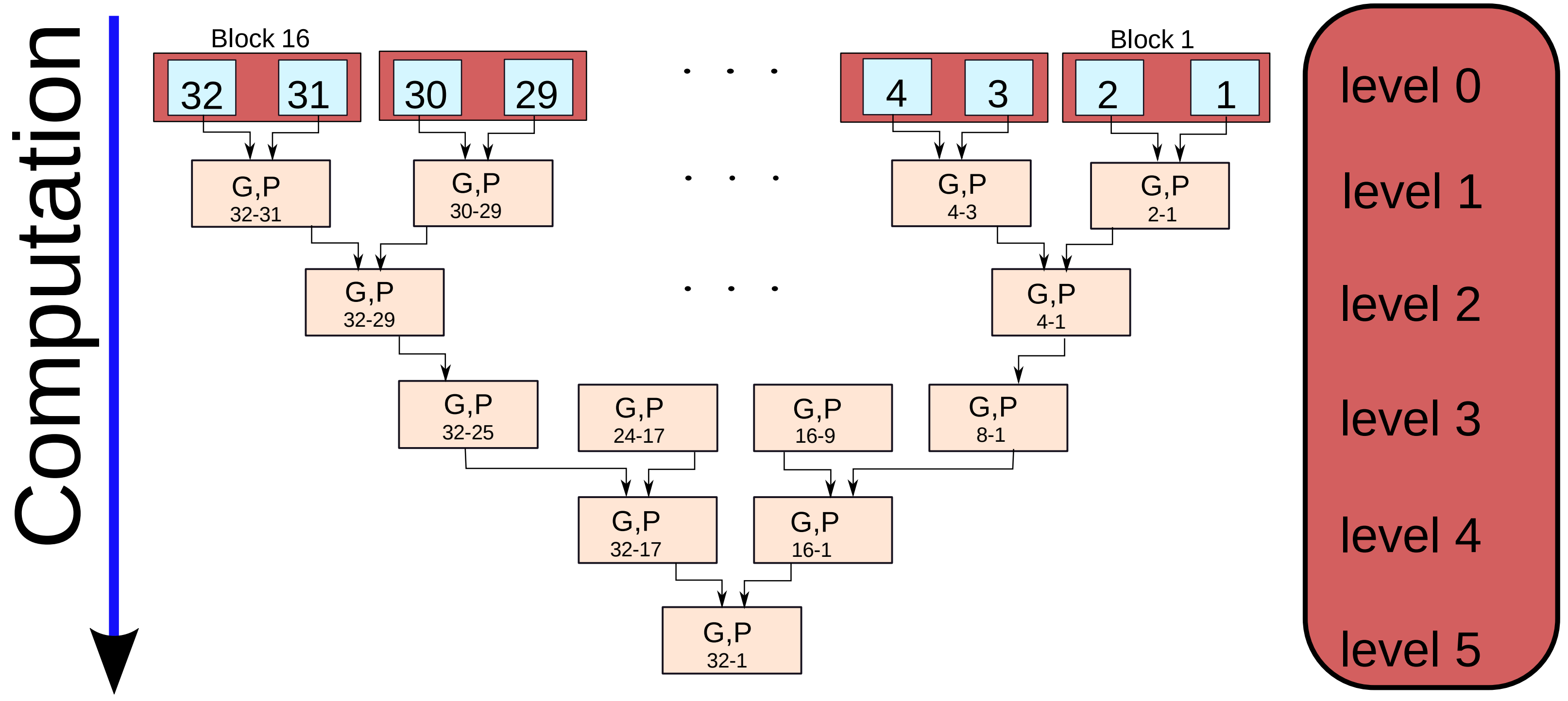
- Insight :
  - We can compute G and P for a large block
    - By first computing G and P for smaller sub-blocks
    - And, then combining the solutions to find the value of G and P for the larger block
  - Fast algorithm to compute G and P
    - Use divide-and-conquer
    - Compute G and P functions in O (log (n)) time

Mc Graw Hill Education

Carry Lookahead Adder – Stage I

- Compute G and P functions for all the blocks
- Combine the solutions to find G and P functions for sets of 2 blocks
- Combine the solutions fo find G and P functions for sets of 4 blocks
- ….
- ….
- Find the G and P functions for a block of size : 32 bits

Carry Lookahead Adder – Stage I

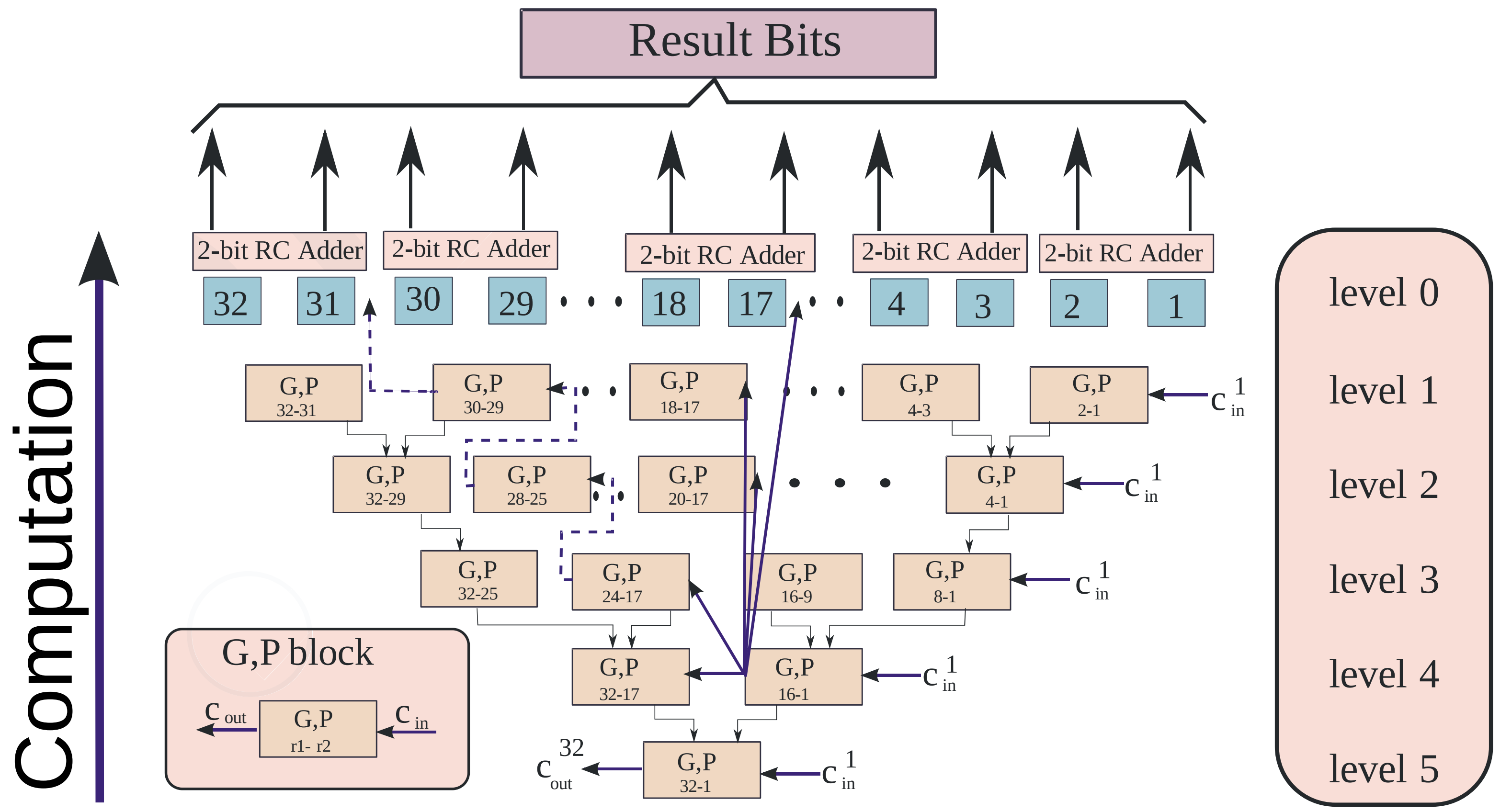


CLA Adder – Stage I

- Compute G, P for increasing sizes of blocks in a tree like fashion
- Time taken :
  - Total : log(n) levels
  - Time per level : O(1)
  - Total Time : O(log(n))

Mc Graw Hill Education

CLA Adder – Stage II



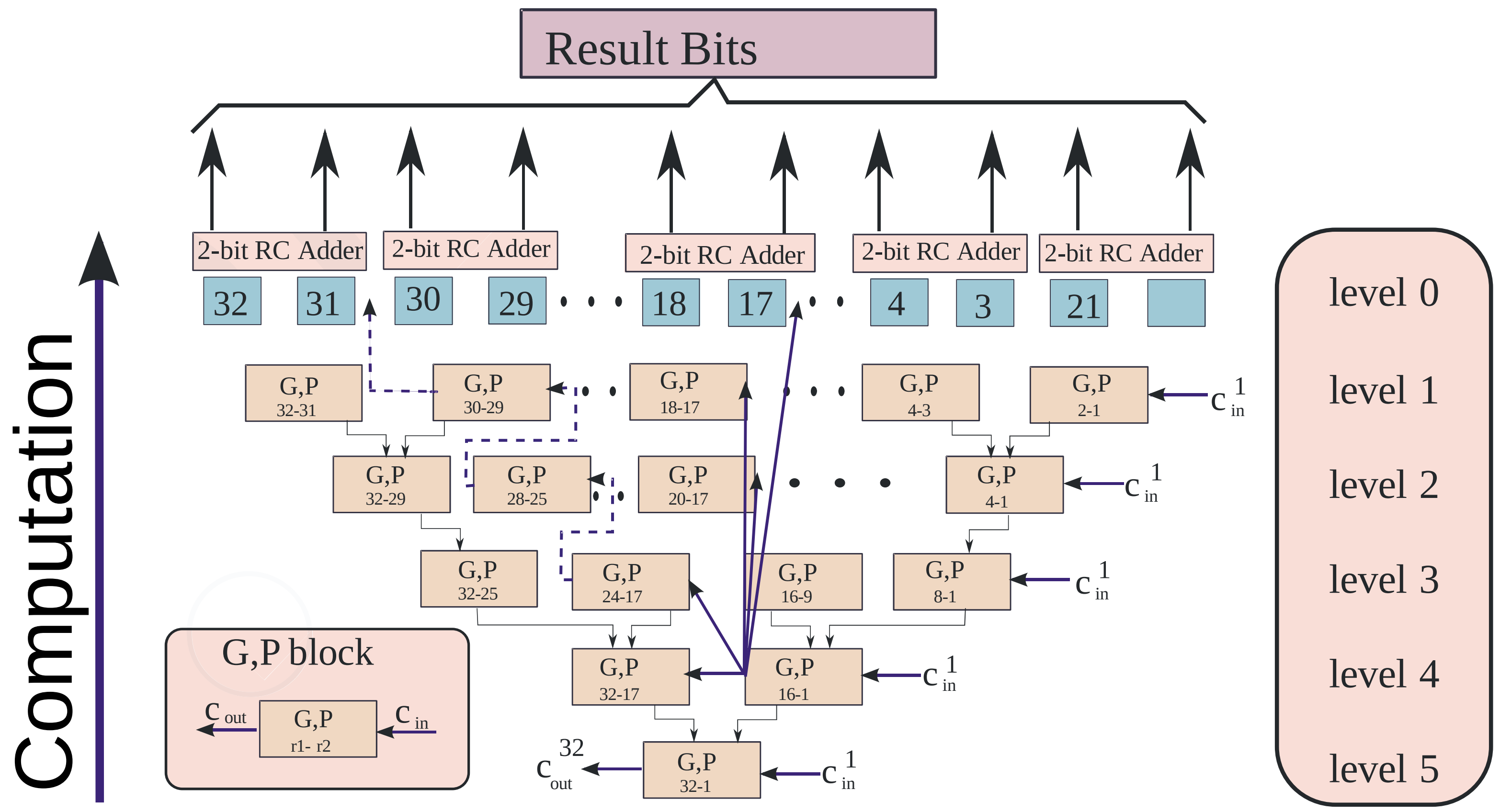
Connection of the G,P Blocks

- Each G,P block represents a range of bits (r2, r1) (r2 > r1)
  - The (r2, r1) G,P block is connected to all the blocks of the form (r3, r2+1)
  - The carry out of one block is an input to all the blocks that it is connected with
- Each block is connected to another block at the same level, and to blocks at lower levels

Operation of CLA – Stage II

- We start at the leftmost blocks in each level
  - We feed an input carry value of C_{in}^1
  - Each such block computes the output carry, and sends it to the all the blocks that it is connected to
- Each connected block
  - Computes the output carry
  - Sends it to all the blocks that it is connected to
- The carry propagates to all the 2 bit RC adders

CLA Adder – Stage II



Time Complexity

- In a similar manner, the carry propagates to all the RC adders at the zeroth level
- Each of them compute the correct result
- Time taken by Stage II :
  - Time taken for a carry to propagate from the (16,1) node to the RC adders
  - O(log(n))
- Total time : O(log(n) + log(n)) = O(log(n))

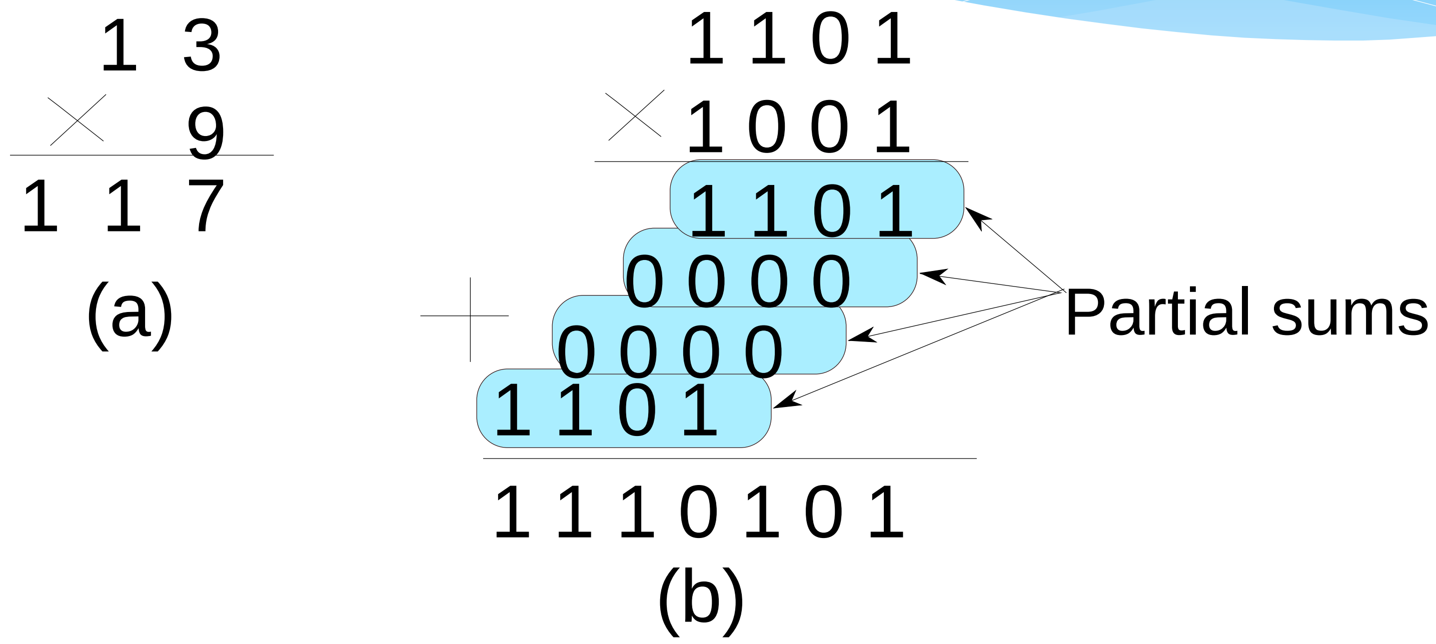
Time complexities of different adders:

- Ripple Carry Adder: $O(n)$
- Carry Select Adder: $O(\sqrt{n})$
- Carry Lookahead Adder: $O(\log(n))$

Outline

* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

Multiplicands



- 13 → Multiplicand
- 9 → Multiplier
- 117 → Product

Basic Multiplication

- Consider the lsb of the multiplier
  - If it is 1, write the value of the multiplicand
  - If it is 0, write 0
- For the next bit of the multiplier
  - If it is 1, write the value of the multiplicand shifted by 1 position to the left
  - If it is 0, write 0
- Keep going ….

Definitions

Partial sum: It is equal to the value of the multiplicand left shifted by a certain number of bits, or it is equal to 0.
Partial product: It is the sum of a set of partial sums.

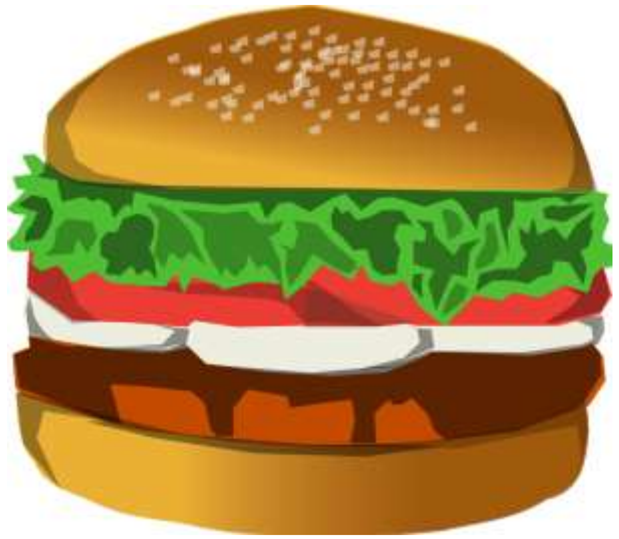
- If the multiplier has m bits, and the multiplicand has n bits
  - The product requires (m+n) bits

Mc Graw Hill Education

Multiplying 32 bit numbers

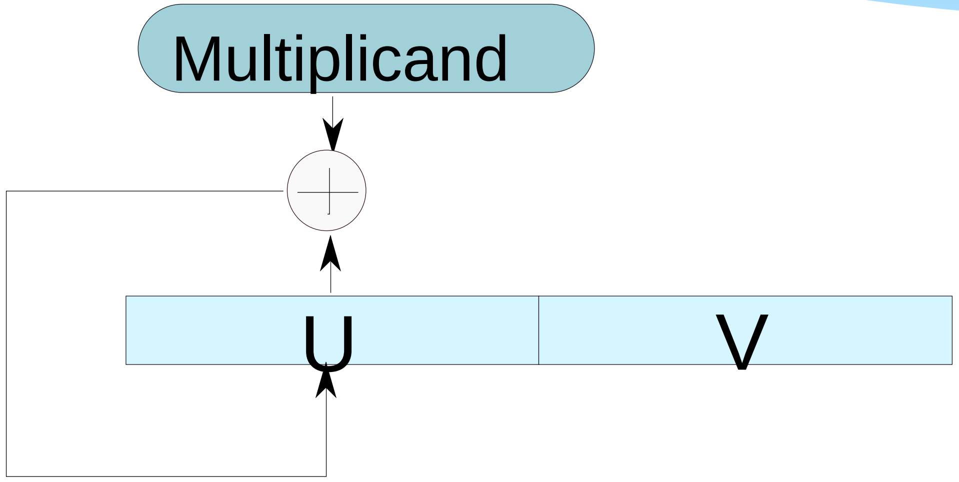
- Let us design an iterative multiplier that multiplies two 32 bit signed values to produce a 64 bit result
- What did we prove before
  - Multiplying two signed 32 bit numbers, and saving the result as a 32 bit number is the same as
  - Multiplying two unsigned 32 bit numbers (assuming no overflows)
- We did not prove any result regarding saving the result as a 64 bit number

Class Work



Theorem: A signed *n* bit number $A = A_{1\ \dots n-1} - A_n 2^{n-1}$. A_i is the ith bit in A's 2's complement based binary representation (the first bit is the LSB). $A_{1\dots n-1}$ is a binary number containing the first *n-1* digits of *A*'s binary 2's complement representation.

Iterative Multiplier



- Multiplicand (N), Multiplier (M), Product(P) = MN
- U is a 33 bit register and V is a 32 bit register
- beginning : V contains the multiplier, U = 0
- UV is one register for the purpose of shifting

Algorithm

Algorithm 1: Algorithm to multiply two 32 bit numbers and produce a 64 bit result

Data: Multiplier in *V* , *U* = 0, Multiplicand in *N*
Result: The lower 64 bits of *UV* contains the product

```
i ← 0
for i < 32 do
    i ← i + 1
    if LSB of V is 1 then
        if i < 32 then
            U ← U + N
        end
        else
            U ← U − N
        end
    end
    UV ← UV >> 1 (arithmetic right shift)
end
```

Example

Multiplicand (N)	0010	2
Multiplier (M)	0011	3

	U	V
beginning:	00000	0011

Step		U	V	
1	before shift:	00010	0011	1 $\rightarrow$ add 2
	after shift:	00001	0001	
2	before shift:	00011	0001	1 $\rightarrow$ add 2
	after shift:	00001	1000	
3	before shift:	00001	1000	0 $\rightarrow$ --
	after shift:	00000	1100	
4	before shift:	00000	1100	0 $\rightarrow$ --
	after shift:	00000	0110	

Product(P)	0110	6

3 * (-2)

Multiplicand (N)	0011	3
Multiplier (M)	1110	-2

		U	V
	beginning:	00000	1110
1	before shift:	00000	1110
	after shift:	00000	0111
2	before shift:	00011	0111
	after shift:	00001	1011
3	before shift:	00100	1011
	after shift:	00010	0101
4	before shift:	11111	0101
	after shift:	11111	1010

0 $\Rightarrow$ --

1 $\Rightarrow$ add 3

1 $\Rightarrow$ add 3

1 $\Rightarrow$ sub 3

Product(P)	1010	-6

Operation of the Algorithm

- Take a look at the lsb of V
  - If it is 0 → do nothing
  - If it is 1 → Add N (multiplicand) to U
- Right shift
  - Right shifting the partial product is the same as left shifting the multiplicand, which
  - Needs to be done in every step
- Last step is different

The Last Step …

- In the last step
  - lsb of V = msb of M (multiplier)
  - If it is 0 → do nothing
- If it is 1
  - Multiplier is negative
  - Recall : $A = A_{1 .. n-1} - 2^{n-1}A_n$
  - Hence, we need to subtract the multiplicand if the msb of the multiplier is 1

Time Complexity

- There are *n* loops
  - Each loop takes log(n) time
  - Total time : O(n log(n))

Booth Multiplier

- We can make our iterative multiplier faster
- If there are a continuous sequence of 0s in the multiplier
  - do nothing
- If there is a continous sequnce of 1s
  - do something smart

$$M = \sum_{k=i}^{k=j} 2^k = 2^{j+1} - 2^i$$

For a Sequence of 1s

- Sequence of 1s from position i to j
  - Perform (j – i + 1) additions

$$\boxed{A = \sum_{k=i}^{k=j} 2^k}$$

- <u>New method</u>
  - Subtract the multiplicand when we scan bit i (! count starts from 0)
  - Keep shifting the partial product
  - Add the multiplicand(N), when we scan bit (j+1)
  - This process, effectively adds $(2^{j+1} – 2^i) * N$ to the partial product
  - Exactly, what we wanted to do …

McGraw Hill Education

Operation of the Algorithm

- Consider bit pairs in the multiplier
  - (current bit, previous bit)
  - Take actions based on the bit pair
  - Action table

(current value, previous value)	Action
0,0	-
1,0	subtract multiplicand from U
1,1	-
0,1	add multiplicand to U

Booth's Algorithm

Algorithm 2: Booth's Algorithm to multiply two 32 bit numbers to produce a 64 bit result

Data: Multiplier in *V* , *U* = 0, Multiplicand in *N*
Result: The lower 64 bits of *UV* contain the result

```
i ← 0
prevBit ← 0
for i < 32 do
    i ← i + 1
    currBit ← LSB of V
    if (currBit,prevBit) = (1,0) then
        U ← U − N
    end
    else if (currBit,prevBit) = (0,1) then
        U ← U + N
    end
    prevBit ← currBit
    UV ← UV >> 1 (arithmetic right shift)
end
```

Outline of a Proof

- Multiplier (M) is positive
  - msb = 0
  - Divide the multiplier into a sequence of continuous 0s and 1s
    - 01100110111000 → 0,11, 00, 11, 0, 111, 000
  - For sequence of 0s
    - Both the algorithms (iterative, Booth) do not add the multiplicand
    - For a run of 1s (length k)
      - The iterative algorithm performs *k* additions
      - Booth's algorithm does one addition, and one subtraction.
      - The result is the same

Outline of a Proof - II

- <u>Negative multipliers</u>
  - msb = 1
- $M = -2^{n-1} + \Sigma_{(i=1 \text{ to } n-1)} M_i 2^{n-1} = -2^{n-1} + M'$
  - $M' = \Sigma_{(i=1 \text{ to } n-1)} M_i 2^{n-1}$
- <u>Consider two cases</u>
  - The two msb bits of M are 10
  - The two msb bits of M are 11

Outline of a Proof - III

- Case 10
  - Till the (n-1)th iteration both the algorithms have no idea if the multiplier is equal to M or M'
  - At the end of the (n-1)th iteration, the partial product is:
    - Iterative algorithm : M'N
    - Booth's algorithm : M'N
      - If we were multiplying (M' * N), no action would have been taken in the last iteration. The two msb bits would have been 00. There is no way to differentiate this case from that of computing MN in the first (n-1) iterations.

Outline of a Proof - IV

- Last step
  - Iterative algorithm :
    - Subtract $2^{n-1}N$ from U
  - Booth's algorithm
    - The last two bits are 10 ($0 \rightarrow 1$ transition)
    - Subtract $2^{n-1}N$ from U
  - Both the algorithms compute :
    - $MN = M'N - 2^{n-1}N$
    - in the last iteration

Outline of a Proof - V

- Case 11
- Suppose we were multiplying M' with N
  - Since (M' > 0), the Booth multiplier will correctly compute the product as M'N
  - The two msb bits of M' are (01)
  - In the last iteration (currBit, prevBit) is 01
  - We would thus add $2^{n-1}N$ in the Booth's algorithm to the partial product in the last iteration
  - The value of the partial product at the end of the (n-1)th iteration is thus :
    - $M'N - 2^{n-1}N$

Outline of a Proof - VI

- When we multiply M with N
  - In the (n-1)th iteration, the value of the partial product is : $M'N - 2^{n-1}N$
  - Because, we have no way of knowing if the multiplier is M or M' at the end of the (n-1)th iteration
  - In the last iteration the msb bits are 11
    - no action is taken
  - Final product : $M'N - 2^{n-1}N = MN$ (correct)

Multiplicand (N)	00011	3
Multiplier (M)	0010	2

		U	V
	beginning:	00000	0010
1	before shift:	00000	0010
	after shift:	00000	0001
2	before shift:	11101	0001
	after shift:	11110	1000
3	before shift:	00001	1000
	after shift:	00000	1100
4	before shift:	00000	1100
	after shift:	00000	0110

Product(P)	0110	6

00 $\Rightarrow$ --

10 $\Rightarrow$ add -3

01 $\Rightarrow$ add 3

00 $\Rightarrow$ --

Multiplicand (N)	00011	3
Multiplier (M)	1110	-2

	U	V
beginning:	00000	1110

Step		U	V	
1	before shift:	00000	1110	00 ➡ --
	after shift:	00000	0111	
2	before shift:	11101	0111	10 ➡ add -3
	after shift:	11110	1011	
3	before shift:	11110	1011	11 ➡ --
	after shift:	11111	0101	
4	before shift:	11111	0101	11 ➡ --
	after shift:	11111	1010	

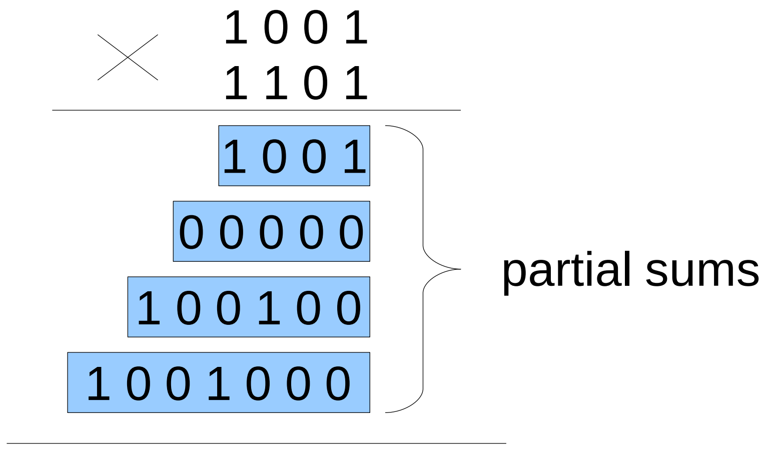
Product(P)	1010	-6

Time Complexity

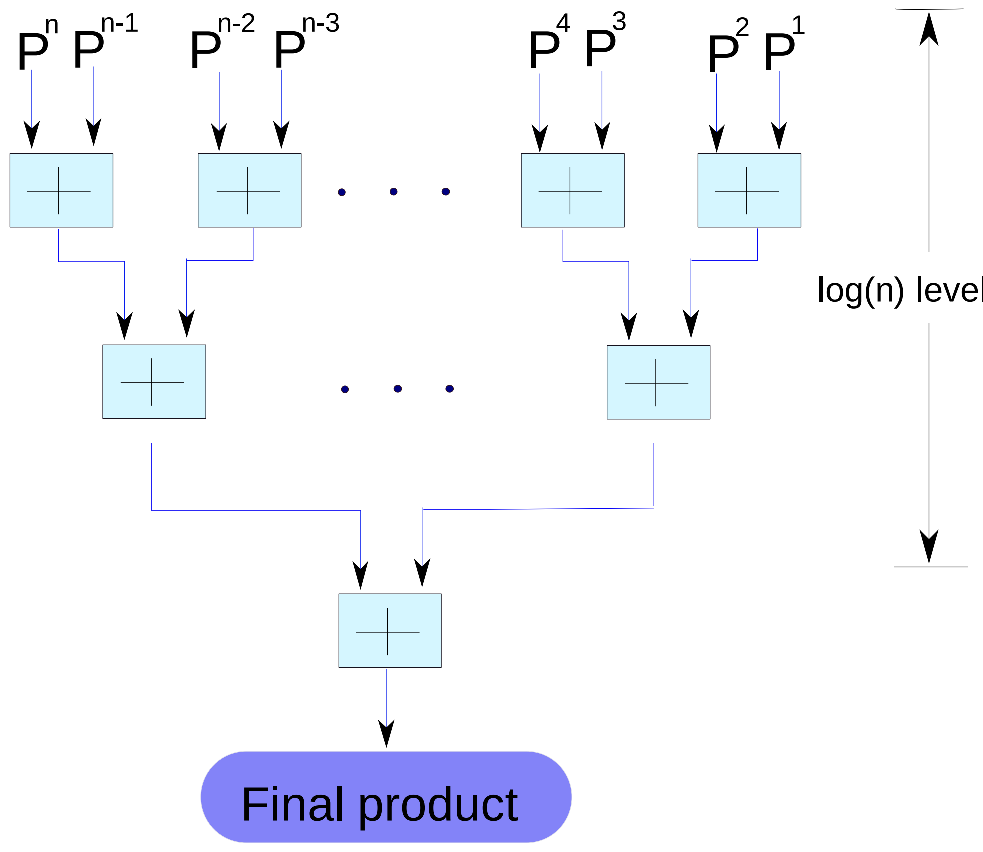
- O(n log(n))
- Worst case input
  - Multiplier = 10101010... 10

$O(\log(n)^2)$ Multiplier

- Consider an *n* bit multiplier and multiplicand
- Let us create *n* partial sums



Tree Based Adder for Partial Sums

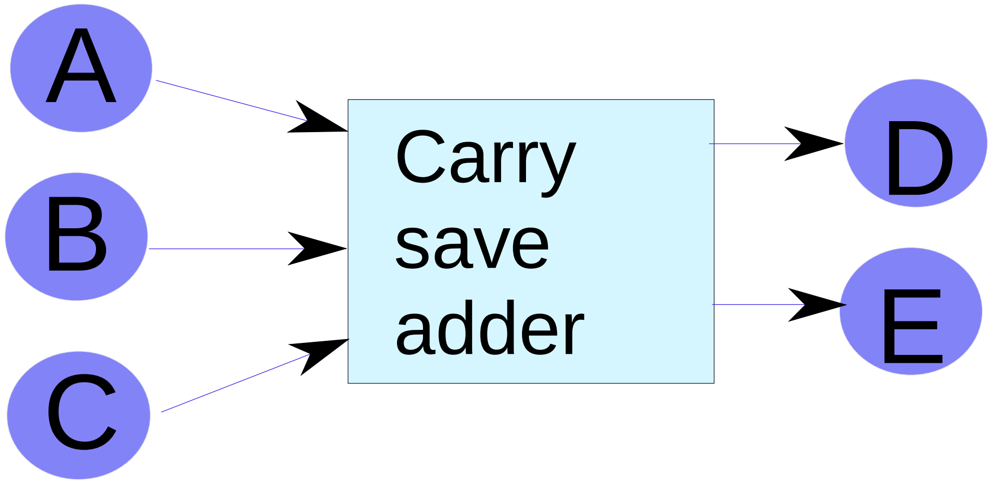


Time Complexity

- There are log(n) levels
- Each level takes
  - Maximum log(2n) time
    - Adds two 2n bit numbers
- Total time :
  - $O(\log(n) * \log(n)) = O(\log(n)^2)$

McGraw Hill Education

Carry Save Adder



- A + B + C = D + E
- Takes three numbers, and produces two numbers

1 bit CSA Adder

- Add three bits – a, b, and c
  - such that $a + b + c = 2d + e$
  - d and e are also single bits
- We can conveniently set
  - e to the sum bit
  - d to the carry bit

n-bit CSA Adder

$$A + B + C = \sum_{i=1}^{n} A_i 2^{i-1} + \sum_{i=1}^{n} B_i 2^{i-1} + \sum_{i=1}^{n} C_i 2^{i-1}$$

$$= \sum_{i=1}^{n} (A_i + B_i + C_i) 2^{i-1}$$

$$= \sum_{i=1}^{n} (2D_i + E_i) 2^{i-1}$$

$$= \underbrace{\sum_{i=1}^{n} D_i 2^{i}}_{D} + \underbrace{\sum_{i=1}^{n} E_i 2^{i-1}}_{E}$$

$$= D + E$$

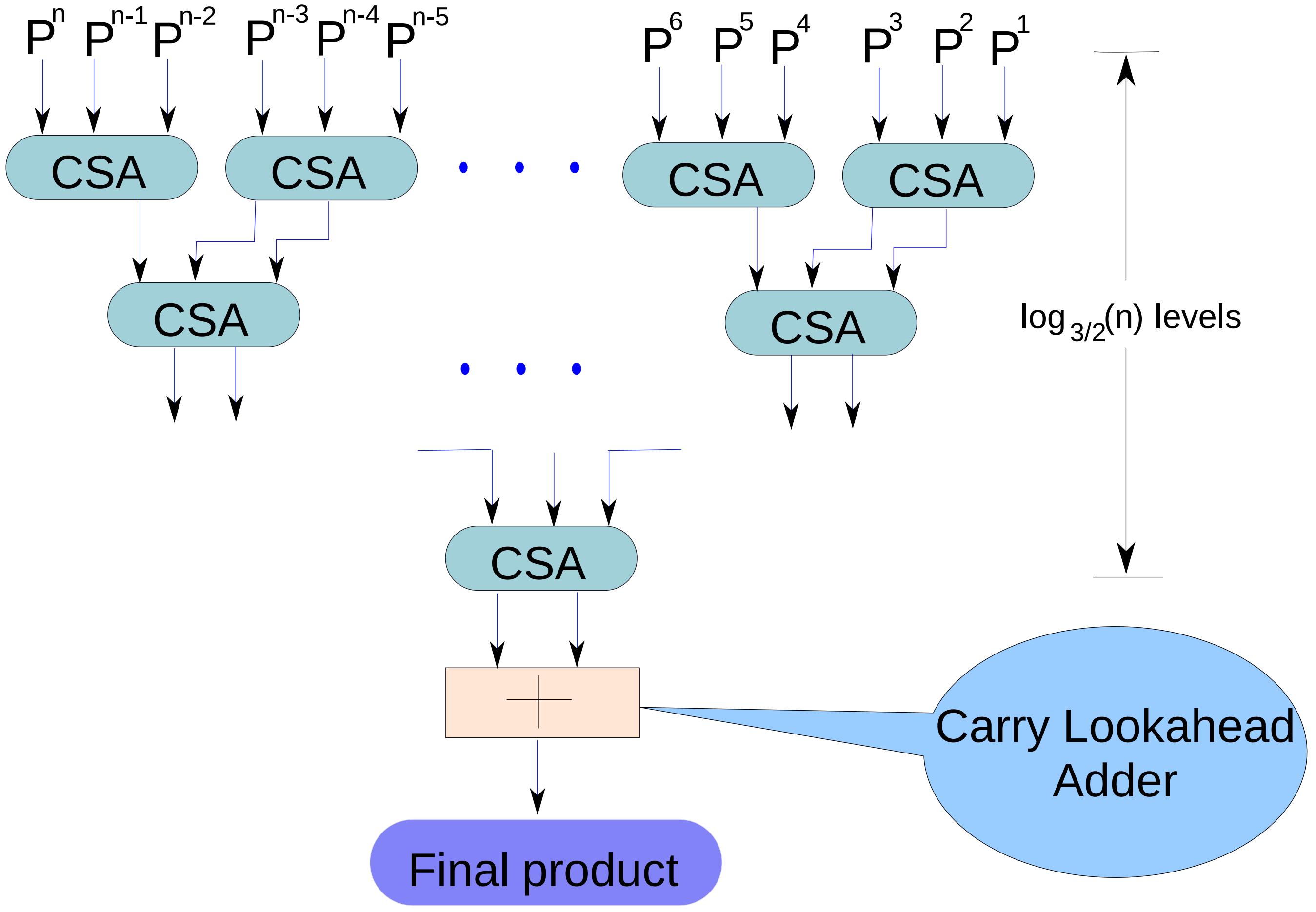
n-bit CSA Adder - II

- How to generate D and E ?
  - Add all the corresponding sets of bits (A_i, B_i, and C_i) independently
  - set D_i to the carry bit produced by adding (A_i, B_i, and C_i)
  - set E_i to the sum bit produced by adding (A_i, B_i, and C_i)
- Time Complexity :
  - All the additions are done in parallel
  - This takes O(1) time

Wallace Tree Multiplier

- Basic Idea
  - Generate n partial sums
  - Partial sum : $P_i = 0$, if the ith bit in the multiplier is 0
  - $P_i = N << (i-1)$, if the the ith bit in the multiplier is 1
  - Can be done in parallel : O(1) time
- Add all the n partial sums
  - Use a tree based adder

Tree of CSA Adders



P^n P^{n-1} P^{n-2} P^{n-3} P^{n-4} P^{n-5}
P^6 P^5 P^4 P^3 P^2 P^1
CSA
CSA
CSA
CSA
CSA
CSA
CSA
$\log_{3/2}(n)$ levels
Carry Lookahead Adder
Final product

Tree of CSA Adders

- Group the partial sums into sets of 3
  - Use an array of CSA adders to add 3 numbers (A,B,C) to produce two numbers (D,E)
  - Hence, reduce the set of numbers by 2/3 in each level
- After $\log_{3/2}(n)$ levels, we are left with only two numbers
- Use a CLA adder to add them

Time Complexity

- Time to generate all the partials sums → O(1)
- Time to reduce n partial sums to sum of two numbers
  - Number of levels → O(log(n))
  - Time per level → O(1)
  - Total time for this stage → O(log(n))
- Last step
  - Size of the inputs to the CLA adder → (2n-1) bits
  - Time taken → O(log(n))
- Total Time : O(log(n))

Outline

- Addition
- Multiplication
- Division
- Floating Point Addition
- Floating Point Multiplication
- Floating Point Division

THE END