Computer Architecture (Integer Arithmetic)

विद्याधनं सर्वधन प्रधानम्

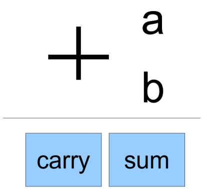
Subhasis Bhattacharjee
Department of Computer Science and Engineering,
Indian Institute of Technology, Jammu

September 18, 2023

Outline I

Addition - Single Bit Adder

Addition - Sum and Carry



Truth Table			
Inputs		Outputs	
a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

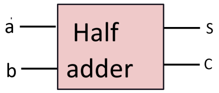
Boolean Expressions

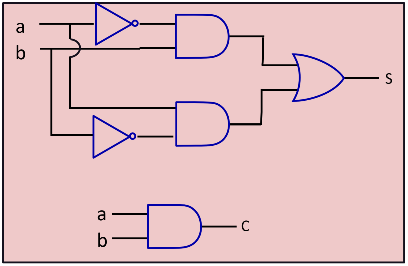
Sum $= a \oplus b = \overline{\text{a}}.b + a.\overline{\text{b}}$

Carry $= a.b$

Symbol	**Meaning**
.	AND operation
$+$	OR operation
$\oplus$	XOR operation

Half Adder





Half-Adder adds two 1 bit numbers (say, a and b) to produce 2 bits result (Sum and Carry).

Full Adder

Truth Table				
Inputs			Outputs	
a	b	c_{in}	Sum	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

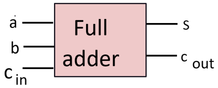
- It can add three 1 bit numbers (say, a, b, c) to produce a 2 bits (Sum and Carry) output.
- Usually, we take the Carry of previous (lower) bit as the 3rd input in Full-Adder. We call it c_{in} or Carry-In.
- Full Adder produces 2 outputs (Sum and c_{out}).
- c_{out} is termed as Carry out of this addition.

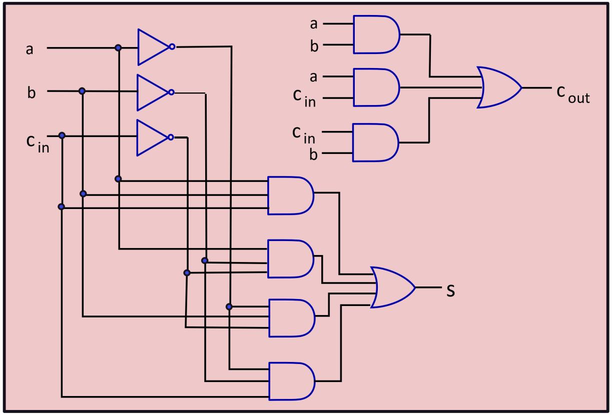
Full Adder - Boolean Expressions

Boolean Expressions for Full Adder

$$\begin{aligned} Sum &= a \oplus b \oplus c_{in} \\ &= a.\overline{b}.\overline{c_{in}} + \overline{a}.b.\overline{c_{in}} + \overline{a}.\overline{b}.c_{in} + a.b.c_{in} \\ c_{out} &= a.b + a.c_{in} + b.c_{in} \end{aligned}$$

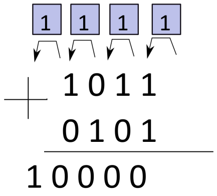
Circuit for the Full Adder





Addition - Multi-Bit Adder

Addition of two n bit numbers



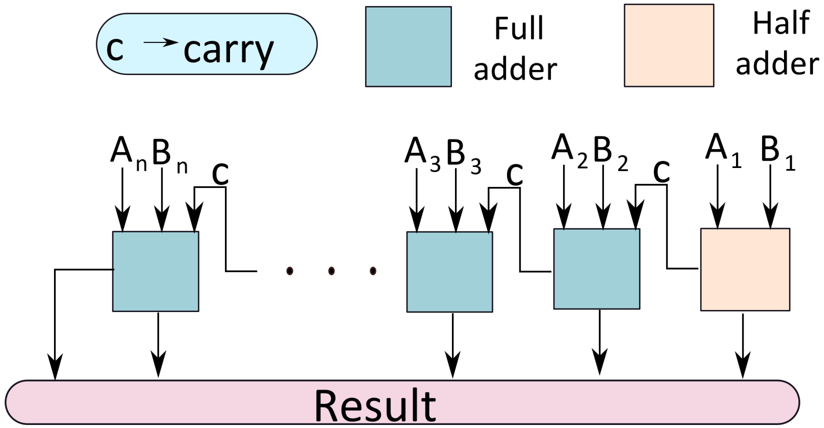
- We start from the lsb
- Add the corresponding pair of bits and the carry in
- Produce a sum bit and a carry out

Observations - Multi-bit Adder Circuit & Carry Propagation

- 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

Ripple Carry Adder



Working of Ripple Carry Adder

A and B are two n-bit Numbers

- Number the bits: A_0 to A_{n-1} and B_0 to B_{n-1}
- lsb $\rightarrow A_0$ and B_0
- msb $\rightarrow A_{n-1}$ and B_{n-1}

Add A + B

- Use a half adder to add A_0 and B_0.
- Send the c_{out} to a full adder that adds: $A_1 + B_1 + c_{in}$.
- Proceed in a similar manner till the msb

Ripple Carry Adder - Computation Time

Time Computation based on Electronic Design / Chip Fabrication Process

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

Time complexity - Let n is the number of bits

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

Issues with Ripple Carry Adder - Computation Time

Issues:

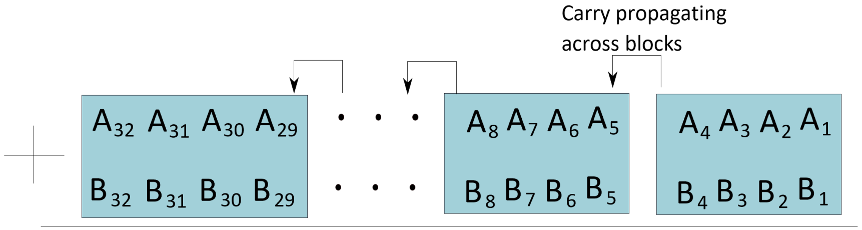
- Addition proceeds from lsb to msb - and one bit position at a time.
- It is very slow.
- $O(n)$ is too big for doing just addition of two n-bit numbers.
- How many additions are there in Multiplication of two n-bit numbers.
- Can we find a better Adder?

Carry Select Adder

Carry Select Adder

- Group bits into **Groups** of size (k) bits
- Ex: Adding two 32 bit numbers A and B, and k = 4, then we have 8 blocks of 4 bits each.
- Produce the result of each group with a small ripple carry adder.
- In this case, the carry propagates across groups
  - Time complexity is O(n)
  - Apparently, No benefit.
- But - If all groups compute in parallel / simultaneously
  - How will a group know the carry of the preceding group?

Carry Select Adder - Example: 32 bits numbers and k = 4


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

Carry Select Adder - Better Implementation

Stage I: Add the numbers in all Groups in parallel

For each group, produce two results:

- $\{Sum^0_{Gi}, Carry^0_{Gi}\}$ = Assuming an input carry of **0** from previous group.
- $\{Sum^1_{Gi}, Carry^1_{Gi}\}$ = Assuming an input carry of **1** from previous group.

So, for each group we have two results (k-bits sum, 1-bit carry out).

Stage II: Carry Propagation & Finalizing Result

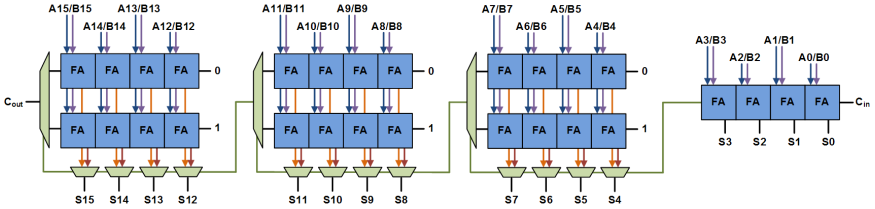
- Start at the least significant block (Input carry is 0).
- Choose the appropriate result from stage I ($\{Sum^0_{G0}, Carry^0_{G0}\}$)
- We now know the input carry $Carry^0_{G0}$ for the second block & Choose the appropriate result
- Result contains the input carry for the third block, and so on $\cdots$.

Carry Select Adder – Stage II - Carry Propagation - Explanation

Stage II: Clarification

- Select the correct carry out of previous group, and,
  - use this to **select the result** of next group.
  - How to do this selection?
- This continue until all groups are processed from least-significant group to most-significant group.
- Input carry for least-significant group is 0 (known prior).

Carry Select Adder - Stage I & II



Carry Select Adder - Time Requirement

- Stage I:
  - Each group size is k.
  - A group takes k units of time (A group is just a Ripple Carry Adder).
- Stage II:
  - There are n/k groups [i.e., G0 to G(k-1)]
  - Stage II takes (n/k) units of time (Carry propagation from least-significant group to most-significant group]
  - Total time : $(k + n/k)$.
- Optimal value of $k = \sqrt{n}$.

Proof: Optimal value of $k = \sqrt{n}$

Total Time (T) = $(k + n/k)$.
We see that, T is **minimized** when $k = n/k$, or, $k^2 = n$, or, $k = \sqrt{n}$.

- What does it signify?
- If n = 64 bits, then we should use 8 groups of 8-bits each. And the delay will be 16 units.

Carry Select Adder - Time Complexity

- T $= O(\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$.
- Can we do better?

Carry Lookahead Adder

Carry Lookahead Adder

- 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:

- **Generate function**: A new carry is generated. denote as g_i
- $g_i = A_i \cdot B_i$.
- Carry will be generated only if both bits (A_i and B_i) are 1.

Propagate function:

- Whether the bits (A_i and B_i) will propagate the input carry to the next stage? (i.e., Carry-out = carry-in).
- $p_i = A_i \oplus B_i$.
- Carry will be propagated to the next stage if exactly one of the bits (either A_i or B_i) is 1.

g_i and p_i are Generate and Propagate functions at bit level.

Using the g_i & p_i Functions

- If we have the generate and propagate values for a bit pair, we can determine the carry out
- $C_{out} = g_i + p_i \cdot C_{in}$

Example 1: Compute g_i, p_i, and C_{out}.

Given: Let $A_i = 0$, $B_i = 1$. Let the input carry be C_{in}.
Answer:

- $g_i = A_i \cdot B_i = 0 \cdot 1 = 0$
- $p_i = A_i \oplus B_i = 0 \oplus 1 = 1$
- $C_{out} = g_i + p_i \cdot C_{in} = 0 + 1 \cdot C_{in} = C_{in}$

Using the g_i & p_i Functions - Example 2 & 3

Example 2: Compute g_i, p_i, and C_{out}.

Given: Let $A_i = 1$, $B_i = 1$. Let the input carry be C_{in}.
Answer:

- $g_i = A_i \cdot B_i = 1 \cdot 1 = 1$ [i.e., this bit will generate Carry = 1 irrespective of C_{in}].
- $p_i = A_i \oplus B_i = 1 \oplus 1 = 0$
- $C_{out} = g_i + p_i \cdot C_{in} = 1 + 1 \cdot C_{in} = \mathbf{1}$

Example 3: Compute g_i, p_i, and C_{out}.

Given: Let $A_i = 0$, $B_i = 0$. Let the input carry be C_{in}.
Answer:

- $g_i = A_i \cdot B_i = 0 \cdot 0 = 0$
- $p_i = A_i \oplus B_i = 0 \oplus 0 = 0$
- $C_{out} = g_i + p_i \cdot C_{in} = 0 + 0 \cdot C_{in} = \mathbf{0}$

Notations - Summary

Let us consider two corresponding bits of A and B (A_i and B_i).

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

Derive Some Carry out bits - $C_{out}^1, C_{out}^2, \cdots, C_{out}^n$ - Pg1

C_{out}^1

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

C_{out}^2

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

C_{out}^3

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

Derive Some Carry out bits - $C_{out}^1, C_{out}^2, \cdots, C_{out}^n$ - Pg2

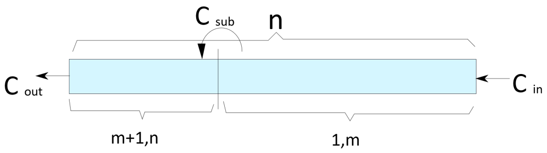
C_{out}^4

$$\begin{aligned}
C_{out}^4 &= g_4 + p_4 \cdot C_{in}^4 = g_4 + p_4 \cdot C_{out}^3 \\
&= g_4 + p_4 \cdot ((g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1) + p_3 \cdot p_2 \cdot p_1 \cdot C_{in}^1) \\
&= (g_4 + p_4 \cdot g_3 + p_4 \cdot p_3 \cdot g_2 + p_4 \cdot p_3 \cdot p_2 \cdot g_1) \\
&\quad + p_4 \cdot p_3 \cdot p_2 \cdot p_1 \cdot C_{in}^1
\end{aligned}$$

G and P for Multibit Systems

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 $(G_{m+1,n}, G_{1,m}, P_{m+1,n}, P_{1,m})$
  - $G_{1,n} \rightarrow$ denotes Carry Generate function for bits from 1 to n.
  - $P_{1,n} \rightarrow$ denotes Carry Propagation function for bits from 1 to n.

Computing G and P Quickly - II

Computing C_{out}

$$\begin{aligned} C_{out} &= G_{m+1,n} + P_{m+1,n} \cdot C_{sub} \\ &= G_{m+1,n} + P_{m+1,n} \cdot (G_{1,m} + P_{1,m} \cdot C_{in}) \\ &= [G_{m+1,n} + P_{m+1,n} \cdot G_{1,m}] + [P_{m+1,n} \cdot P_{1,m} \cdot C_{in}] \\ &= G_{1,n} + P_{1,n} \cdot C_{in} \end{aligned}$$

Computing G and P

$$\begin{aligned} G_{1,n} &= G_{m+1,n} + P_{m+1,n} \cdot (G_{1,m} \\ P_{1,n} &= P_{m+1,n} \cdot P_{1,m} \end{aligned}$$

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

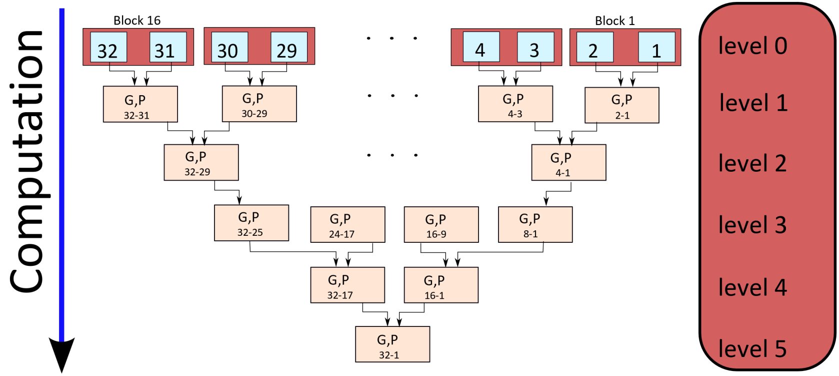
Fast algorithm to compute G and P:

- Use divide-and-conquer
- Compute G and P functions in $O(\log(n))$ time

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
- and, so on...
- Find the G and P functions for a block of size: 32 bits (all bits).

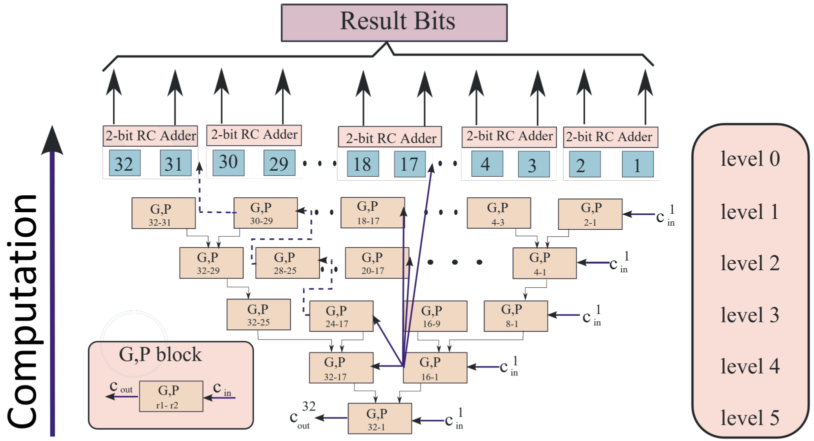
Carry Lookahead Adder – Stage I



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 : $O(\log(n))$

CLA Adder – Stage II



Connection of the G,P Blocks

- Each G,P block represents a range of bits (r2, r1) for (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 Cin1
  - 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 - Time Complexity

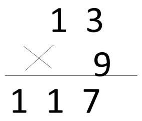
- In a similar manner, the carry propagates to all the RC adders at the zeroth level
- Each of them compute the correct result
- Time taken by Stage II:
  - Time taken for a carry to propagate from the (16,1) node to the RC adders
  - $O(\log(n))$
- Time taken by Stage I: $O(\log(n))$
- Total time :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

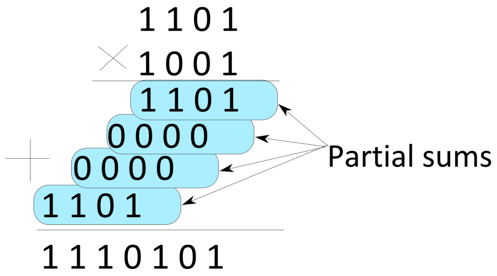
Time Complexities of Various Adders

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

Integer Multiplication

Multiplication Example





Multiplicand = 13, Multiplier = 9
Product = 117

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...

Size of the Product:

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

Some Definitions

Partial sum:

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:

Partial product: It is the sum of a set of partial sums.

Multiplying 32 bit numbers

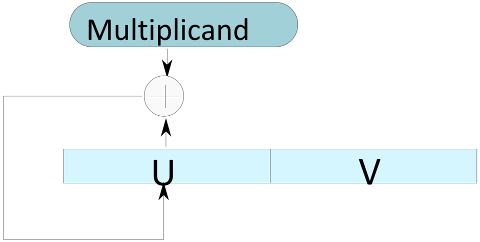
- Let us design an iterative multiplier that multiplies two 32 bit signed values to produce a 64 bit result
- Multiplying two signed 32 bit numbers, and saving the result as a 32 bit number is the same as
- Multiplying two unsigned 32 bit numbers (assuming no overflows)

Theorem on 2's Complement Number

Theorem

A signed n bit number $A = A_{1\cdots n-1} - A_n 2^{n-1}$.
Where,
A_i is the i^th bit in A's 2's complement based binary representation (the first bit i.e., A_1 is the LSB).
$A_{1\cdots n-1}$ 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.

Iterative Multiplication 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

		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	no operation:	00000	1110	0 $\Rightarrow$ --
	after shift:	00000	0111	
2	after add:	00011	0111	1 $\Rightarrow$ add 3
	after shift:	00001	1011	
3	after add:	00100	1011	1 $\Rightarrow$ add 3
	after shift:	00010	0101	
4	after sub:	01111	0101	1 $\Rightarrow$ sub 3
	after shift:	01111	1010	

Product(P)	1010	-6

-7 * (-7)

Multiplicand (N)	1001	-7
Multiplier (M)	1001	-7

		U	V	
	beginning:	00000	1001	
1	after add:	01001	1001	1 $\Rightarrow$ add -7
	after shift:	01100	1100	
2	no operation:	01100	1100	0 $\Rightarrow$ ----
	after shift:	00110	0110	
3	no operation:	00110	0110	0 $\Rightarrow$ ----
	after shift:	00011	0011	
4	after sub:	01010	0011	1 $\Rightarrow$ sub -7
	after shift:	01100	0001	

Product(P)		-6

11001010

-7 * (-7)

Multiplicand (N)	1001	-7
Multiplier (M)	1001	-7

	U	V
beginning:	00000	1001

Step		U	V	
1	after add:	01001	1001	1 $\Rightarrow$ add -7
	after shift:	00100	1100	
2	no operation:	00100	1100	0 $\Rightarrow$ ----
	after shift:	00010	0110	
3	no operation:	00010	0110	0 $\Rightarrow$ ----
	after shift:	00001	0011	
4	after sub:	01000	0011	1 $\Rightarrow$ sub -7
	after shift:	01100	0001	

Product(P)		-6

11001010

Algorithm - Explanation

- Take a look at the lsb of V
  - If it is 0 $\rightarrow$ do nothing
  - If it is 1 $\rightarrow$ 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: lsb of V = msb of M (multiplier)
  - If it is 0 $\rightarrow$ do nothing
  - If it is 1 then
    - Multiplier is negative
    - Recall: $A = A_{1\cdots n-1} - A_n 2^{n-1}$.
    - Hence, we need to subtract the multiplicand if the msb of the multiplier is 1.

Iterative Multiplication - Time Complexity

- There are n loops
  - Each loop takes $\log(n)$ time
  - Each time we need to addition & Best Addition (eg, Carry Lookahead Adder) takes $O(\log n)$ time.
- Total time: $O(n \log(n))$

Booth Multiplier

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
  - Assume there is a continuous sequence of 1s from i^th bit to j^th bit
  - We Need to perform (j - i + 1) additions.
  - Can we do something smart?

A continuous sequence of 1s from i^th bit to j^th bit can be written:

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

Booth Multiplication Idea

For a Sequence of 1s

- 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.

Operation of the Algorithm

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

(current bit, previous bit)	**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
```

Example 1

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

	U	V
beginning:	00000	0010

Step		U	V	
1	before shift:	00000	0010	00 $\rightarrow$ --
	after shift:	00000	0001	
2	before shift:	11101	0001	10 $\rightarrow$ add -3
	after shift:	11110	1000	
3	before shift:	00001	1000	01 $\rightarrow$ add 3
	after shift:	00000	1100	
4	before shift:	00000	1100	00 $\rightarrow$ --
	after shift:	00000	0110	

Product(P)	0110	6

Example 2

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

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

Product(P)	1010	-6

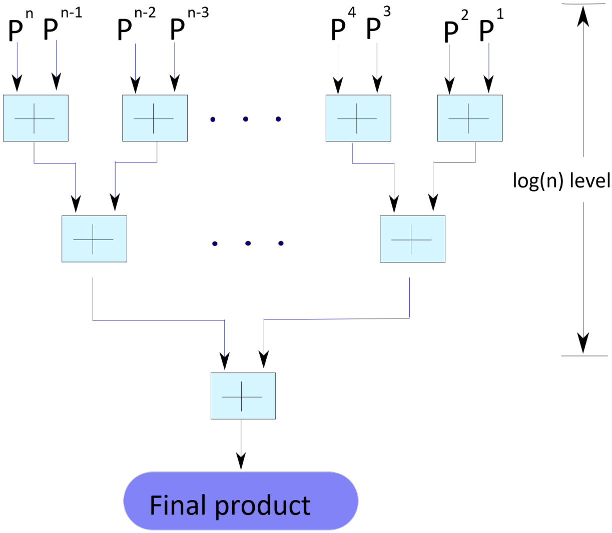
Booth Algorithm - Time Complexity

- $O(n \log(n))$
- Worst case input (sequence of alternating 0's and 1's)
- Multiplier = 10101010...

Parallel Multiplier

$$\begin{array}{rr}
 & 1\,0\,0\,1 \\
\times & 1\,1\,0\,1 \\
\hline
 & \left.\begin{array}{r}
\boxed{1\,0\,0\,1} \\
\boxed{0\,0\,0\,0\,0} \\
\boxed{1\,0\,0\,1\,0\,0} \\
\boxed{1\,0\,0\,1\,0\,0\,0}
\end{array}\right\} \text{partial sums} \\
\hline
\end{array}$$

Tree Based Adder for Partial Sums

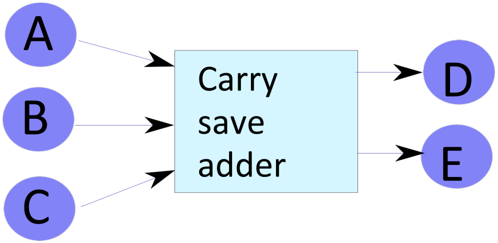


Time Complexity

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

Better Way of Adding Partial Products

Carry Save Adder



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

Carry Save Adder - Example

Example 1

A	1011
B	1101
C	0001
E	0111
D	1001-

Example 2

A	10111010101011011111000000001101
B	11011110101011011011111011101111
C	00010010101101110101001101010010
E	01110110101101110001110110110000
D	100110101010110111110010010011110

1-bit CSA Adder

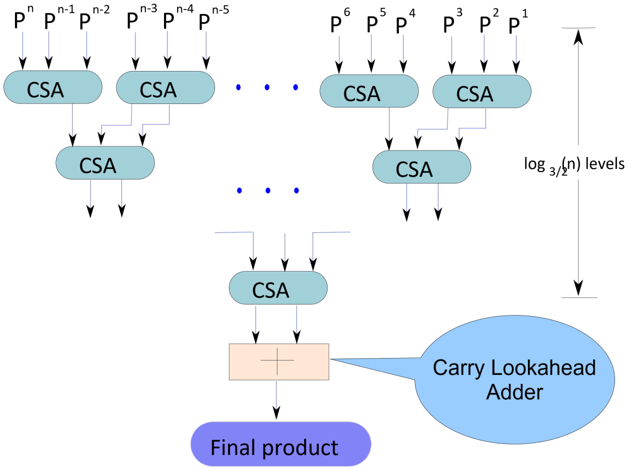
- Add three bits - a, b, and c
  - such that $a + b + c = 2d + e$
  - d and e are also single bits
- We can conveniently set
  - e to the sum bit
  - d to the carry bit74

Wallace Tree Multiplier

Basic Idea:

- Generate n partial sums
- Partial sum : $P_i = 0$, if the ith bit in the multiplier is 0
- $P_i = N \ll (i-1)$, 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
log $_{3/2}$(n) levels
Carry Lookahead Adder
Final product

Tree of CSA Adders - Operations

- 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
  - size of each number = 2n bits.
- Use a CLA adder to add them

Wallace Tree Multiplier - 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 $\rightarrow O(\log(n))$
    - Time per level $\rightarrow O(1)$
    - Total time for this stage $\rightarrow O(\log(n))$
- Last step
    - Size of the inputs to the CLA adder $\rightarrow (2n - 1)$ bits
    - Time taken $\rightarrow O(\log(n))$
- Total Time : $O(log(n))$

Integer Division

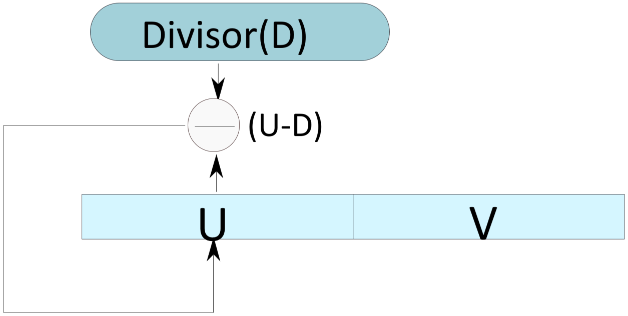
Integer Division

- Let us only consider positive numbers
  - N = DQ + R
  - N $\rightarrow$ Dividend
  - D $\rightarrow$ Divisor
  - Q $\rightarrow$ Quotient
  - R $\rightarrow$ Remainder
- Properties
  - Property 1: R < D, R >= 0
  - Property 2: Q is the largest positive integer satisfying the equation (N = DQ +R) and Property 1

How to Reduce the Problem

- We need to find Q_n
- We can try both values: 0 and 1
  - First try 1
    - If: $N - D2^{n-1} >= 0$, $Q_n = 1$ (maximize the quotient)
  - Otherwise it is 0
- Once we have reduced the problem
  - We can proceed recursively

Iterative Divider



Initial Value:

V holds the dividend (N), U = 0

Restoring-based Division

Restoring Division

Algorithm 3: Restoring algorithm to divide two 32 bit numbers

Data: Divisor in D, Dividend in V, $U = 0$

Result: U contains the remainder (lower 32 bits), and V contains the quotient

```
i ← 0
for i < 32 do
    i ← i + 1
    /* Left shift UV by 1 position                                    */
    UV ← UV << 1
    U ← U - D
    if U ≥ 0 then
        q ← 1
    end
    else
        U ← U + D
        q ← 0
    end
    /* Set the quotient bit                                           */
    LSB of V ← q
end
```

Restoring Division - Example

Dividend (N)	00111
Divisor (D)	0011

		U	V
	beginning:	00000	0111
1	after shift:	00000	111X
	end of iteration:	00000	1110
2	after shift:	00001	110X
	end of iteration:	00001	1100
3	after shift:	00011	100X
	end of iteration:	00000	1001
4	after shift:	00001	001X
	end of iteration:	00001	0010

Quotient(Q)	0010
Remainder(R)	0001

Restoring Division Algorithm - Explanation

Iteratively Do This:

- Consider each bit of the dividend
- Try to subtract the divisor from the U register
  - If the subtraction is successful, set the relevant quotient bit to 1
  - Else, set the relevant quotient bit to 0
- Left shift

Restoring Division - Time Complexity

- There are n iterations.
- Each iteration takes $\log(n)$ time
- Total time: $n \log(n)$

Restoring vs Non-Restoring Division

- We need to restore the value of register U
- Requires an extra addition or a register move
- Can we do without this?
  - Non Restoring Division

Non-Restoring-based Division

Algorithm 4: Non-restoring algorithm to divide two 32 bit numbers

Algorithm 4: Non-restoring algorithm to divide two 32 bit numbers

Data: Divisor in D, Dividend in V, $U = 0$

Result: U contains the remainder (lower 32 bits), and V contains the quotient

```
$i \leftarrow 0$
for $i$ < 32 do
    i ← i + 1
    /* Left shift $UV$ by 1 position                                    */
    $UV \leftarrow UV$ << 1
    if $U \geq 0$ then
        $U \leftarrow U - D$
    end
    else
        $U \leftarrow U + D$
    end
    if $U \geq 0$ then
        $q \leftarrow 1$
    end
    else
        $q \leftarrow 0$
    end
    /* Set the quotient bit                                             */
    lsb of $V \leftarrow q$
end
if $U$ <0 then
    $U \leftarrow U + D$
end
```

Non-Restoring Division - Example

Dividend (N)	00111
Divisor (D)	0011

		U	V
	beginning:	00000	0111
1	after shift:	00000	111X
	end of iteration:	00000	1110
2	after shift:	00001	110X
	end of iteration:	00001	1100
3	after shift:	00011	100X
	end of iteration:	00000	1001
4	after shift:	00001	001X
	end of iteration:	00001	0010

Quotient(Q)	0010
Remainder(R)	0001