Digital Systems

Arithmetic

Introduction

- Digital circuits are frequently used for arithmetic operations
- Fundamental arithmetic operations on binary numbers and digital circuits which perform arithmetic operations will be examined.

Binary Addition

- Binary numbers are added like decimal numbers.
- In decimal, when numbers sum more than 9 a carry results.
- In binary when numbers sum more than 1 a carry takes place.
- Addition is the basic arithmetic operation used by digital devices to perform subtraction, multiplication, and division.

Binary Addition

- 0 + 0 = 0
- 1 + 0 = 1
- 1 + 1 = 0 + carry 1
- 1 + 1 + 1 = 1 + carry 1
- E.g.:

```
  1010    (10)          001 (1)
+1100     (12)        +101 (5)
10110     (22)        +111 (7)
                      1101 (13)
```

Representing Signed Numbers

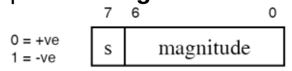
- Since it is only possible to show magnitude with a binary number, the sign (+ or –) is shown by adding an extra “sign” bit.
- A sign bit of 0 indicates a positive number.
- A sign bit of 1 indicates a negative number.
- The 2’s complement system is the most commonly used way to represent signed numbers.

Representing Signed Numbers

- So far, numbers are assumed to be unsigned (i.e. positive)
- How to represent signed numbers?
- Solution 1: **Sign-magnitude** - Use one bit to represent the **sign**, the remain bits to represent **magnitude**

+27 = 0001 1011 b

-27 = 1001 1011 b



  – Problem: need to handle sign and magnitude separately.

- Solution 2: **One’s complement** - If the number is negative, invert each bits in the magnitude

  +27 = 0001 1011 b

  -27 = 1110 0100 b

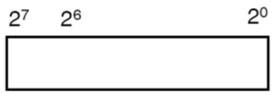
- Not convenient for arithmetic - add 27 to -27 results in $1111\ 1111_b$
  – Two zero values

Representing Signed Numbers

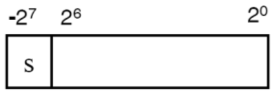
- Solution 3: **Two’s complement** - represent negative numbers by taking its magnitude, invert all bits and add one:

–	**Positive number**	**+27 = 0001 1011b**
–	**Invert all bits**	**1110 0100b**
–	**Add 1**	**-27 = 1110 0101b**

- Unsigned number



- Signed 2’s complement



$$x = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \bullet\bullet\bullet + b_1 2^1 + b_0 2^0$$

Examples of 2’s Complement

- A common method to represent *-ve* numbers:
  – use half the possibilities for positive numbers and half for negative numbers
  – to achieve this, let the MSB have a negative weighting
- Construction of 2's Complement Numbers
  – 4-bit example

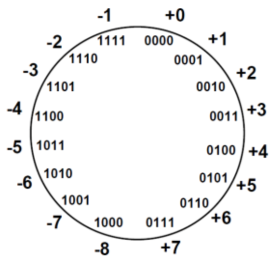
Decimal	2's Complement (Signed Binary)			
	-8	+4	+2	+1
5	0	1	0	1
-5	1	0	1	1
7	0	1	1	1
-3	1	1	0	1

Why 2's complement representation?

- If we represent signed numbers in 2's complement form, subtraction is the same as addition to negative (2's complemented) number.

```
    27  0001 1011 b
  - 17  0001 0001 b
  + 10 0000 1010 b

 +   27 0001 1011 b
 + - 17 1110 1111 b
 +   10 0000 1010 b
```



- Note that the range for 8-bit unsigned and signed numbers are different.
- 8-bit unsigned: **0 …… +255**
- 8-bit 2's complement signed number: **-128 …… +127**

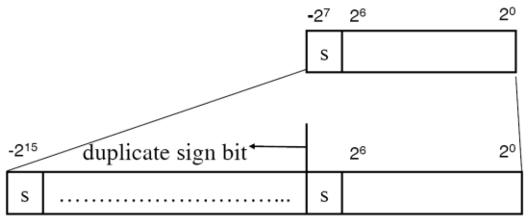
Comparison Table

- Note the "**wrap-around**" effect of the binary representation
  - i.e. The top of the table wraps around to the bottom of the table

Unsigned	Binary	2' comp
7	0111	7
6	0110	6
5	0101	5
4	0100	4
3	0011	3
2	0010	2
1	0001	1
0	0000	0
15	1111	-1
14	1110	-2
13	1101	-3
12	1100	-4
11	1011	-5
10	1010	-6
9	1001	-7
8	1000	-8

Sign Extension

- How to translate an 8-bit 2's complement number to a 16-bit 2's complement number?



- This operation is known as **sign extension**.

Sign Extension

- Sometimes we need to extend a number into more bits
- Decimal
  - converting 12 into a 4 digit number gives 0012
  - we add 0's to the left-hand side
- Unsigned binary
  - converting 0011 into an 8 bit number gives 00000011
  - we add 0's to the left-hand side
- For signed numbers we duplicate the sign bit (MSB)
- Signed binary
  - converting 0011 into 8 bits gives 00000011 (duplicate the 0 MSB)
  - converting 1011 into 8 bits gives 11111011 (duplicate the 1 MSB)
  - Called "**Sign Extension**"

Representing Signed Numbers

- In order to change a binary number to 2’s complement it must first be changed to 1’s complement.
  - To convert to 1’s complement, simply change each bit to its complement (opposite).
  - To convert 1’s complement to 2’s complement add 1 to the 1’s complement.
- A positive number is true binary with 0 in the sign bit.
- A negative number is in 2’s complement form with 1 in the sign bit.
- A number is negated when converted to the opposite sign.
- A binary number can be negated by taking the 2’s complement of it.

Signed Addition

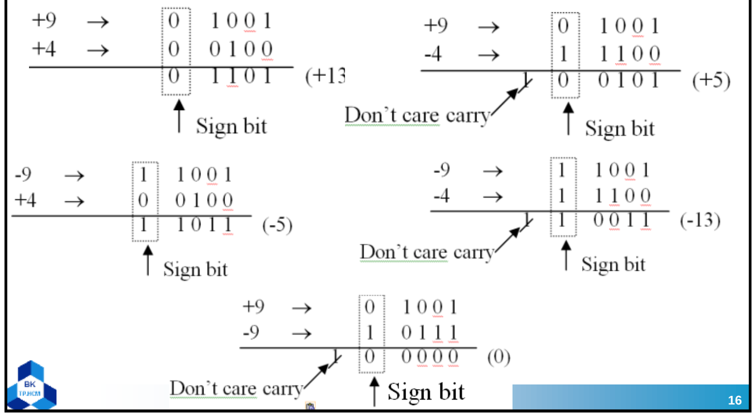
- The same hardware can be used for 2's complement signed numbers as for unsigned numbers
  - this is the main advantage of 2's complement form
- Consider 4 bit numbers:
  - the Adder circuitry will "think" the negative numbers are 16 greater than they are in fact
  - but if we take only the 4 LSBs of the result (i.e. ignore the carry out of the MSB) then the answer will be correct providing it is with the range: -8 to +7.
- To add 2 n-bit signed numbers without possibility of overflow we need to:
  - sign extend to n+1 bits
  - use an n+1 bit adder

Addition in the 2’s Complement System

- Perform normal binary addition of magnitudes.
- The sign bits are added with the magnitude bits.
- If addition results in a carry of the sign bit, the carry bit is ignored.
- If the result is positive it is in pure binary form.
- If the result is negative it is in 2’s complement form.

Addition in the 2’s Complement System

- Perform normal binary addition of magnitudes.



Subtraction in the 2's Complement System

- The number subtracted (subtrahend) is negated.
- The result is added to the minuend.
- The answer represents the difference.
- If the answer exceeds the number of magnitude bits an overflow results.

```
+9  →   0   1 0 0 1
+8  →   0   1 0 0 0
-------------------
        1   0 0 0 1
        ↑
        Sign bit
```

Multiplication and Division by 2^N

- In decimal, multiplying by 10 can be achieved by
  - shifting the number left by one digit adding a zero at the LS digit
- In binary, this operation multiplies by 2
- In general, left shifting by N bits multiplies by 2^N
  - zeros are always brought in from the right-hand end
  - E.g.

Binary	Decimal
1101	13
11010	26
110100	52

Multiplication of Binary Numbers

- This is similar to multiplication of decimal numbers.
- Each bit in the multiplier is multiplied by the multiplicand.
- The results are shifted as we move from LSB to MSB in the multiplier.
- All of the results are added to obtain the final product.

```
       1001    (9)
  x    1011    (11)
  ---------
       1001
      1001
     0000
    1001
  ---------
    1100011    (99)
```

Binary Division

- This is similar to decimal long division.
- It is simpler because only 1 or 0 are possible.
- The subtraction part of the operation is done using 2's complement subtraction.
- If the signs of the dividend and divisor are the same the answer will be positive.
- If the signs of the dividend and divisor are different the answer will be negative.

Summary of Signed and Unsigned Numbers

Unsigned	Signed
MSB has a positive value (e.g. +8 for a 4-bit system)	MSB has a negative value (e.g. -8 for a 4-bit system)
The carry-out from the MSB of an adder can be used as an extra bit of the answer to avoid overflow	To avoid overflow in an adder, need to sign extend and use an adder with one more bit than the numbers to be added
To increase the number of bits, add zeros to the left-hand side	To increase the number of bits, sign extend by duplicating the MSB
Complementing and adding 1 converts X to (2N - X)	Complementing and adding 1 converts X to -X

BCD Addition

- When the sum of each decimal digit is less than 9, the operation is the same as normal binary addition.
- When the sum of each decimal digit is greater than 9, a binary 6 is added. This will always cause a carry.

47		0100	0111	→	47 BCD
+35	+	0011	0101	→	35 BCD
82		0111	1100	→	invalid
	+	1	0110	→	+ 6
		1000	0010	→	valid

Hexadecimal Arithmetic

- Hex addition:
  - Add the hex digits in decimal.
  - If the sum is 15 or less express it directly in hex digits.
  - If the sum is greater than 15, subtract 16 and carry 1 to the next position.
- Hex subtraction – use the same method as for binary numbers.
- When the MSD in a hex number is 8 or greater, the number is negative. When the MSD is 7 or less, the number is positive.

```
  3AF
+23C
  5EB
```

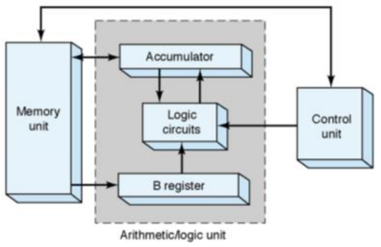
$592_{(16)} - 3A5_{(16)}$ FFF – 3A5 = C5A

C5A + 1 = C5B is 2's complement of 3A5

```
       592
+      C5B
     ✗ 1ED
```

Arithmetic Circuits

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



Arithmetic Circuits

- Typical sequence of operations:
  - Control unit is instructed to add a specific number from a memory location to a number stored in the accumulator register.
  - The number is transferred from memory to the B register.
  - Number in B register and accumulator register are added in the logic circuit, with sum sent to accumulator for storage.
  - The new number remains in the accumulator for further operations or can be transferred to memory for storage.

Binary Addition

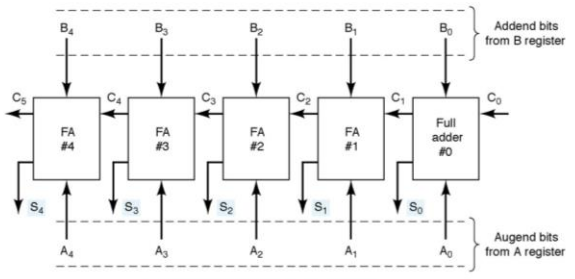
- Recall the binary addition process

```
      A 1 0 0 1
  +   B 0 0 1 1
      S 1 1 0 0
```

- LS Column has 2 inputs 2 outputs
  - Inputs: A0 B0
  - Outputs: S0 C1
- Other Columns have 3 inputs, 2 outputs
  - Inputs: An Bn Cn
  - Outputs: Sn Cn+1
  - We use a "half adder" to implement the LS column
  - We use a "full adder" to implement the other columns
  - Each column feeds the next-most-significant column.

Parallel Binary Adder

- The A and B variables represent 2 binary numbers to be added. The C variables are the carries. The S variables are the sum bits.



Sum appears at S_4, S_3, S_2, S_1, S_0 outputs.

Half Adder

- Truth Table

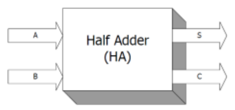
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

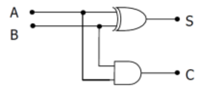
- Boolean Equations

$$S = \overline{A}B + A\overline{B} = A \oplus B$$

$$C = AB$$

- Implementation

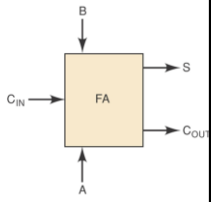




Full Adder

- Truth Table

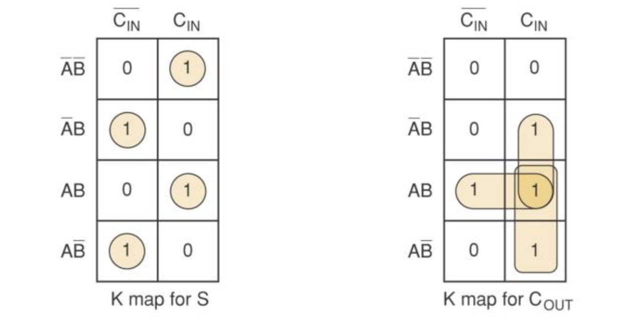
Augend bit input	Addend bit input	Carry bit input	Sum bit output	Carry bit output
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



- Boolean Equations

$$S = \overline{A}.\overline{B}.C_i + \overline{A}.B.\overline{C_i} + A.\overline{B}.\overline{C_i} + A.B.C_i$$
$$= A \oplus B \oplus C_i$$
$$C_o = \overline{A}BC_i + A\overline{B}C_i + AB\overline{C_i} + ABC_i$$
$$= AB + AC_i + BC_i$$
$$= AB + C_i(A+B)$$

K maps for the full-adder outputs.



	$\overline{C}_{IN}$	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)

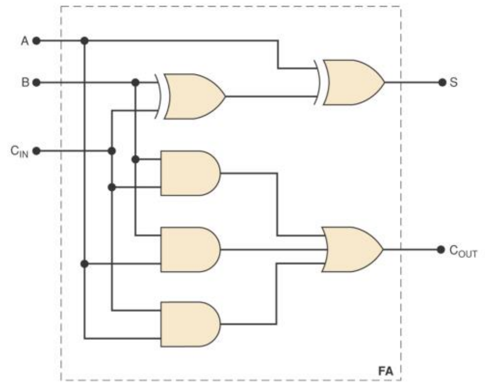
	$\overline{C}_{IN}$	C_{IN}
$\overline{A}\overline{B}$	0	0
$\overline{A}B$	0	1
AB	1	1
$A\overline{B}$	0	1

K map for C_{OUT}

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

(b)

Circuitry for a full adder

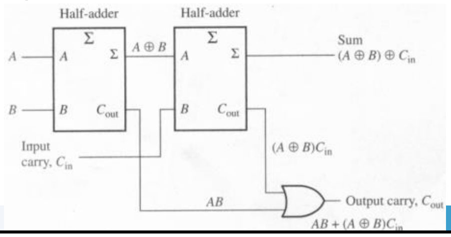


Full Adder from Half Adders

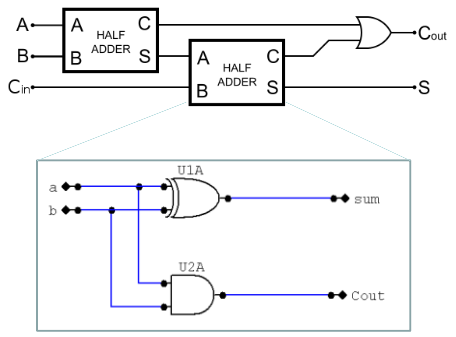
- Truth Table

A	B	HA_s	HA_c	C_i	S	C_o
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	1	1	0	0	1	0
0	1	1	0	1	0	1
1	0	1	0	0	1	0
1	0	1	0	1	0	1
1	1	0	1	0	0	1
1	1	0	1	1	1	1

- Boolean Equations



Adder Example



Hierarchy

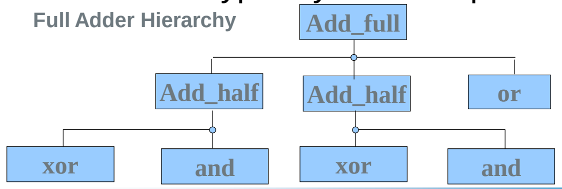
- Any Verilog design you do will be a module
- This includes testbenches!

- Interface ("black box" representation)
  - Module name, ports
- Definition
  - Describe functionality of the block
  - Includes interface
- Instantiation
  - Use the module inside another module

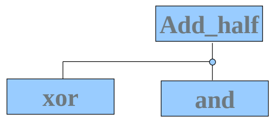
Hierarchy

- Build up a module from smaller pieces
  - Primitives
  - Other modules (which may contain other modules)
- Design: typically top-down
- Verification: typically bottom-up

Full Adder Hierarchy



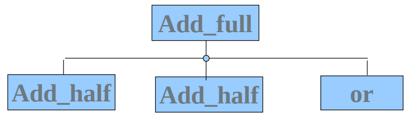
Add_half Module



```
module Add_half(c_out, sum, a, b);
   output sum, c_out;
   input a, b;

   xor sum_bit(sum, a, b);
   and carry_bit(c_out, a, b);
endmodule
```

Add_full Module



```
module Add_full(c_out, sum, a, b, c_in) ;
   output sum, c_out;
   input a, b, c_in;
   wire w1, w2, w3;

   Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
   Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
   or carry_bit(c_out, w2, w3);
endmodule
```

Can Mix Styles In Hierarchy!

```
module Add_half_bhv(c_out, sum, a, b);
   output reg sum, c_out;
   input a, b;
   always @(a, b) begin
      sum = a ^ b;
      c_out = a & b;
   end
endmodule

         module Add_full_mix(c_out, sum, a, b, c_in) ;
            output sum, c_out;
            input a, b, c_in;
            wire w1, w2, w3;
            Add_half_bhv AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
            Add_half_bhv AH2(.sum(sum), .c_out(w3),.a(c_in), .b(w1));
            assign c_out = w2 | w3;
         endmodule
```

Full Adder: RTL/Dataflow

```
module fa_rtl (A, B, CI, S, CO) ;

   input A, B, CI ;
   output S, CO ;

   // use continuous assignments
   assign S = A ^  B ^ CI;
   assign C0 = (A & B) | (A & CI) | (B & CI);

endmodule
```

Full Adder: Behavioral

- Circuit “reacts” to given events (for simulation)
  - Actually list of signal changes that affect output

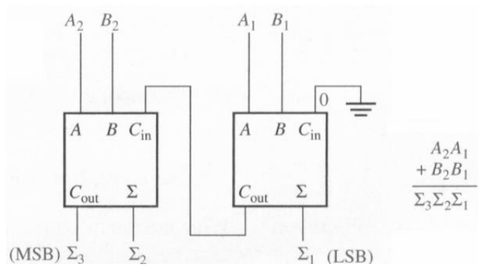
```
module fa_bhv (A, B, CI, S, CO) ;

   input A, B, CI;
   output S, CO;
   reg S, CO;          // explained in later lecture – “holds” values

   // use procedural assignments
   always@(A or B or CI)
     begin
        S = A ^ B ^ CI;
        CO = (A & B) | (A & CI) | (B & CI);
     end
endmodule
```

Parallel Adder

- Uses 1 full adder per bit of the numbers
- The carry is propagated from one stage to the next most significant stage
  - takes some time to work because of the carry propagation delay which is n times the propagation delay of one stage.

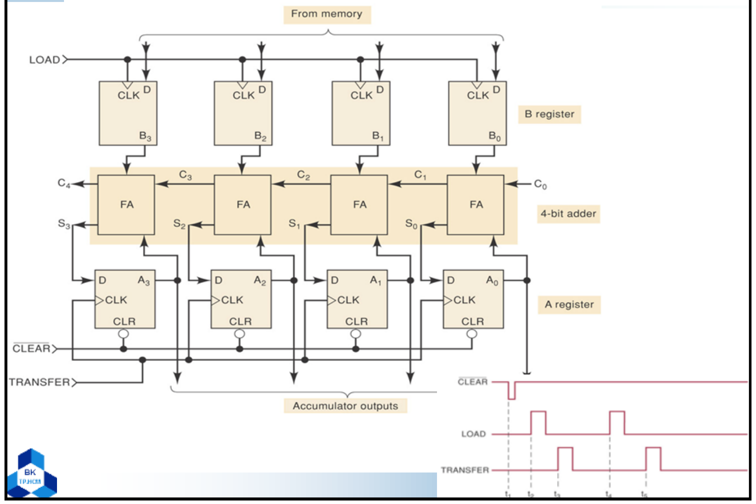


Complete Parallel Adder With Registers

- Register notation – to indicate the contents of a register we use brackets:

  [A]=1011 is the same as A_3=1, A_2=0, A_1=1, A_0=1
- A transfer of data to or from a register is indicated with an arrow
  - [B]→[A] means the contents of register B have been transferred to register A.
- Eg.: 1001 + 0101 using the parallel adder:
  - t1 : A CLR pulse is applied
  - t2 : 1001 from mem-> B
  - t3 : 1001 + 0000 -> A
  - t4 : 0101 from mem-> B
  - t5 : The sum outputs -> A
  - The sum of the two numbers is now present in the accumulator.

Complete Parallel Adder With Registers



Carry Propagation

- Parallel adder speed is limited by carry propagation (also called carry ripple).
- Carry propagation results from having to wait for the carry bits to "ripple" through the device.
- Additional bits will introduce more delay.
- Various techniques have been developed to reduce the delay. The **look-ahead carry** scheme is commonly used in high speed devices.

Design a carry look-ahead adder

$C1 = [A0\,B0 + A0\,C0 + B0\,C0]$

$C2 = (A1\,B1 + A1\,C1 + B1\,C1) = A1\,B1 + (A1 + B1)[C1]$

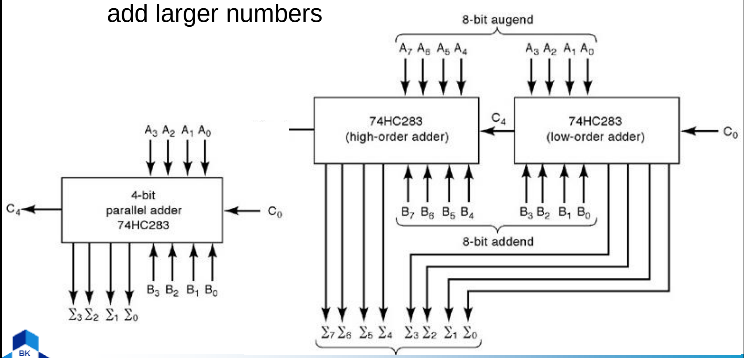
$C3 = A2\,B2 + A2\,C2 + B2\,C2$

$C3 = A2\,B2 + (A2+B2)\,\{A1B1 + (A1+B1)\,[A0B0 + B0C0 + A0C0]\}$

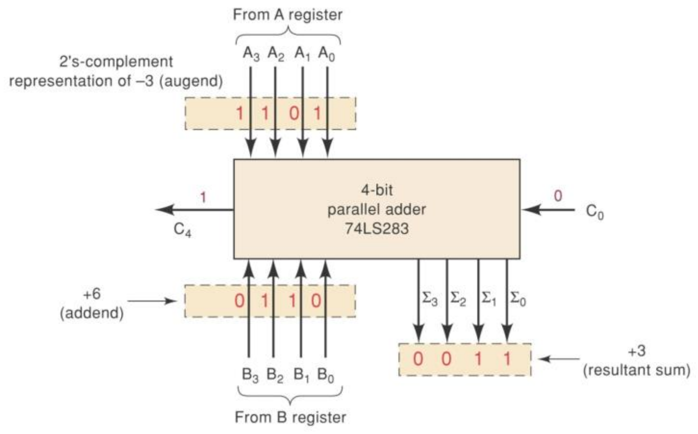
Final expression for C3 can be put into S-of-P form by multiplying all terms out. This results in a circuit with TWO levels of gating. The arrangement of Figure 6.9 requires that A0, B0, and C0 propagate through as many as 6 levels of gates before producing C3.

Integrated Circuit Parallel Adder

- The most common parallel adder is a 4 bit device with 4 interconnected FAs and look-ahead Carry circuits.
- Parallel adders may be cascaded together as shown to add larger numbers

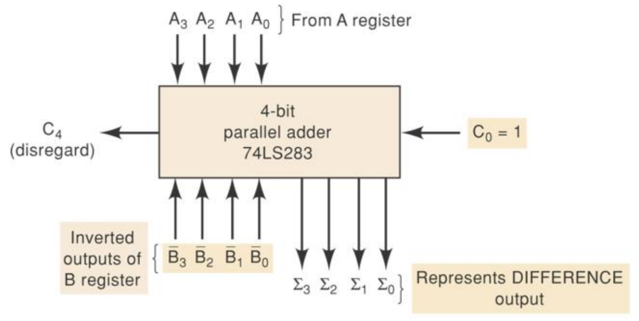

8-bit augend
$A_7\ A_6\ A_5\ A_4$
$A_3\ A_2\ A_1\ A_0$
74HC283 (high-order adder)
C_4
74HC283 (low-order adder)
C_0
$A_3\ A_2\ A_1\ A_0$
C_4
4-bit parallel adder 74HC283
C_0
$B_7\ B_6\ B_5\ B_4$
$B_3\ B_2\ B_1\ B_0$
8-bit addend
$B_3\ B_2\ B_1\ B_0$
$\Sigma_3\ \Sigma_2\ \Sigma_1\ \Sigma_0$
$\Sigma_7\ \Sigma_6\ \Sigma_5\ \Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1\ \Sigma_0$

Parallel adder used to add and subtract numbers in 2's-complement system.



2's Complement Addition using 1's Complement Operands

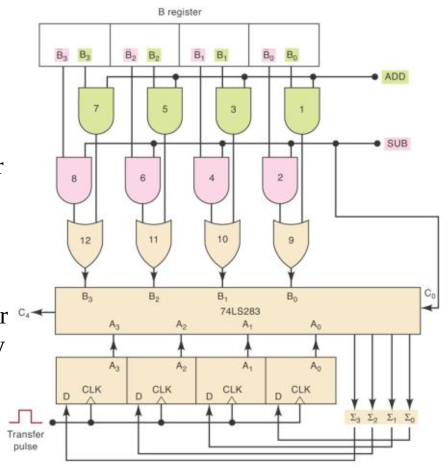
Parallel adder used to perform subtraction ($A – B$) using the 2's-complement system. The bits of the subtrahend (B) are inverted (1's complement), and $C_0 = 1$ to produce the 2's complement


$A_3\ A_2\ A_1\ A_0$
From A register
C_4 (disregard)
4-bit parallel adder 74LS283
$C_0 = 1$
Inverted outputs of B register
$\bar{B}_3\ \bar{B}_2\ \bar{B}_1\ \bar{B}_0$
$\Sigma_3\ \Sigma_2\ \Sigma_1\ \Sigma_0$
Represents DIFFERENCE output

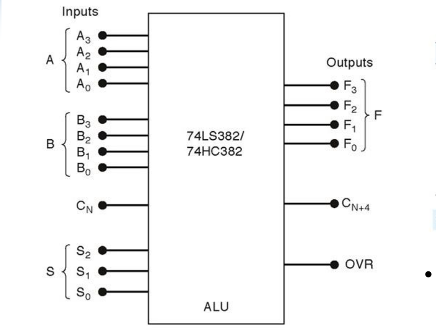
Parallel adder/subtractor using the 2's-complement system

ADD = 1, SUB = 0:
B register passes to adder and Carry in = 0

ADD = 0, SUB = 1:
Complement of B register passes to adder and Carry in = 1



ALU Integrated Circuits



A = 4-bit input number
B = 4-bit input number
C_N = carry into LSB position
S = 3-bit operation select inputs
F = 4-bit output number
C_{N+4} = carry out of MSB position
OVR = overflow indicator

Function Table

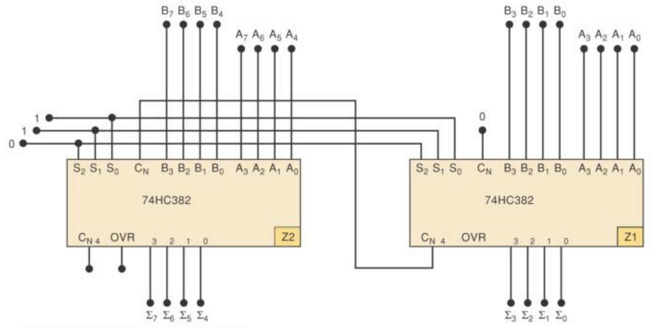
S_2	S_1	S_0	Operation	Comments
0	0	0	CLEAR	$F_3F_2F_1F_0 = 0000$
0	0	1	B minus A	} Needs $C_N = 1$
0	1	0	A minus B	
0	1	1	A plus B	Needs $C_N = 0$
1	0	0	$A \oplus B$	Exclusive-OR
1	0	1	A + B	OR
1	1	0	AB	AND
1	1	1	PRESET	$F_3F_2F_1F_0 = 1111$

Notes: S inputs select operation.
OVR = 1 for signed-number overflow.

(b)

- ALUs can perform different arithmetic and logic functions as determined by a binary code on the function select inputs.

Two 74HC382 ALU chips connected as an eight-bit adder



Notes: Z1 adds lower-order bits.
Z2 adds higher-order bits.
Σ_7–Σ_0 = 8-bit sum.
OVR of Z2 is 8-bit overflow indicator.