Arithmetic Circuits

1. Adder
2. Multiplier
3. Arithmetic Logic Unit (ALU)
4. HDL for Arithmetic Circuit

Introduction

1. Digital circuits are frequently used for arithmetic operations

2. Fundamental arithmetic operations on binary numbers and digital circuits which perform arithmetic operations will be examined.

3. HDL will be used to describe arithmetic circuits.

4. An arithmetic/logic unit (ALU) accepts data stored in memory and executes arithmetic and logic operations as instructed by the control unit.

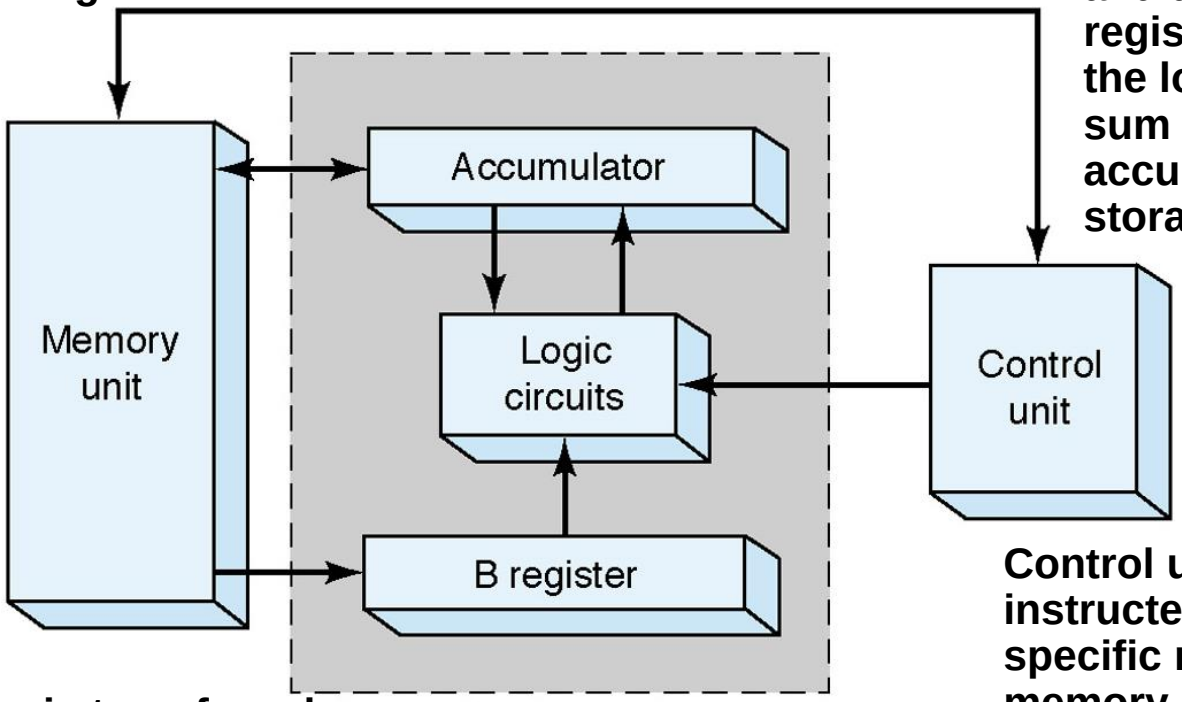
Arithmetic Circuits

The new number remains in the accumulator for further operations or can be transferred to memory for storage

4

3

Number in B register and accumulator register are added in the logic circuit, with sum sent to accumulator for storage.



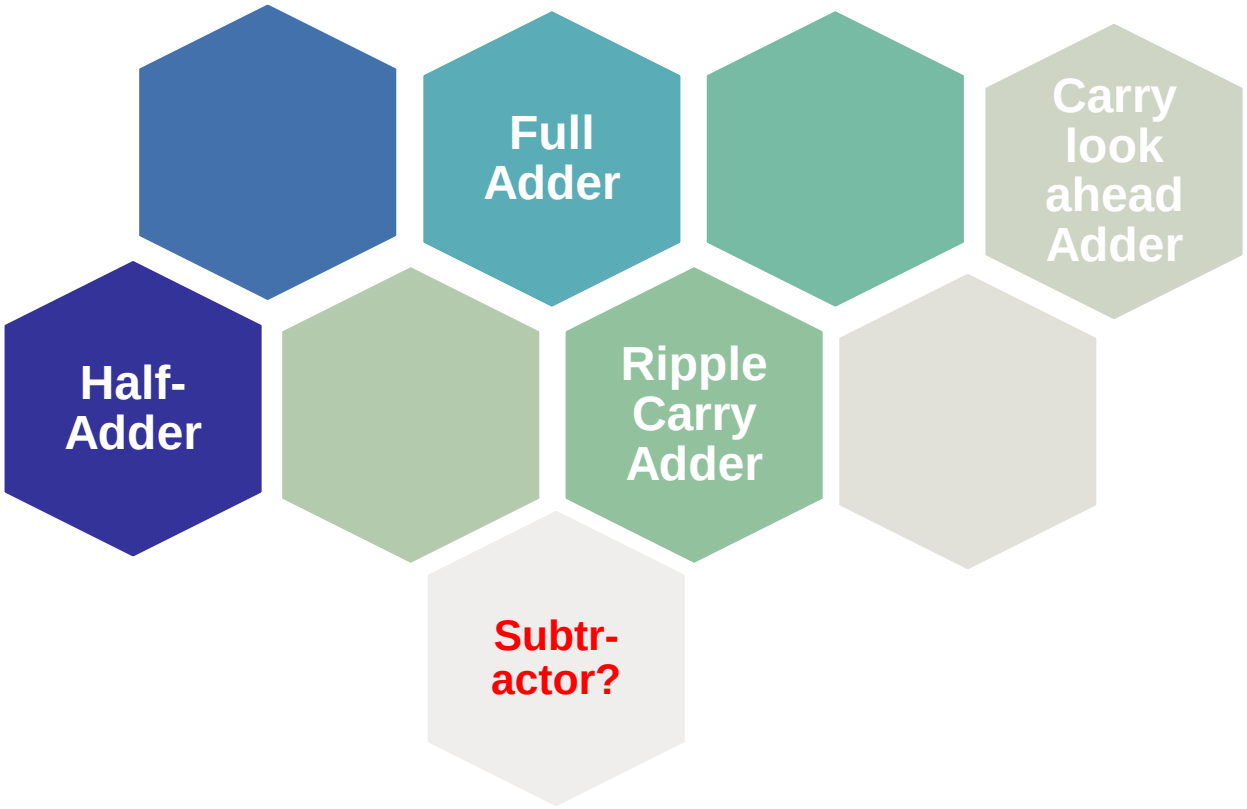
2

The number is transferred from memory to the B register.

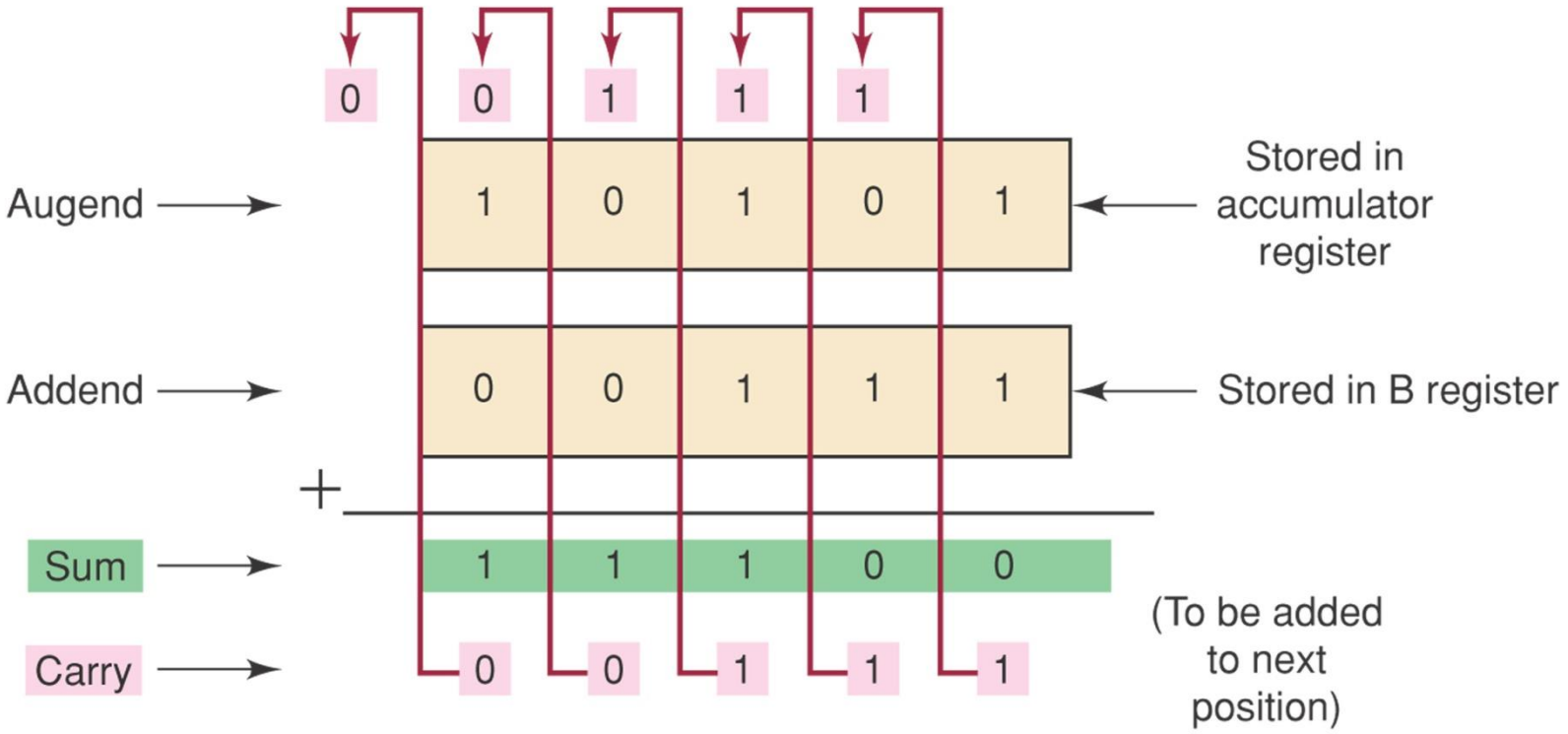
1

Control unit is instructed to add a specific number from a memory location to a number stored in the accumulator register.

Adder


Full Adder
Carry look ahead Adder
Half-Adder
Ripple Carry Adder
Subtr-actor?

Typical binary addition process



Half Adder

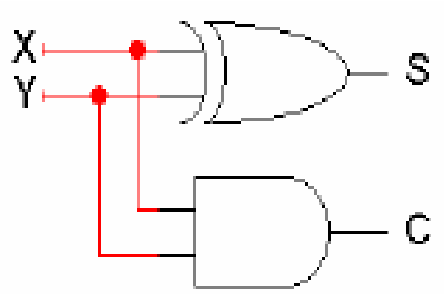
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

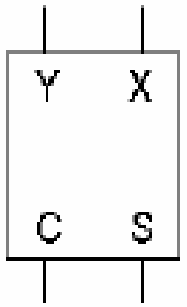
Half Adder equation

$C = XY$

$S = X'Y + XY'$

$= X \oplus Y$





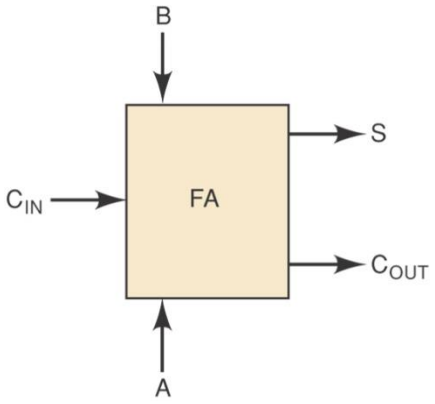
Full Adder

Augend bit input	Addend bit input	Carry bit input		Sum bit output	Carry bit output
A	B	C_{IN}		S	C_{OUT}
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

	1	1	1	0	
		1	0	1	1
+		1	1	1	0
	1	1	0	0	1

$$
\begin{aligned}
S &= \Sigma m(1,2,4,7) \\
&= X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in} \\
&= X'(Y'C_{in} + YC_{in}') + X(Y'C_{in}' + YC_{in}) \\
&= X'(Y \oplus C_{in}) + X(Y \oplus C_{in})' \\
&= X \oplus Y \oplus C_{in}
\end{aligned}
$$

$$
\begin{aligned}
C_{out} &= \Sigma m(3,5,6,7) \\
&= X'YC_{in} + XY'C_{in} + XYC_{in}' + XYC_{in} \\
&= (X'Y + XY')C_{in} + XY(C_{in}' + C_{in}) \\
&= (X \oplus Y)C_{in} + XY
\end{aligned}
$$



Full-adder- K map, Complete circuitry

	$\overline{C_{IN}}$	C_{IN}
$\overline{A}\overline{B}$	0	1
$\overline{A}B$	1	0
AB	0	1
$A\overline{B}$	1	0

K map for S

$S = \overline{A}\overline{B}C_{IN} + \overline{A}B\overline{C}_{IN} + ABC_{IN} + A\overline{B}\overline{C}_{IN}$

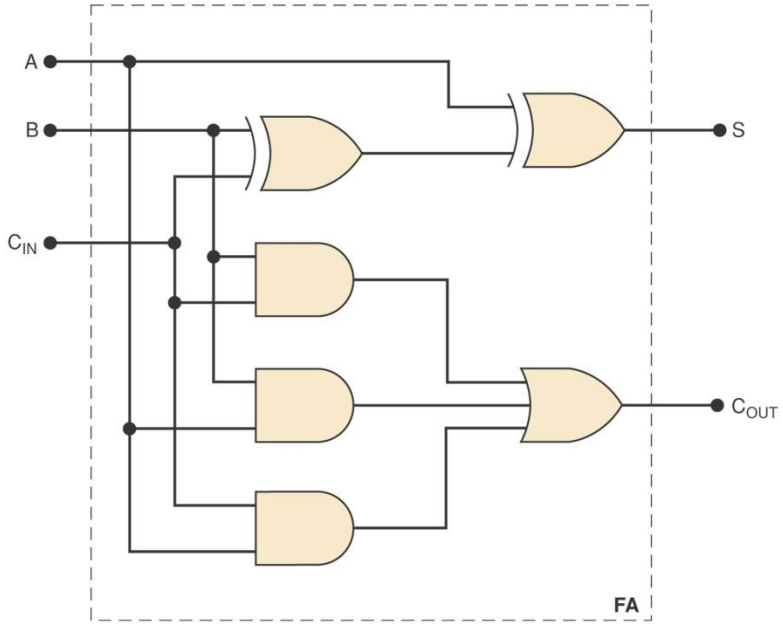
(a)

	$\overline{C_{IN}}$	C_{IN}
$\overline{A}\overline{B}$	0	0
$\overline{A}B$	0	1
AB	1	1
$A\overline{B}$	0	1

K map for C_{OUT}

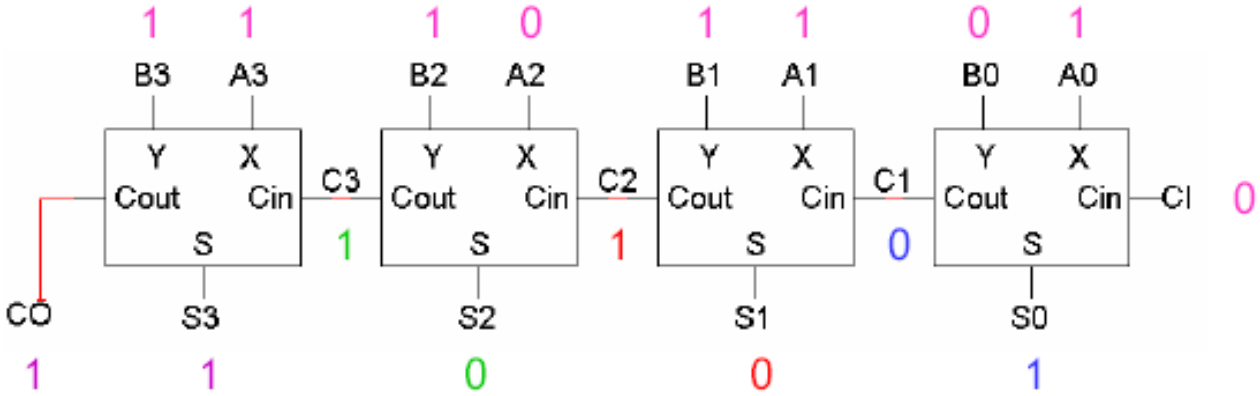
$C_{OUT} = BC_{IN} + AC_{IN} + AB$

(b)



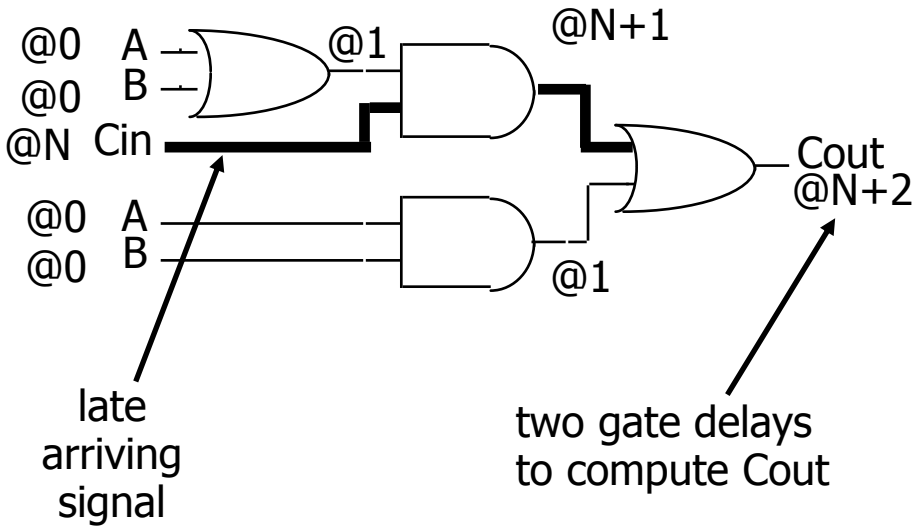
Ripple Carry Adder

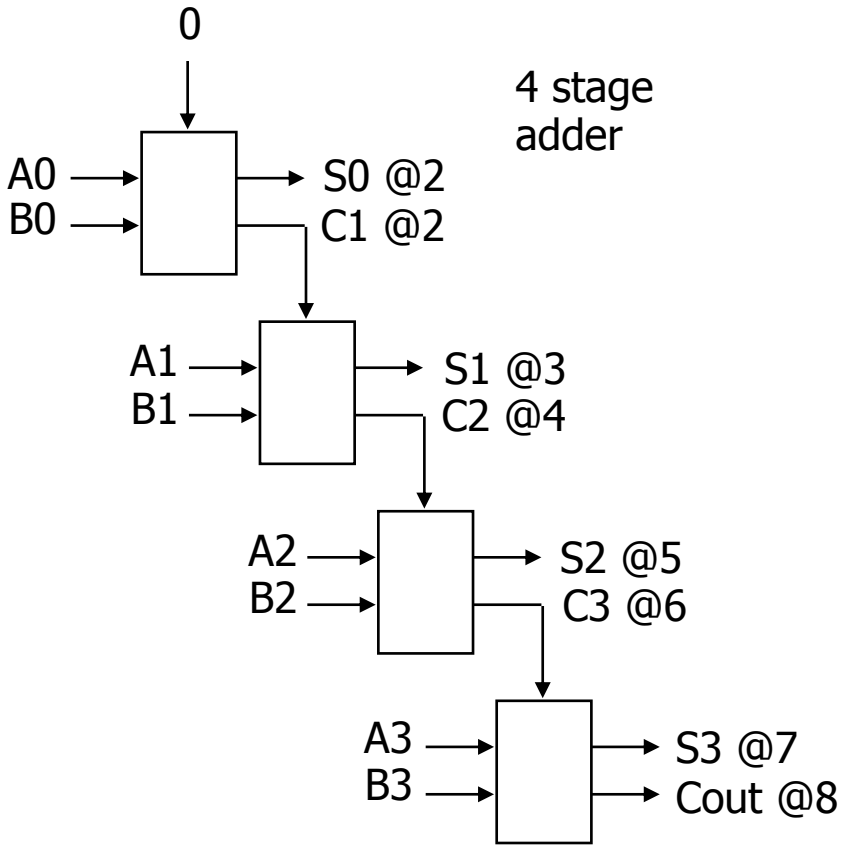
- This is called a ripple carry adder, because the inputs A0, B0 and CI "ripple" leftwards until CO and S3 are produced.
- Ripple carry adders are slow!
  - There is a very long path from A0, B0 and CI to CO and S3.
  - For an *n*-bit ripple carry adder, the longest path has 2*n*+1 gates.
  - The longest path in a 64-bit adder would include 129 gates!



Ripple-carry adders

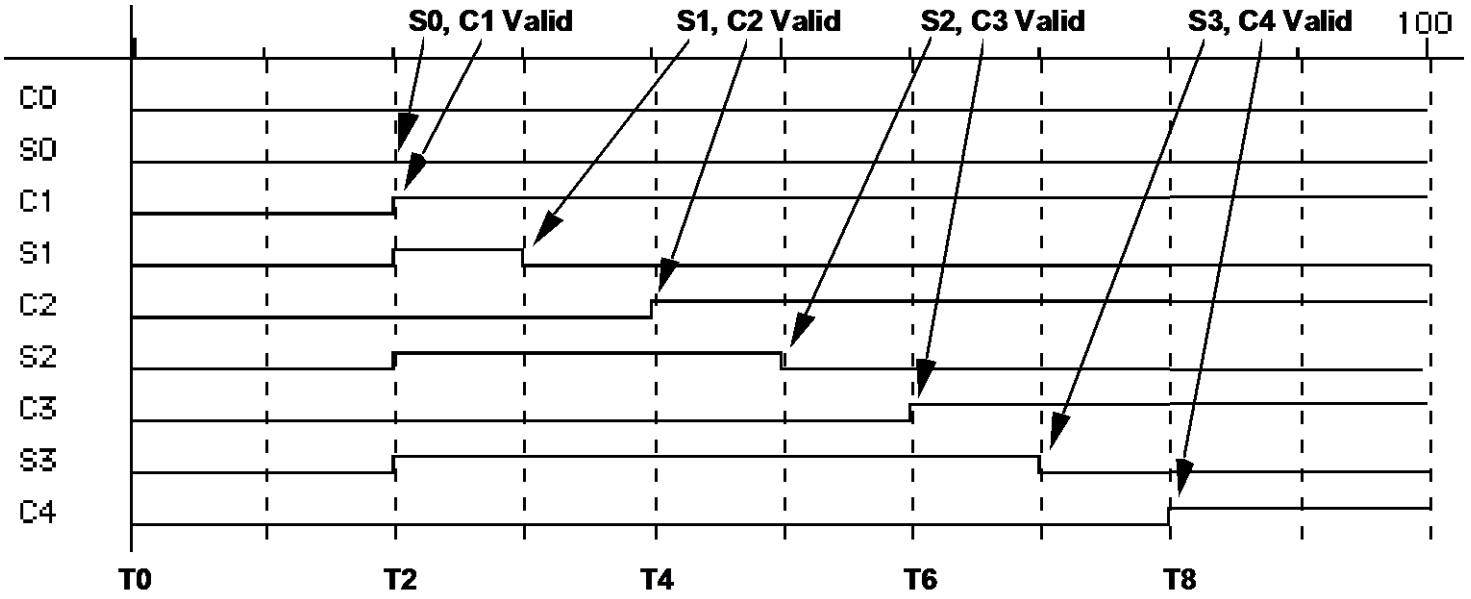
- Critical delay
  - the propagation of carry from low to high order stages





Ripple-carry adders (cont'd)

- Critical delay
  - the propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - carry must propagate through all bits



Carry Look-ahead Adder

Solution!

- Carry look-ahead solves this problem

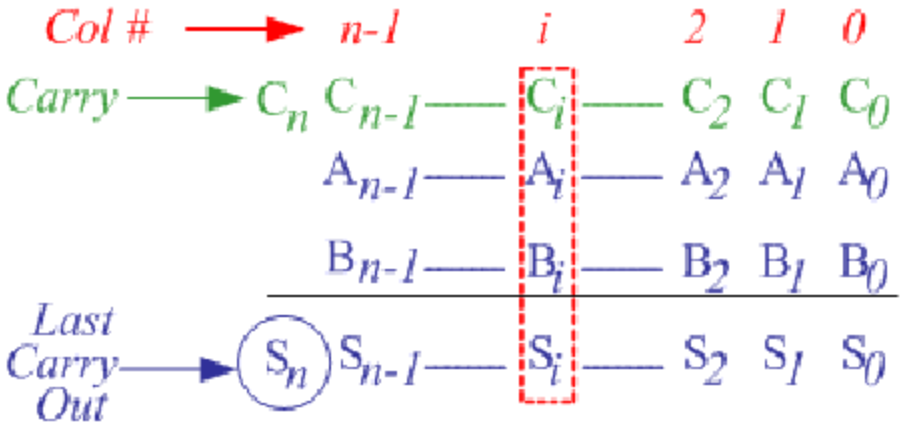
  by calculating the carry signals in advance, based on the input signals.
- It is based on the fact that a carry signal will be generated in two cases:

(1) when both bits Ai and Bi are 1, or

(2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

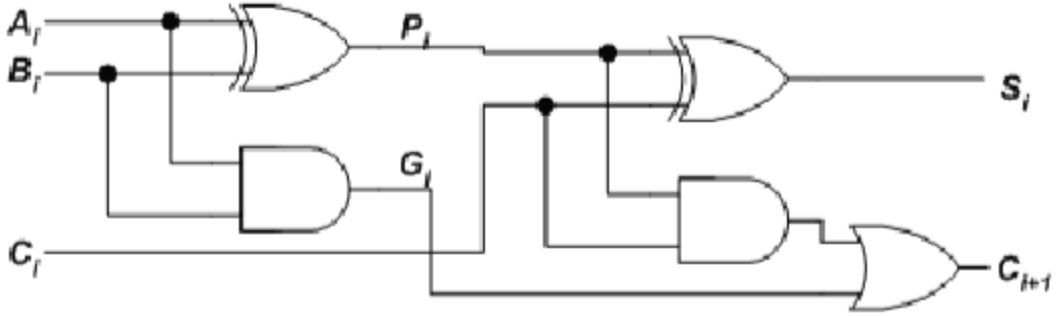
Carry look ahead adder

- To understand the carry propagation problem, let's consider the case of adding two n-bit numbers A and B.



Carry look ahead adder

The Figure shows the full adder circuit used to add the operand bits in the ith column; namely Ai & Bi and the carry bit coming from the previous column (Ci).



In this circuit, the 2 internal signals Pi and Gi are given by:

Pi = Ai $\oplus$ Bi ……..(1)

G i = Ai B i……(2)

The output sum and carry can be defined as :

Si = Pi $\oplus$ Ci ……(3)

C i +1 = G i + Pi C i ……(4)

Carry look ahead adder

- Gi is known as the carry Generate signal since a carry (Ci+1) is generated whenever Gi=1, regardless of the input carry (Ci).
- Pi is known as the carry propagate signal since whenever Pi =1, the input carry is propagated to the output carry, i.e., Ci+1. = Ci (note that whenever Pi =1, Gi =0).
- Computing the values of Pi and Gi only depend on the input operand bits (Ai & Bi) as clear from the Figure and equations.
- Thus, these signals settle to their steady-state value after the propagation through their respective gates.
- Computed values of all the Pi's are valid one XOR-gate delay after the operands A and B are made valid.
- Computed values of all the Gi's are valid one AND-gate delay after the operands A and B are made valid.

Carry-lookahead logic

- Carry generate: Gi = Ai Bi
  - must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

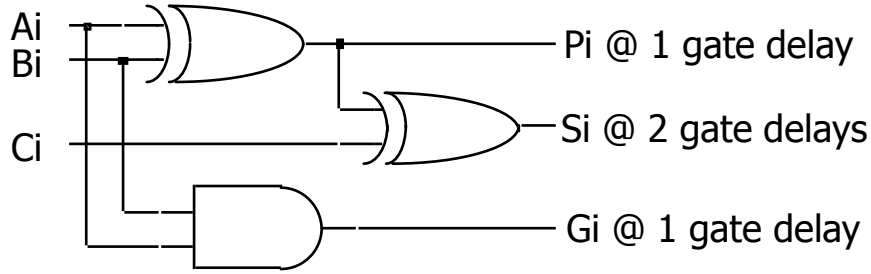
Carry-lookahead logic (cont'd)

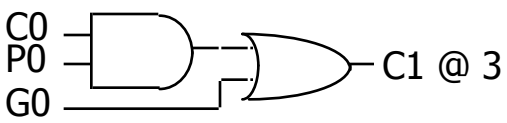
- Re-express the carry logic as follows:
  - $C1 = G0 + P0\ C0$
  - $C2 = G1 + P1\ C1 = G1 + P1\ G0 + P1\ P0\ C0$
  - $C3 = G2 + P2\ C2 = G2 + P2\ G1 + P2\ P1\ G0 + P2\ P1\ P0\ C0$
  - $C4 = G3 + P3\ C3 = G3 + P3\ G2 + P3\ P2\ G1 + P3\ P2\ P1\ G0 + P3\ P2\ P1\ P0\ C0$

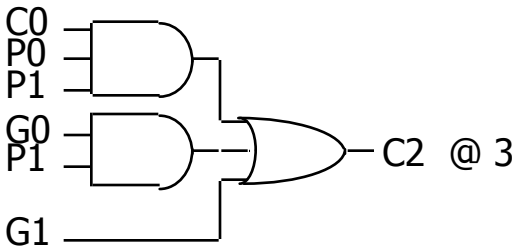
- Each of the carry equations can be implemented with two-level logic
  - all inputs are now directly derived from data inputs and not from intermediate carries
  - this allows computation of all sum outputs to proceed in parallel

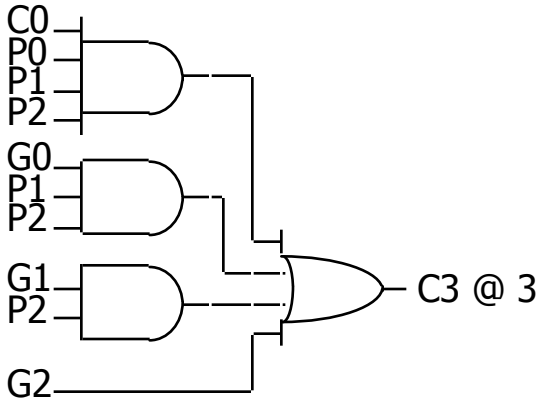
Carry-lookahead implementation

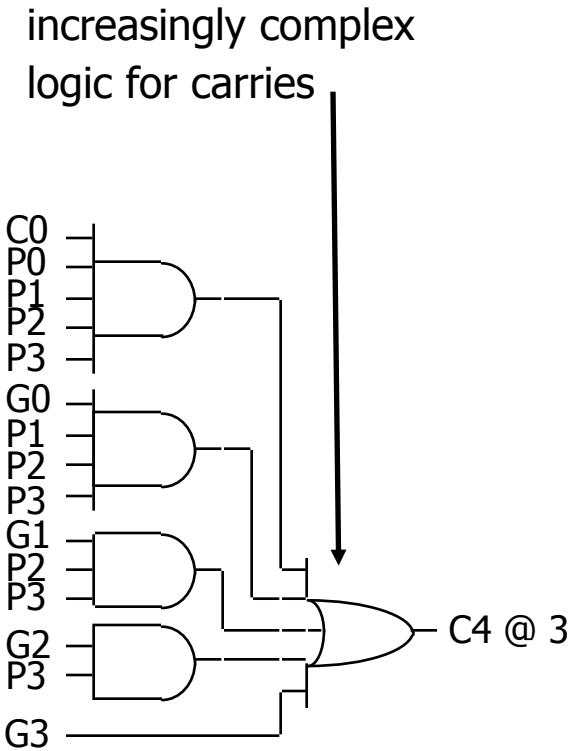
- Adder with propagate and generate outputs





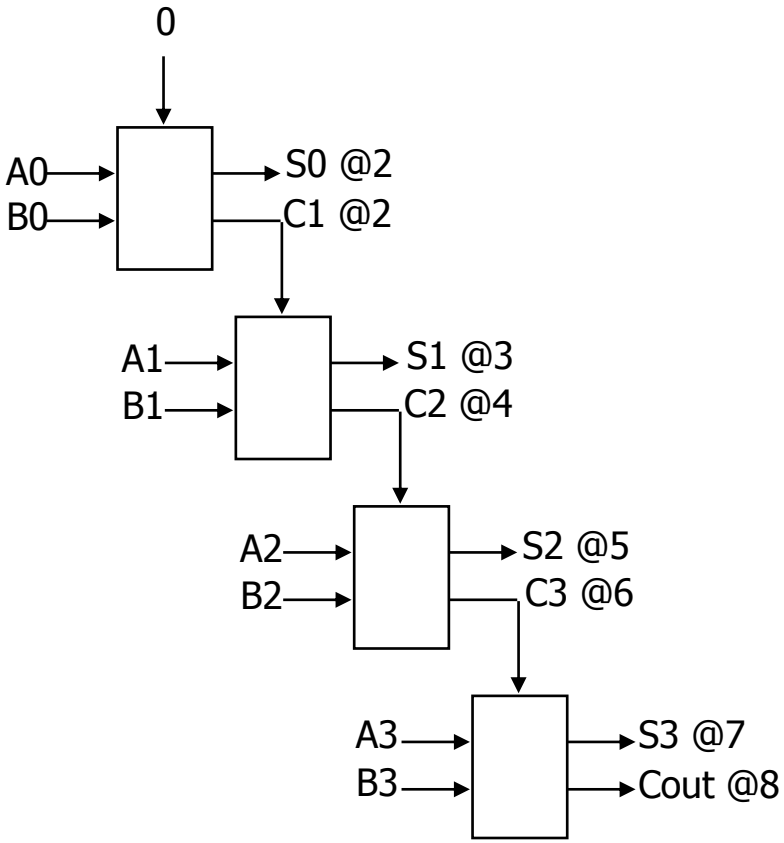


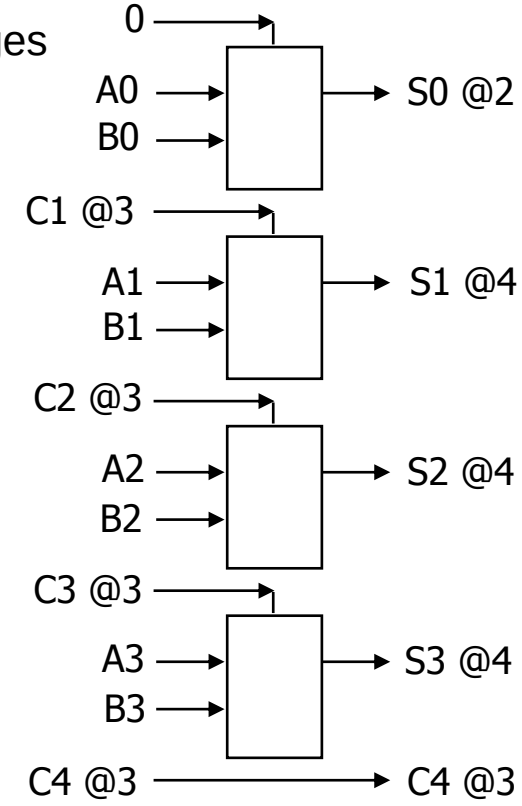




Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries
  - sums computed much more quickly in parallel
  - however, cost of carry logic increases with more stages





MULTIPLIER

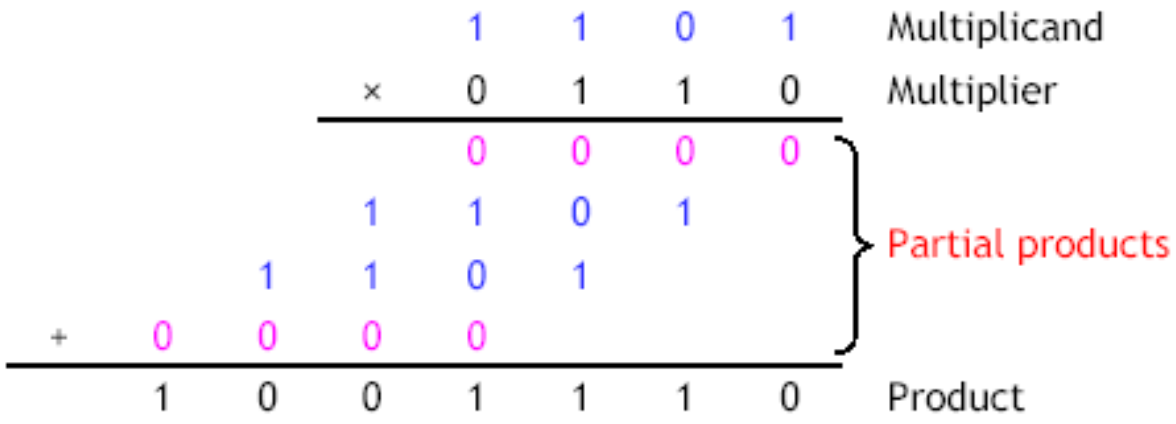
Multiplication

- Multiplication can't be that hard! It's just repeated addition, so if we have adders, we should be able to do multiplication also.
- Here's an example of binary multiplication

				1	1	0	1
			×	0	1	1	0
				0	0	0	0
			1	1	0	1	
		1	1	0	1		
+	0	0	0	0			
	1	0	0	1	1	1	0

Binary Multiplication

- Since we always multiply by either 0 or 1, the partial products are always either 0000 or the multiplicand (1101 in this example).
- There are four partial products which are added to form the result.
  - We can add them in pairs, using three adders.
  - The product can have up to 8 bits, but we can use four-bit adders if we stagger them leftwards, like the partial products themselves.



2X2 Binary Multiplication

- Here is an outline of multiplying the two-bit numbers A1A0 and B1B0, to produce the four-bit product P3-P0.

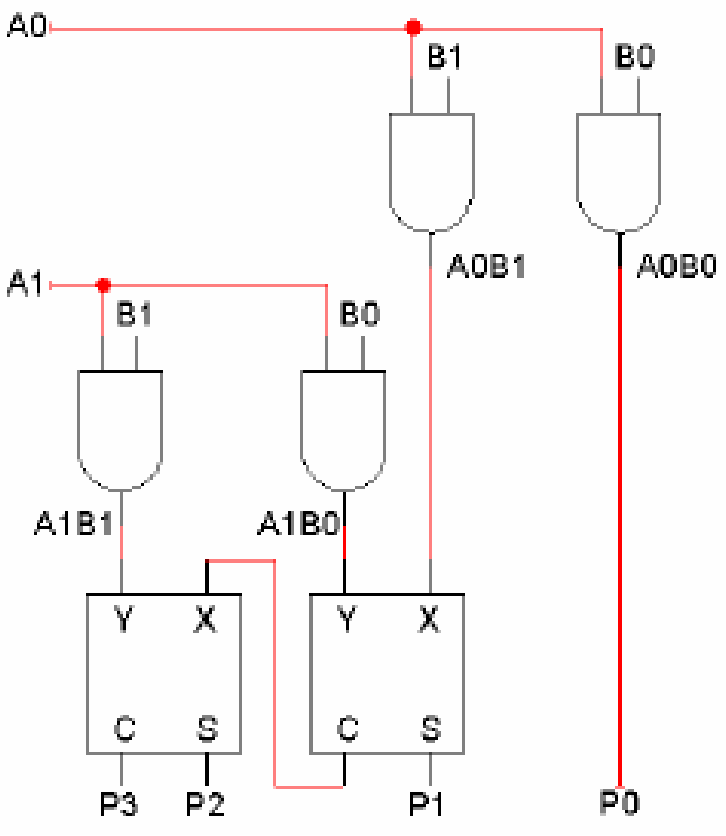
			B1	B0
		×	A1	A0
			A0B1	A0B0
	+	A1B1	A1B0	
	P3	P2	P1	P0

A	B	A×B	A•B
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

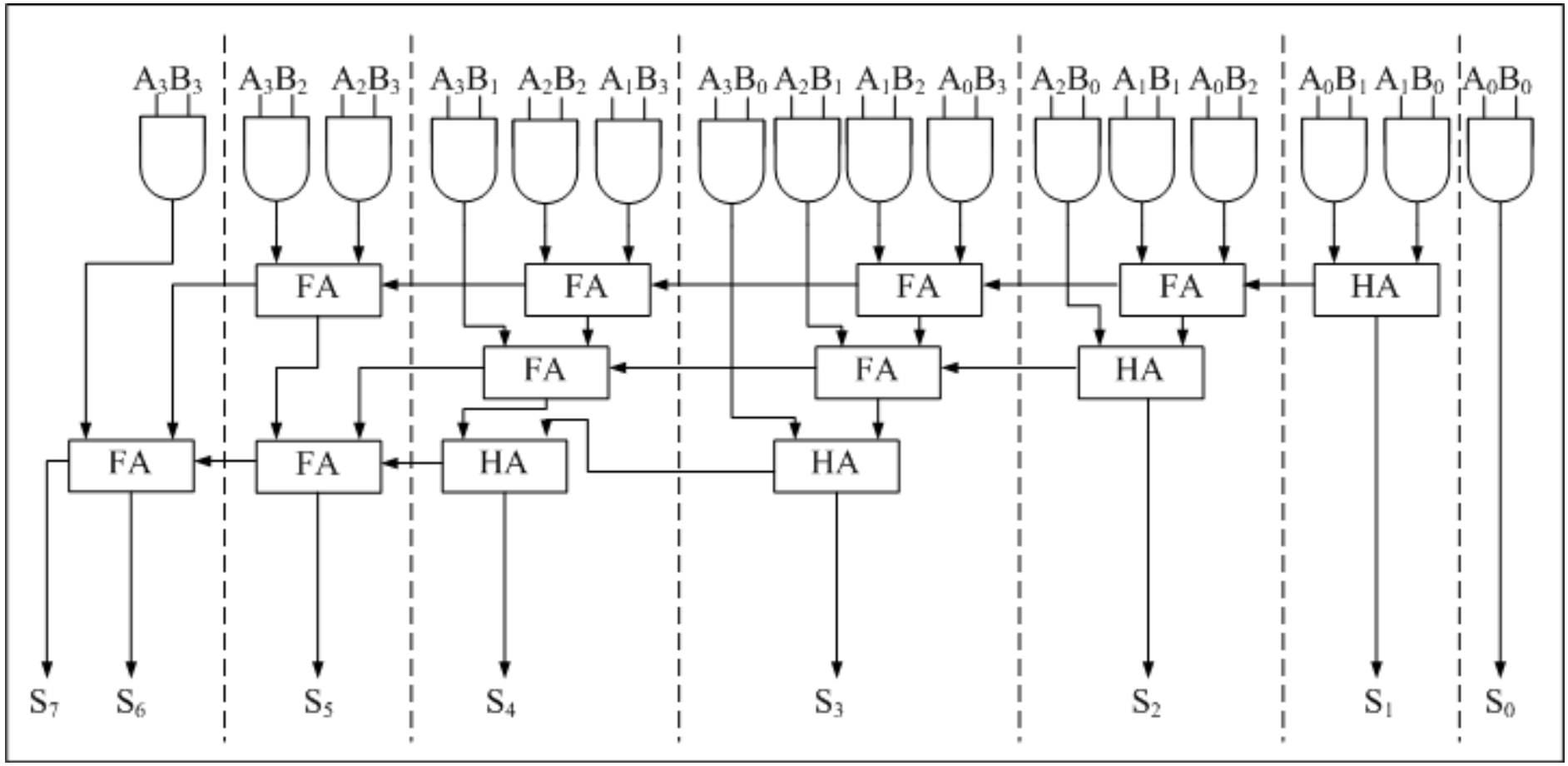
- The bits of each partial product are computed by multiplying two bits of the input.
- Since two-bit multiplication is the same as the logical AND operation, we can use AND gates to generate the partial products.

2X2 Binary Multiplication

- Here is a circuit that multiplies the two-bit numbers A1A0 and B1B0, resulting in the four-bit product P3-P0.
- For a 2×2 multiplier we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- The diagram on the next page shows how this can be extended to a four-bit multiplier, taking inputs A3-A0 and B3-B0 and outputting the product P7-P0.



A 4×4 binary multiplier

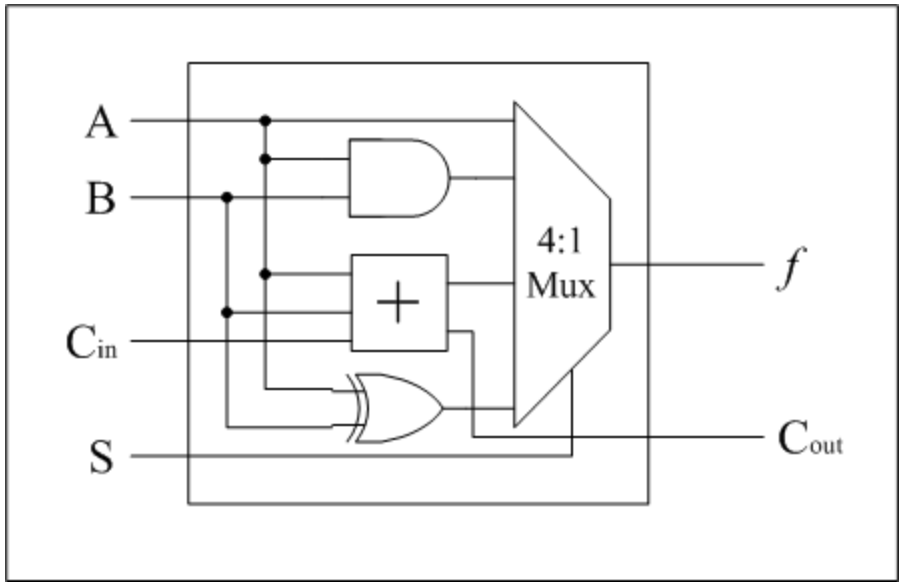


Complexity of multiplication circuits

- In general, when multiplying an *m*-bit number by an *n*-bit number:
  - There will be *n* partial products, one for each bit of the multiplier.
  - This requires *n*-1 adders, each of which can add *m* bits.
- The circuit for 32-bit or 64-bit multiplication would be huge!

ARITHMETIC LOGIC UNIT

A 1-Bit ALU



Operation S1 S0	Function
0 0	A
0 1	A•B
1 0	A + B
1 1	A XOR B

- The multiplexer selects either

  A, A•B, A + B or A XOR B

  depending on whether the value of *operation, S* is *00*, *01, 10 or 11.*

- To add an operation, the multiplexer has to be expanded

A 32-Bit ALU

- A full 32-bit ALU can be created by connecting adjacent 1-bit ALU's
- using the Carry in and carry out lines
- The carry out of the least significant bit can ripple all the way through the adder (*ripple carry adder*)
- Ripple carry adders are slow since the carry propagates from a unit to the next sequentially
- Subtraction can be performed by inverting the operand and setting the "CarryIn" input for the whole adder to 1 (i.e. using two's complement)

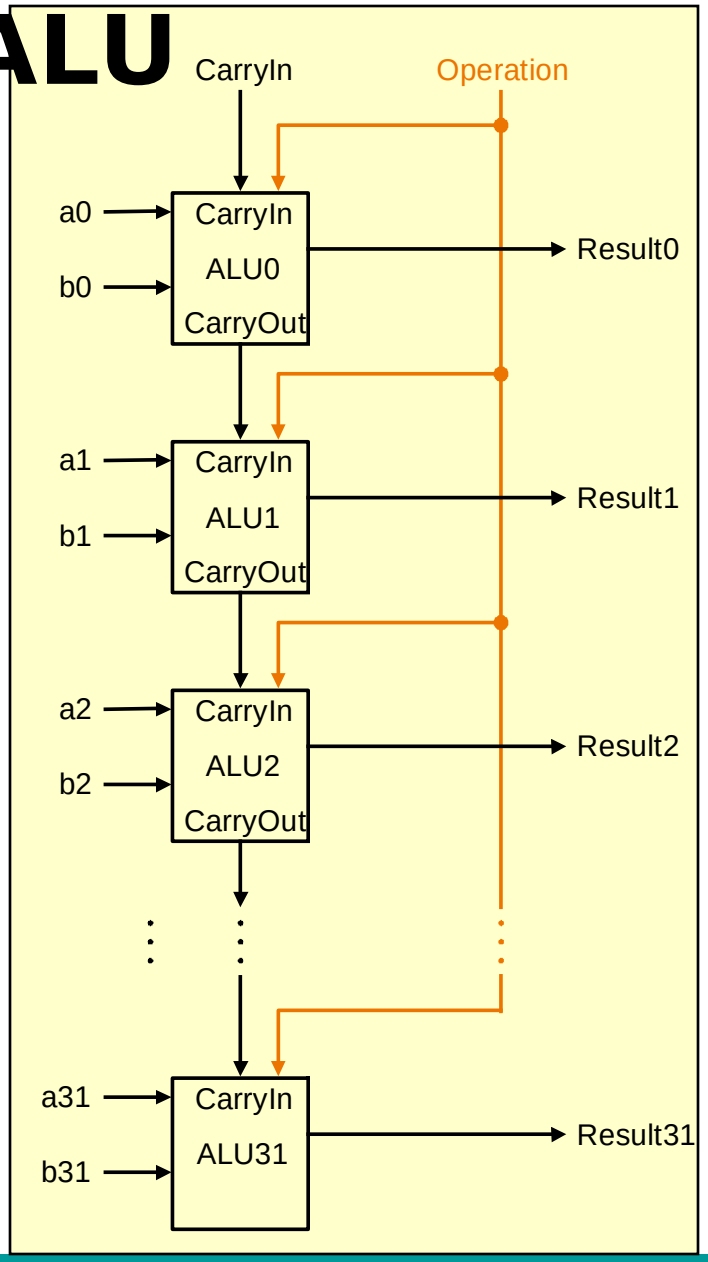
A 32-Bit ALU

1. A full 32-bit ALU can be created by connecting adjacent 1-bit ALU's using the Carry in and carry out lines

2. The carry out of the least significant bit
3. can ripple all the way through the adder (*ripple carry adder*)

4. Ripple carry adders are slow since the carry propagates from a unit to the next sequentially

5. Subtraction can be performed by inverting the operand and setting the "CarryIn" input for the whole adder to 1
6. (i.e. using two's complement)



Summary

- Adder and multiplier circuits reflect human algorithms for addition and multiplication.
- Adders and multipliers are built hierarchically.
  - We start with half adders and full adders and work our way up.
  - Building these circuits from scratch using truth tables and K-maps would be pretty difficult.
- Adder circuits are limited in the number of bits that can be handled. An overflow occurs when a result exceeds this limit.
- There is a tradeoff between simple but slow ripple carry adders and more complex but faster carry lookahead adders.
- Multiplying and dividing by powers of two can be done with simple shifts.

HARDWARE DESCRIPTION LANGUAGE

Design of a Half Adder (halfadd)

- Behavioural

```
LIBRARY ieee
USE ieee.std_logis_1164.all

ENTITY halfadd IS
    Port (      A, B        : IN      STD_LOGIC;
                sum, Cout   : OUT     STD_LOGIC);
END HA;

ARCHITECTURE behavioural OF halfadd IS
BEGIN
    Cout <= A AND B;
    sum <= A XOR B;
END behavioural;
```

Design of Full Adder (fulladd)

- Behavioural

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
        PORT (      Cin, x, y     : IN        STD_LOGIC ;
                    s, Cout       : OUT       STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE Behavioural OF fulladd IS
BEGIN
        s <= x XOR y XOR Cin ;
        Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END Behavioural ;
```

Design of Full Adder (FA)

- Structural (use halfadd)

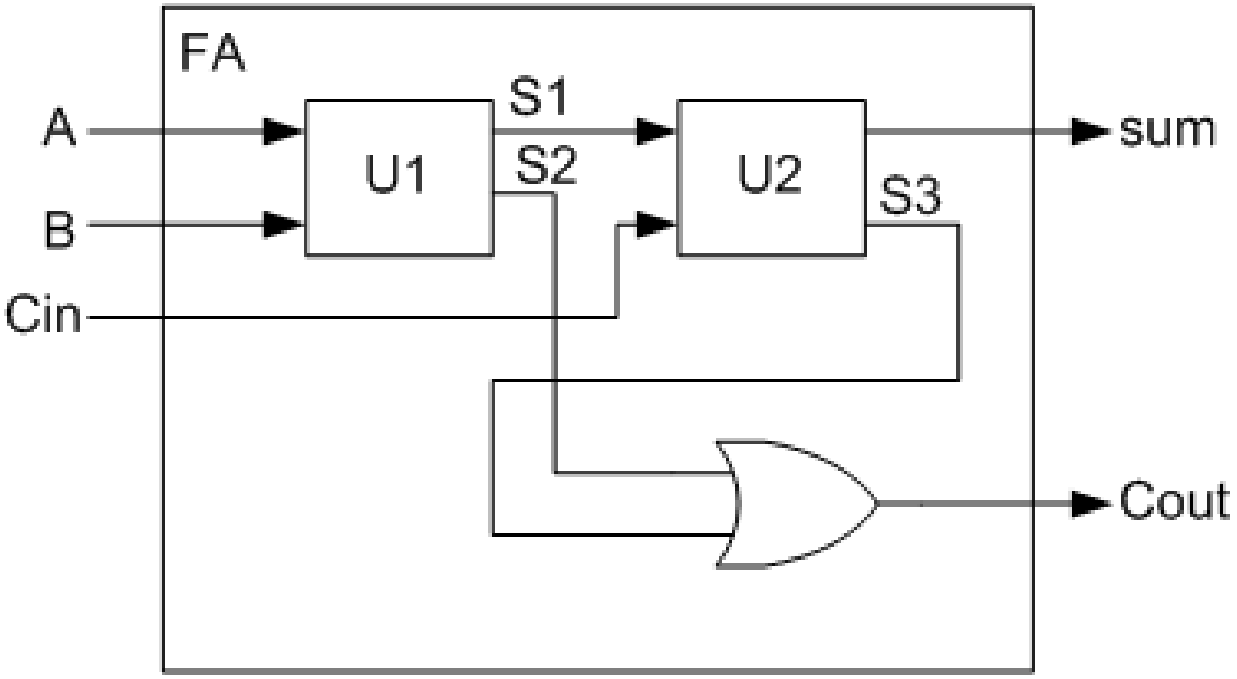
```
ENTITY FA IS
        PORT (      A, B, Cin    : IN         STD_LOGIC ;
                    sum, Cout    : OUT        STD_LOGIC ) ;
END FA;

ARCHITECTURE Structural OF FA IS
signal S1, S2, S3   : STD_LOGIC;
COMPONENT halfadd
        Port (      A, B         : IN         STD_LOGIC;
                    sum, Cout    : OUT        STD_LOGIC);
BEGIN
        U1 : halfadd             PORTMAP ( A, B, S1, S2);
        U2 : halfadd             PORTMAP ( S1, Cin, Sum, S3);

        Cout <= S2 OR S3;
END LogicFunc ;
```

Design of Full Adder (FA)

- Structural (use halfadd)



Design a 4-bit adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT (          Cin
            : IN    STD_LOGIC ;
            x3, x2, x1, x0          : IN
            STD_LOGIC ;
            y3, y2, y1, y0          : IN
            STD_LOGIC ;
            s3, s2, s1, s0          : OUT
            STD_LOGIC ;
            Cout                    : OUT
            STD_LOGIC ) ;
END adder4 ;
```

```
ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT (  Cin, x, y   : IN
        STD_LOGIC ;
                s, Cout     : OUT
        STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0,
    c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2
    ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3
    ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y =>
    y3, s => s3  ) ;
END Structure ;
```

Design a 4-bit adder (using Package)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
      PORT ( Cin                          : IN      STD_LOGIC ;
                        x3, x2, x1, x0    : IN      STD_LOGIC ;
                        y3, y2, y1, y0    : IN      STD_LOGIC ;
                        s3, s2, s1, s0    : OUT     STD_LOGIC ;
                        Cout                        : OUT     STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
      SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
      stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
      stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
      stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
      stage3: fulladd PORT MAP (
            Cin => c3, Cout => Cout, x => x3, y => y3, s => s3  ) ;
END Structure ;
```

Design a 4-bit adder

- Fulladd package

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT (   Cin, x, y   : IN      STD_LOGIC ;
                 s, Cout     : OUT     STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Designing ALU

- Functionality ALU

	Inputs	Outputs
Operation	S2S1S0	F
Clear	000	0000
B – A	001	B – A
A – B	010	A – B
ADD	011	A + B
XOR	100	A XOR B
OR	101	A OR B
AND	110	A AND B
Preset	111	1111

Designing ALU

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY alu IS
    PORT (s : IN STD_LOGIC_VECTOR(2
    DOWNTO 0) ;
        A, B : IN STD_LOGIC_VECTOR(3
    DOWNTO 0) ;
        F : OUT STD_LOGIC_VECTOR(3
    DOWNTO 0) ) ;
END alu ;
ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS ( s, A, B )
    BEGIN
    CASE s IS
    WHEN "000" =>
    <= "0000" ;
    WHEN "001" =>
        F <= B - A ;
    WHEN "010" =>
        F <= A - B ;
    WHEN "011" =>
        F <= A + B ;
    WHEN "100" =>
        F <= A XOR B ;
    WHEN "101" =>
        F <= A OR B ;
    WHEN "110" =>
        F <= A AND B ;
    WHEN OTHERS =>
        F <= "1111" ;
    END CASE ;
    END PROCESS ;
END Behavior ;
```

F