Chapter 7

Digital Arithmetic and Arithmetic Circuits

Basic Digital Arithmetic

- Signed binary number
  - A binary number of fixed length whose sign (+/-) is represented by one bit (usually MSB) and its magnitude by the remaining bits
- Unsigned binary number
  - A binary number of fixed length whose sign is not specified by a bit
  - All bits are magnitude and the sign is assumed +

Unsigned Binary Arithmetic

- Sum
  - Result of an addition operation of two (or more) binary numbers (operands)
    - Augend
    - Addend
- Carry
  - A digit (or bit) that is carried over to the next most significant bit during an N-bit addition operation
  - The carry bit is a 1 if the result was too large to be expressed in N bits

Basic Rules (Unsigned)

- One bit unsigned addition

Cin	A		B		Sum	Cout
	0	+	0	=	0	0
	0	+	1	=	1	0
	1	+	1	=	0	1
1	+ 1	+	1	=	1	1

Binary Addition (example 1)

```
  1
 10010
+ 1010
 11100
```


Binary Addition (example 2)

```
 1 11111
  10101110
+ 10010011
 101000001
```

Basic Subtraction

- X = A – B
  - A = minuend
  - B = subtrahend
  - X = difference or result
  - Requires a borrow bit if A < B
- Other forms of subtraction exist
  - 2's complement addition (used in microprocessors)

Basic Subtraction Rules

- One bit unsigned subtraction

Borrow	A		B		Diff	
	0	-	0	=	0	
	1	-	0	=	1	
	1	-	1	=	0	
1	0	-	1	=	1	($2_{10} - 1_{10} = 1_{10}$)

Binary Subtraction (example 1)

```
  1110        110(10)   Borrow Stage
- 1001      - 100  1
------      -------
     ?        010  1

(14 – 9 = 5)
```


Binary Subtraction (example 2)

```
  10000      0111(10)   Borrow ripples to LSB
-   101    -   10  1
------     --------
     ?       101  1

(16 – 5 = 11)
```


Signed Binary Numbers

- Sign bit
  - A bit (usually the MSB) that indicates whether a number is positive(=0) or negative (=1)
- Magnitude Bits
  - The bits of a signed binary number that tell how large it is in value

Note:
Positive numbers are the same in all three forms

Signed Binary Numbers

- True Magnitude Form
  - A form of signed binary whose magnitude bits are the TRUE binary form (not complements)
- 1's Complement
  - A form of signed binary in which negative numbers are created by complementing all bits
- 2's Complement
  - A form of signed binary number in which the negative numbers are created by complementing all the bits and adding a 1 (1's Complement + 1)

True Magnitude Form

- MSB is the sign bit (Negative: S = 1)
  - Other bits are the magnitude
  - 5-bit number examples:

$+25_{10}$ = 011001
-25_{10} = 111001 (Same as +25 with S=1)

$+12_{10}$ = 001100
-12_{10} = 101100 (Same as +12 with S=1)

1's Complement Form

1's Complement Form
A form of binary number in which negative numbers are generated by complementing all bits of a number including the sign bit

- MSB is the sign bit (Negative: S = 1)
  - Other bits are the magnitude
  - 8-bit number examples:

$+57_{10}$ = 00111001
-57_{10} = 11000110 (All bits inverted)

$+72_{10}$ = 01001000
-72_{10} = 10110111 (All bits inverted)

2's Complement Form

2's Complement Form
A form of binary number in which negative numbers are generated adding 1 to the 1's complement form of the number

- Used in μP arithmetic
- A negative number in 2's complement form can be made positive by 2's complementing it again

```
+57 = 00111001
-57 = 11000110 (1's comp)
    +        1
      11000111 (2's comp)

+72 = 01001000
-72 = 10110111 (1's comp)
    +        1
      10111000 (2's comp)
```


Signed Binary Addition

- Done in the same way as unsigned addition except…
  - Both operands must have the same number of magnitude bits
  - Both operands must have a sign bit
- Positive: Sign = 0

```
+30:   00011110
+75: + 01001011
+105:  01101001
```


Signed Binary Subtraction

- Using complement notation, we can add a negative number rather than subtracting a positive number
  - Same circuitry for both operations
  - This does not work for true magnitude numbers

1's Complement Subtraction

- Add the 1's complement and then add any carry

$+80_{10}$ = 01010000
$+65_{10}$ = 01000001
-65_{10} = 10111110 (1's comp)

```
 80    01010000
-65  + 10111110
   1  00001110
    +         1 (End around carry)
+15    00001111
```

2's Complement Subtraction

- Add the 2's complement to the minuend
- If a carry results, discard it

$+80_{10}$ = 01010000
$+65_{10}$ = 01000001
-65_{10} = 10111110 (1's complement)
+ 1
-65_{10} = 10111111 (2's complement)

```
 80     01010000
-65   + 10111111
+15   1 00001111 (Discard the carry)
```


Negative Sum or Difference

- If True Magnitude Form is used for subtraction, results will be incorrect
- If the results from a 1's complement or 2's complement is negative (S=1), the magnitude is found by taking the complement of the result

Negative Sum or Difference

```
+65:    0100 0001
-80:  + 1011 0000 (2's C.)
        1111 0001 (Negative sum)
        0000 1110 (Invert)
      +         1 (Add 1)
        0000 1111 (Final Answer: 15, negative)
```


Range of Signed Numbers

- Range of positive numbers is 0 to $2^N - 1$ for an N-bit magnitude
- Range of negative numbers is -1 to -2^N for an N-bit magnitude
- Example: 8-bit range (i.e. 7-bit magnitude)
  $(-2^7 < x < 2^7 - 1)$ or $(-128 < x < 127)$

Exercise

- Write -16_{10} as an 8-bit 2's complement number

```
+16 = 00010000
-16 = 11101111 (1'S C.)
    +        1 (Add 1)
      11110000 (2's C.)
```


Sign Bit Overflow

- Overflow
  - A erroneous carry into the sign bit of a signed binary number that results from a sum or difference that is larger than can be represented by the magnitude bits
- Results in a false positive or a false negative number

False Negative Overflow

- 8-bit addition
- Two positive numbers added with a result greater than +127 for 8-bit numbers causes an overflow

```
+75:   01001011
+96: + 01000001
       10101011 (negative - False)
```


False Positive Overflow

- Addition of two 8-bit negative numbers
- Two positive numbers added with a result greater than +127 for 8-bit numbers causes an overflow

```
-80:   10110000 (2's C.)
-65: + 10111111 (2's C.)
       01101111 (positive - False)
```

A sum of 2 positive numbers is always positive. A sum of 2 negative numbers is always negative. Any 2's complement arithmetic which contradicts this has produced an overflow in the sign bit.

Hexadecimal Addition

- Similar to decimal addition with additional digits A – F

  F + 1 = 10
  F + F = 1E
  F + F + 1 = 1F

- Easier than working with large binary numbers

BCD Codes

- Binary Coded Decimal
  - A code used to represent each decimal digit of a number by a 4-bit binary value
  - Valid Digits for 0-9 are (0000 to 1001) the binary codes 1010 to 1111 are invalid
  - Called an 8421 Code due to the decimal weight of each bit position

BCD (examples)

4987_{10} = 0100 1001 1000 0111 (BCD)

84_{10} = 1000 0100 (BCD)

Each digit is a 4 Bit Binary group

Gray Code

- A binary code that progresses such that only one bit changes between two successive codes
- Generated as
  - g3 = b3
  - g2 = b3 xor b2
  - g1 = b2 xor b1
  - g0 = b1 xor b0
- Very useful in hardware design

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

ASCII Code

- American Standard Code for Information Interchange
- A seven bit alphanumeric code used to represent text letters, numerals, punctuation, and special controls
- An expanded 8 bit form is often used also
  - Allows for some graphical characters

Binary Adders

- Half Adder(HA): A circuit that will add two bits and produce a sum and carry result
- Full Adder(FA): A circuit that will add a carry bit from another HA or FA (previous stage) and two operand bits to produce a sum and carry result

Basic HA Addition

- One bit addition rules
  - Three possible combinations

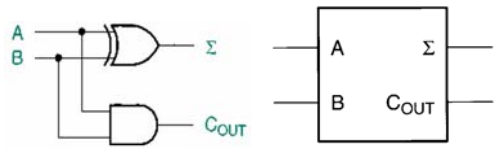
0 + 0 = 00
0 + 1 = 01
1 + 1 = 10

HA Circuit

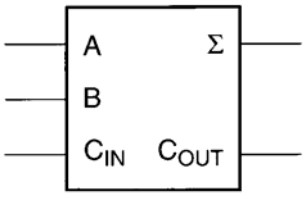
- Basic Equations
  (S = Sum, C = Carry)
  - S = A xor B
  - C = A and B

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

HA Circuit and Symbol



Full Adder



Full Adder

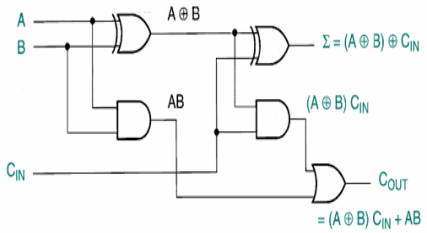
- Adds a Cin to the HA
- Basic Equations:

  Cout = (A xor B)Cin + AB

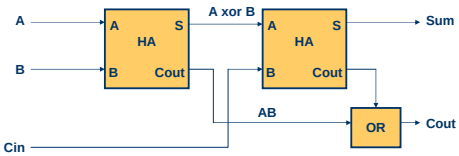
  S = (A xor B) xor Cin

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

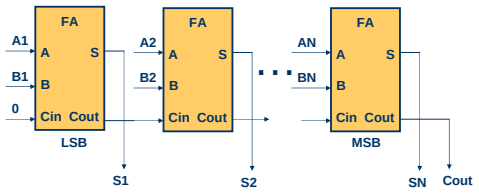
FA Circuit



FA Circuit (Using HAs)



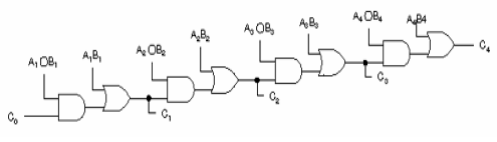
Parallel (N-bit) Binary Adder



Ripple Carry

- In the N-Bit Parallel Adder (FA Stages) the C_{OUT} is generated by the last stage (FA_N)
- This is called a Ripple Carry Adder because the final C_{OUT} (Last Stage) is based on a ripple through each stage by C_{IN} at the LSB Stage

Ripple Carry



Text: figure 7.11

Ripple Carry

- Each Stage will have a propagation delay on the C_{IN} to C_{OUT} of one AND gate and one OR gate
- A 4-Bit ripple carry adder will then have a propagation delay on the final C_{OUT} of 4 X 2 = 8 gates
- A 32 Bit adder such as in a µP in a PC could have a delay of 64 Gates

Fast Carry (or Look-Ahead Carry)

- A combinational network that generates the final C_{OUT} directly from the operand bits (A_1 to A_N, B_1 to B_N)
- It is independent of the operations of each FA Stage (unlike the ripple carry)

Fast (Look-Ahead) Carry

- Has a small propagation delay compared to the ripple carry
- Delay is 3 gates for a 4-bit adder compared to 8 for the ripple carry

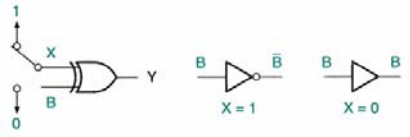
Subtractor (2's Complement)

- Subtraction using 2's complement addition allows use of parallel FAs
- The subtract operation involves adding the inverse of the subtrahend to the minuend and then add a 1

Subtraction (2's Complement)

- Difference = A - B = A + !B + 1
- This operation can be done in a parallel N-Bit FA by Inverting B_1 through B_N and connecting C_{IN} at the LSB Stage to +5V (adding 1)
- The circuit can be modified to allow either the ADD or SUBTRACT operation to be performed

XOR as Programmable Inverter



XOR as a Programmable Inverter – (See Text pp.381-382)

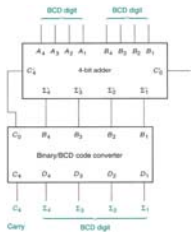
Adder / Subtractor

- See Dueck for Figure 7.15

BCD Adder (1)

- A parallel adder whose output sum is in groups of 4 bits each representing a BCD (8421) digit
- Basic design is a 4-bit binary parallel adder to generate a 4-bit sum of A + B
- Sum is input to the four bit input of a B_{IN} to BCD code converter

BCD Adder (2)



- Standard binary adder with a code converter
- If the sum is greater than 9, add 6 to convert to BCD
- Greater than 9 if Σ1 and Σ3 or Σ2

BCD Code Converter

- Code converter design is based on the 4-bit adder used with Table 7.11 in the text
- The complete design is shown in Fig. 7.26
- The A_i inputs of the code converter adder are fixed to be either a 0000 (C =0) or 0110 (C=1). The 0110 corrects binary overflow to BCD

Entity Example

2-to-4 decoder

```
ENTITY decode1 IS
    PORT(D1,D0  : IN BIT;
         Y0,Y1,Y2,Y3 : OUT BIT);
END decode1;
```


Architecture Example

```
ARCHITECTURE behavioral OF decode1 IS
BEGIN
    Y0 <= (not D1) and (not D0);
    Y1 <= (not D1) and (D0);
    Y2 <= (D1) and (not D0);
    Y3 <= (D1) and (D0);
END decoder;
```

Another Entity Format

```
ENTITY decode1 IS
  PORT(D : IN  STD_LOGIC_VECTOR(1 downto 0);
       Y : OUT STD_LOGIC_VECTOR(3 downto 0));
END decode1;
```


Another Entity Format

- This format groups signals of similar purpose
- into a *bus* (or *vector*)

- Instead of D1, D0 we use D(1 downto 0)
- Instead of Y3, Y2, Y1, Y0, we use Y(3 downto 0)

Selected Signal Assignments

- Uses a VHDL construct called
  `WITH SELECT`
- Basic Format
  `WITH (my_vector) SELECT`
- The selected signal state us used to determine the output changes

Architecture Using Selected Signal Assignment

```
ARCHITECTURE behavioral OF decode1 IS
BEGIN
    WITH (D) SELECT
        Y <= "0001" WHEN "00",
             "0010" WHEN "01",
             "0100" WHEN "10",
             "1000" WHEN "11",
             "0000" WHEN OTHERS;
END decoder;
```


Seven Segment Decoder Entity

```
ENTITY bcd_7seg IS
      PORT( d3, d2, d1, d0 : IN  BIT;
            a,b,c,d,e,f,g  : OUT BIT);
END bcd_7seg;
-- Defines binary inputs d0 to d3
-- Defines SS outputs a to g
```


Seven Segment Decoder (CA) Architecture

```
ARCHITECTURE seven_segment OF bcd_7seg IS
    SIGNAL input  : BIT_VECTOR(3 downto 0);
    SIGNAL output : BIT_VECTOR(6 downto 0);
BEGIN
    input <= D3 & D2 & D1 & D0

-- Uses two intermediate signals called
-- input and output (internal no pins)
-- Creates an array by using the concatenate
-- operator (&) In this case input(3) <= D3,
-- input(2) <= D2 etc.
```

SS (CA) Architecture Internal States

```
WITH input SELECT
    output <= “0000001” WHEN “0000”,
    output <= “1001111” WHEN “0001”,
    output <= “0010010” WHEN “0010”,
    output <= “0000110” WHEN “0011”,
    output <= “1001100” WHEN “0100”,
    output <= “0100100” WHEN “0101”,
              ...
    output <= “1111111” WHEN OTHERS;
```


SS Architecture Outputs

```
        a <= output(6);
        b <= output(5);
        c <= output(4);
        d <= output(3);
        e <= output(2);
        f <= output(1);
        g <= output(0);
END seven_segment;
```

Intermediate signal output(6 downto 0) is mapped to ports a - g

VHDL Sequential Process

- A *process* is a VHDL construct which encloses statements which are to be evaluated sequentially
- A process is executed when a signal in a *sensitivity list* changes

```
PROCESS(Sensitivity List)
BEGIN
    Sequential Statements;
END PROCESS;
```


Ripple Blanking Process (3)

- Process steps are evaluated in order:
  - First `IF-THEN` statements
  - Then `CASE` statements
  - and so on until `END PROCESS;`

IF-THEN-ELSE

- IF-THEN-ELSE statements are used for conditional testing on certain inputs or signals
- Used extensively in HDLs and sequential logic
- Implies priority

Priority Encoder Architecture

```
ARCHITECTURE priorenc OF enc3to8 IS
BEGIN
    Q(2) <= D(7) OR D(6) OR D(5) OR D(4);
    Q(1) <= D(7) OR D(6) OR ((not D(5)) and
        (not D(4)) and D(3))
        OR ((not D(5)) and (not D(4)) and D(2)) ;
    Q(0) <= -- In a similar fashion
END priorenc;
```

Another Encoder Form

- **WHEN-ELSE is similar to IF-THEN-ELSE**

```
BEGIN
    Q <= 7 WHEN D(7) = '1' ELSE
         6 WHEN D(6) = '1' ELSE
         5 WHEN D(5) = '1' ELSE
         4 WHEN D(4) = '1' ELSE
         3 WHEN D(3) = '1' ELSE
         2 WHEN D(2) = '1' ELSE
             |          |
         0;
```


4-to-1 Mux Architecture

```
ARCHITECTURE mux4to1 OF mux4 IS
BEGIN
  PROCESS(S)  -- Process is sensitive to S (S1,S0) Selects
    BEGIN
      CASE S IS
        WHEN "00" => Y <= D(0);
        WHEN "01" => Y <= D(1);
        WHEN "10" => Y <= D(2);
        WHEN "11" => Y <= D(3);
        WHEN OTHERS => Y <= '0';
      END CASE;
  END PROCESS;
END mux4to1;
```


4-Bit Magnitude Comparator Architecture

```
ARCHITECTURE behavioral OF mag4 IS
  SIGNAL compare : STD_LOGIC_VECTOR(2 downto 0);
BEGIN
PROCESS (ai,bi)  -- Sensitive to Ai and Bi Integer Arrays
  BEGIN
    IF    ai < bi THEN compare <= "110";
    ELSIF ai = bi THEN compare <= "101";
    ELSIF ai > bi THEN compare <= "011";
    ELSE compare <= "111";
    END IF;

    agtb <= compare(2);
    aeqb <= compare(1);
    altb <= compare(0);
```


Parity Generator (6-Bit) Architecture

```
ARCHITECTURE behavioral OF pargen IS
BEGIN
  paritv <= A0 XOR A1 XOR A2 XOR A3 XOR A4 XOR A5;
END behavioral;
```


Structured VHDL (Terms)

- **Hierarchy**
  - A group of design entities associated in a series of levels (the hierarchy) in which complete designs form portions or subsections of the upper design
- **Component**
  - A complete VHDL Design Entity that can be used as part of a higher level file in a hierarchical design

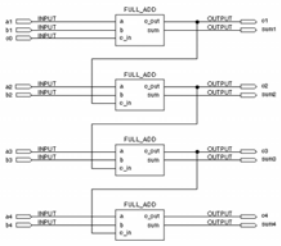
Structured VHDL (Terms)

- **Port**
  - An input or output of a VHDL design entity or component
- **Component Declaration**
  - A statement that defines the IO port names of a component in a VHDL design entity
- **Instantiate**
  - To use an instance of a component

Structured VHDL (Terms)

- Component Instantiation
  - A statement that maps the port names of a VHDL component to port names, internal signals, or variables of a higher level VHDL design entity
  - Also called a PORT MAP

Structured VHDL (example)



Full Adder VHDL Entity

```
ENTITY full_add IS
PORT( a, b, cin  :IN BIT;
      cout, sum  :OUT BIT);
END full_add;
```


Full Adder VHDL Architecture

```
ARCHITECTURE adder OF full_add IS
BEGIN
      cout <= ((a XOR b) AND CIN) OR (a AND b);
      sum  <= (a XOR b) XOR cin;
END adder;
```


4-Bit Parallel Adder Entity

```
ENTITY add4par IS
      PORT( c0    :IN BIT;
            a, b  :IN BIT_VECTOR(4 downto 1);
            c4    :OUT BIT;
            sum   :OUT BIT_VECTOR(4 downto 1));
END add4par;
```


4-Bit Parallel Adder Component

```
COMPONENT full_add -- Previous FA Design Entity
PORT( a, b, cin   :IN BIT;
      cout, sum   :OUT BIT);
END COMPONENT

      SIGNAL c    :BIT_VECTOR(3 downto 1)
-- Internal signal used for intermediate carries
```

4-Bit Parallel Adder

- This design uses the 1-bit FA as a Component to create the 4-bit parallel adder
- The basic adder component is mapped four times uses a component instantiation such as adder1, adder2, etc.
- The connections are set as a PORT MAP for each instance of the component

4-Bit Adder Structured Architecture

```
BEGIN
adder1: full_add -- This defines the first component
        PORT MAP(a => A(1), b=> B(1), cin => C(0),
                 cout => C(1), sum=> SUM(1));
adder2: full_add -- This defines the second component
        PORT MAP(a => A(2), b=> B(2), cin => C(1),
                 cout => C(2), sum=> SUM(2));

adder3: full_add -- This defines the third component
        PORT MAP(a => A(3), b=> B(3), cin => C(2),
                 cout => C(3), sum=> SUM(3));

adder4: full_add -- This defines the fourth component
        PORT MAP(a => A(4), b=> B(4), cin => C(3),
                 cout => C4, sum=> SUM(4));
END adder;
```


Other Structures

- The preceding example mapped directly to 4 FA components
- Another approach would be to use a VHDL repetitive loop construct called a GENERATE statement
- This is similar to a FOR loop with an index variable
- Both types are still ripple-carry adders

4-Bit Adder Architecture with GENERATE Statement

```
ARCHITECTURE adder of add4gen IS
    COMPONENT full_add
        PORT ( a, b, cin : IN BIT;
               cout, sum : OUT BIT );
    SIGNAL c : BIT_VECTOR (4 downto 0);
BEGIN
    C(0) <= C0;
    FOR i IN 1 to 4 GENERATE
        adders: full_add PORT MAP( a(i), b(i), C(i -1), C(i), sum(i) );
    END GENERATE;
END adder;
```