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

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

  \[
  \begin{array}{c|c|c|c}
    \text{Borrow} & \text{A} & \text{B} & \text{Diff} \\
    \hline
    0 & 0 & 0 & 0 \\
    1 & 0 & 1 & 0 \\
    1 & 1 & 1 & 1 \quad (2_{10} - 1_{10} = 1_{10})
  \end{array}
  \]

Binary Subtraction (example 1)

\[
\begin{array}{c}
  1110 \\
  - 1001 \\
  \hline
  ? \\
\end{array}
\quad \text{Borrow Stage}
\begin{array}{c}
  110 \quad (10) \\
  - 100 \quad 1 \\
  \hline
  010 \quad 1
\end{array}
\]

\[(14 - 9 = 5)\]

Binary Subtraction (example 2)

\[
\begin{array}{c}
  10000 \\
  - 101 \\
  \hline
  ? \\
\end{array}
\quad \text{Borrow ripples to LSB}
\begin{array}{c}
  0111 \quad (10) \\
  - 101 \quad - 10 \quad 1 \\
  \hline
  101 \quad 1
\end{array}
\]

\[(16 - 5 = 11)\]

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

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)

\[\text{Note: Positive numbers are the same in all three forms}\]
**True Magnitude Form**

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

5-bit number examples:
- $+25_{10} = 011001$ (Same as $+25$ with $S=1$)
- $-25_{10} = 111001$ (Same as $+25$ with $S=1$)

**1’s Complement Form**

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

8-bit number examples:
- $+57_{10} = 00111001$ (All bits inverted)
- $-57_{10} = 11000110$ (All bits inverted)
- $+72_{10} = 01001000$ (All bits inverted)
- $-72_{10} = 10110111$ (All bits inverted)

**2’s Complement Form**

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

Examples:

- $+57 = 00111001$
- $-57 = 11000110$ (1’s comp)
- $-57 + 1 = 11000111$ (2’s comp)
- $+72 = 01001000$
- $-72 = 10110111$ (1’s comp)
- $-72 + 1 = 10111000$ (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: $Sign = 0$

Examples:

- $+30: 00011110$
- $+75: 01001011$
- $+105: 01101001$

**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

- Add the 1’s complement and then add any carry:
  - $+80 = 01010000$
  - $+85 = 01010011$
  - $-85 = 10111110$ (1’s comp)
  - $80 + 01010000$
  - $-85 + 10111110$
  - $-85 + 10111110 + 1 = 00001110$ (End around carry)
  - $+15 = 00001111$
2’s Complement Subtraction

- Add the 2’s complement to the minuend
  \[ +80_{10} = 01010000 \]
  \[ +65_{10} = 01000001 \]
  \[ -65_{10} = 10111110 \text{ (1's complement)} \]
  \[ +1 \]
  \[ -65_{10} = 10111111 \text{ (2's complement)} \]

- If a carry results, discard it
  \[ 80 \quad 01010000 \]
  \[ -65 \quad +10111111 \text{ (Discard the carry)} \]

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

\[ +65: \quad 0100 \quad 0001 \]
\[ -80: \quad +1011 \quad 0000 \text{ (2's C.)} \]
\[ 1111 \quad 0001 \text{ (Negative sum)} \]
\[ 0000 \quad 1110 \text{ (Invert)} \]
\[ +1 \quad \text{ (Add 1)} \]
\[ 0000 \quad 1111 \text{ (Final Answer: 15, negative)} \]

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

\[ +16 = 00010000 \]
\[ -16 = 11101111 \text{ (1'S C.)} \]
\[ +1 \text{ (Add 1)} \]
\[ 11110000 \text{ (2's C.)} \]

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

\[ +75: 01001011 \]
\[ +96: 01001011 \]
\[ 10101011 \text{ (negative - False)} \]

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

\[ -60: 10110000 \text{ (2's C.)} \]
\[ -65: 10111111 \text{ (2's C.)} \]
\[ 01101111 \text{ (positive - False)} \]

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)

\[ 4987_{10} = 0100 \ 1001 \ 1000 \ 0111 \text{ (BCD)} \]
\[ 84_{10} = 1000 \ 0100 \text{ (BCD)} \]

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
  - \( g_3 = b_3 \)
  - \( g_2 = b_2 \oplus b_3 \)
  - \( g_1 = b_1 \oplus b_2 \)
  - \( g_0 = b_0 \oplus b_1 \)
- Very useful in hardware design

<table>
<thead>
<tr>
<th>Decimal</th>
<th>True Binary</th>
<th>Gray Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0110</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0100</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>
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
    
    \[
    \begin{array}{c|c}
    0 + 0 & 0 \\
    0 + 1 & 1 \\
    1 + 1 & 0 \\
    \end{array}
    \]

HA Circuit

- Basic Equations
  \[
  \begin{align*}
  S &= A \text{ xor } B \\
  C &= A \text{ and } B
  \end{align*}
  \]

    \[
    \begin{array}{ccc|c}
    A & B & C & S \\
    0 & 0 & 0 & 0 \\
    0 & 1 & 0 & 1 \\
    1 & 0 & 0 & 1 \\
    1 & 1 & 1 & 0 \\
    \end{array}
    \]

HA Circuit and Symbol

Full Adder
**Full Adder**

- Adds a Cin to the HA

**Basic Equations:**

\[
\begin{align*}
C_{out} &= (A \oplus B) \text{Cin} + AB \\
S &= (A \oplus B) \oplus \text{Cin}
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Cin</th>
<th>Cout</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FA Circuit**

\[
\begin{align*}
C_{out} &= (A \oplus B)C_{in} + AB \\
S &= (A \oplus B) \oplus C_{in}
\end{align*}
\]

**FA Circuit (Using HAs)**

**Parallel (N-bit) Binary Adder**

- 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

*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 $4 \times 2 = 8$ 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 = $A - B = A + \bar{B} + 1$
- This operation can be done in a parallel N-Bit FA by Inverting $B_i$ 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_{s4} 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 $\Sigma_1$ and $\Sigma_3$ or $\Sigma_2$

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

```
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;

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 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
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;

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 <= '1' 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;
END;
```

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
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);
END behavioral;
```

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

- 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

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

Full Adder VHDL Architecture

```vhdl
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

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

4-Bit Parallel Adder Component

```vhdl
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.

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.