14   Adder and Subtractor Circuits

14.1 Negative Numbers: Two’s Complement Representation

- Different digital numbering systems exist:

<table>
<thead>
<tr>
<th>Binary $m -1 = 3$</th>
<th>Positive Decimal</th>
<th>Sign and Magnitude</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>0</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>0 001</td>
<td>1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>0 010</td>
<td>2</td>
<td>+2</td>
<td>+2</td>
</tr>
<tr>
<td>0 011</td>
<td>3</td>
<td>+3</td>
<td>+3</td>
</tr>
<tr>
<td>0 100</td>
<td>4</td>
<td>+4</td>
<td>+4</td>
</tr>
<tr>
<td>0 101</td>
<td>5</td>
<td>+5</td>
<td>+5</td>
</tr>
<tr>
<td>0 110</td>
<td>6</td>
<td>+6</td>
<td>+6</td>
</tr>
<tr>
<td>0 111</td>
<td>7</td>
<td>+7</td>
<td>+7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary $m -1 = 3$</th>
<th>Positive Decimal</th>
<th>Sign and Magnitude</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>8</td>
<td>-0</td>
<td>-8</td>
</tr>
<tr>
<td>1 001</td>
<td>9</td>
<td>-1</td>
<td>-7</td>
</tr>
<tr>
<td>1 010</td>
<td>10</td>
<td>-2</td>
<td>-6</td>
</tr>
<tr>
<td>1 011</td>
<td>11</td>
<td>-3</td>
<td>-5</td>
</tr>
<tr>
<td>1 100</td>
<td>12</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td>1 101</td>
<td>13</td>
<td>-5</td>
<td>-3</td>
</tr>
<tr>
<td>1 110</td>
<td>14</td>
<td>-6</td>
<td>-2</td>
</tr>
<tr>
<td>1 111</td>
<td>15</td>
<td>-7</td>
<td>-1</td>
</tr>
</tbody>
</table>

- For the interpretation of digital numbers it is important to know the underlying numbering system!
- If you have to enlarge the number of bits in the two's complement system without changing the numbers value you need appropriate **sign extension**.
Two’s complement (I)

- An explicit sign for negative numbers cannot be produced in binary number representation. Hence one half of all binary numbers which can be described with m digits is reserved for the representation of negative numbers.

- **Every number has a unique representation (including zero!).** The two’s complement is the usual representation of negative numbers within arithmetical operations of microprocessors.

- Subtraction is carried out by the addition of two’s complement numbers.

- Two’s complement $C(z)$ for negative numbers $z$ with radix $R = 2$:
  - $z$ : negative number
  - $z^*$ : magnitude $|z|$ of a negative number $z$
  - $z_p$ : positive number
  - **Two’s complement rule** for numbers with $m$ digits and radix $R = 2$:

  \[
  \begin{array}{c}
  C(z) \quad \text{complement} \\
  + z^* \quad \text{magnitude} \\
  R^m \quad \text{auxiliary number}
  \end{array}
  \]

  \[R^m = \text{highest number} + 1, \text{ out of representable range}\]
Two’s complement (II)

Non overlapping ranges of numbers characterise a true complement:

Negative numbers: \(-R^{m-1} \leq z < 0\)

Positive numbers: \(0 \leq z_p \leq R^{m-1} - 1\)

Two’s complement: \(R^{m-1} \leq C(z) < R^m\)

\[ C(z) = R^m - z \]
Simplified two’s complement representation

- Modified writing of the two’s complement rule: \[ C(z) = \{ (R^m - 1) - z^* \} + 1 \]
  
  \( (R^m - 1) : \) \( R = 2 \); binary number with \( m \) digits; all digits are \( \text{‘1’} \)
  
  \( \{ (R^m - 1) - z^* \} : \) Because \( 1 - 0 = 1 \) and \( 1 - 1 = 0 \) it follows that \( z^* \) is inverted.

- The two’s complement rule:
  By inverting the magnitude \( z^* \) of the negative number \( z \) and adding a \( \text{‘1’} \) to it, we get a two’s complement representation.

- Retransformation of a two’s complement number \( C(z) \):
  \[ z^* = \{ (R^m - 1) - C(z) \} + 1 \]

- This expression means that we can get the magnitude \( z^* \) of a negative number \( z \) which is represented by its two’s complement \( C(z) \) in the same way which is written above: Inversion of \( C(z) \) and adding a \( \text{‘1’} \).
Two’s complement addition and subtraction

- The rules are the same as for unsigned binary number representation.

- But we have to consider two additional factors:
  - Carry and borrow bits which exceed the number \( m \) of digits we have selected for representation of signed numbers have to be ignored (truncated) in order to get a correct result.
  - Results can leave (exceed) the agreed range (interval) of two’s complement representation. In such cases we call the result incorrect because of an overflow (OV bit). The solution to the problem is to enlarge the number \( m \) of digits and repeat the operation.

- An overflow occurs under special circumstances:
  - Addition \((A + B)\):
    - Both addends are positive and the result is negative.
    - Both addends are negative and the result is positive.
  - Subtraction \((A - B)\):
    - The minuend \( A \) is negative, the subtrahend \( B \) is positive and the result is positive.
    - The minuend \( A \) is positive, the subtrahend \( B \) is negative and the result is negative.
14.2 Full Adder Circuit

Logic function:
Sum: \( \text{SUM} = A \oplus B \oplus \text{CI} \)
Carry: \( \text{COUT} = (A \land B) \lor (\text{CI} \land (A \oplus B)) \)
Generate: \( \text{CG} = A \land B \)
Propagate: \( \text{CP} = A \oplus B \)

<table>
<thead>
<tr>
<th>CI</th>
<th>B</th>
<th>A</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>0</td>
<td>1</td>
<td>0</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
14.3 Ripple Carry Adder

- With n 1-bit full adders an addition of two n-bit words can be implemented. All 1-bit adders have to be cascaded.
- The ripple carry effect belongs to conventional arithmetic where a carry bit from an addition operation is always added to the next most significant stage.
- If fractional parts of numbers have to be added then the number of fractional digits of both addends have to be equal. The position of the radix point is arbitrary because the adder does not sense it. It is up to the user to design an appropriate display of results.
- The subscripts of signals within a formula and block diagram correspond to the weight of a digit in a binary number. The subscript of carry signals represents the position where it is generated.
Ripple carry adder simulation
4-bit Ripple carry adder timing analysis

**Example:** Adder operation with hexadecimal numbers $x01 + 0x0E + CIN = 0x10$

- Propagation delays: $t_{pLH} = 9\text{ns}$ and $t_{pHL} = 20\text{ns}$.
- The sum bits $S0 \ldots S3$ will go HIGH temporarily and finally change to LOW when the carry signal $C$ is propagating from adder stage to stage. Meanwhile the adder result $S[3:0]$ is erroneous.
- A stationary valid sum result will be developed after several steps:
  - The three Carry bits $C0, C1, C2$ will set in sequence one after the other: $3 \times 9\text{ns} = 27\text{ns}$
  - The last Carry bit $C2$ will force the sum MSB $S3$ to go LOW: $20\text{ns}$
  
  Finally the result is valid after $47\text{ns}$.

- The total propagation delay before a result is valid depends on the preceding and new adder inputs.
  What will be the worst-case situation and how long will this ripple carry process last?

**Conclusion:**
- There are practical limitations associated with the operation of an R-C adder because the final result will not be valid until the carry bits have been propagated to the most significant position. During the propagation time all inputs to the adder stages must remain stable. Thus the maximum frequency of input data representation for a series of additions will be determined by the ripple time.
14.4 Carry-Look-Ahead Adder

- A solution to the ripple-carry propagation delay problem providing a reduced carry signal delay (but with more implemented hardware) can be made by using a modified carry chain representation which uses two internal signals of an adder stage. This logic function will estimate all carry bits in parallel with fewer logic gates to propagate through.
- Two internal full adder signals will be fed into the so called Carry-Look-Ahead logic:

**Generate signal:** \( CG_i = 1 \) shows that an adder at position \( i \) generates a carry \( C_i \) because of the input pattern \( A_i, B_i \).

**Propagate signal:** \( P_i = 1 \) shows that a carry \( C_{i-1} \) from the preceding stage \( i-1 \) will be propagated to the following stage \( i+1 \).

<table>
<thead>
<tr>
<th>No.</th>
<th>( C_{i-1} )</th>
<th>( B_i )</th>
<th>( A_i )</th>
<th>( CG_i )</th>
<th>( CP_i )</th>
<th>( C_i )</th>
<th>( S_i )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( C_i ) generated</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( C_{i-1} ) absorbed</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( C_{i-1} ) propagated</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( C_{i-1} ) propagated</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( C_i ) generated</td>
</tr>
</tbody>
</table>

**Logic functions:**

- \( CG_i = A_i \land B_i \)
- \( CP_i = \neg(A_i \leftrightarrow B_i) \) (xor)
4-bit Carry-Look-Ahead Adder (CLA) circuit (I)
A carry look-ahead adder (CLA) makes use of a different approach for carry calculation. The expression for the carry bit of a preceding stage will be substituted into the expression of the following stage:

- **General expression:** \( C_i = C_{i-1} \land CP_i \lor CG_i \)
- **1st stage \( i = 0 \):** \( C_0 = C_{-1} \land CP_0 \lor CG_0 \)
- **Successive substitution of the preceding stage will yield:**
  - **2nd stage \( i = 1 \):**
    \[
    C_1 = C_0 \land CP_1 \lor CG_1 = (C_{-1} \land CP_0 \lor CG_0) \land CP_1 \lor CG_1
    \]
    \[
    C_1 = (C_{-1} \land CP_1 \land CP_0) \lor (CP_1 \land CG_0) \lor CG_1
    \]

- Now a full adder (FA) module has two additional outputs: propagate \( CP_i \) and generate \( CG_i \).
- Whereas the \( S_i \) terms are produced within each FA stage, the \( C_i \) terms must be formed externally by a carry propagate/generate (CPG) network.
- The main advantage of this CPG solution is made up by two properties:
  - **Each carry bit depends** on the carry input of the first stage.
  - **All carry bits** are calculated in parallel and each with **three level logic**.
  - **The hardware requirement** for the GPG network increases by a quadratic function as the number \( n \) of stages increases:

\[
\text{Total gate count for CLA : } (n^2 + 3n)/2 + 3n = (n^2 + 9n)/2
\]
Simulation waveforms of a 4-bit Carry-Look-Ahead adder
VHDL-description of a 4-Bit Carry-Look-Ahead adder

- The VHDL source code consists of three entities:
  1. Full adder ADD_COMP
  2. Carry-Look-Ahead generator CLA_GEN
  3. Structural VHDL model where all components will be instantiated CLA_ADD

- Symbolic propagation delays are applied:
  10ns with full adder signals, 15ns with CLA-generator signals

```vhdl
-- 1-bit full adder component with generate and propagate outputs
entity ADD_COMP is
  port ( A, B, CIN: in bit;
         SUM, CO, CG, CP: out bit);
end ADD_COMP;
architecture CLA_ARCH of ADD_COMP is
begin
  SUM  <= A xor B xor CIN after 10 ns;
  CO   <= (A and B) or (CIN and (A xor B)) after 10 ns;
  CP   <= A xor B after 10 ns;
  CG   <= A and B after 10 ns;
end CLA_ARCH;
```
-- 4-bit Carry-Look-Ahead generator
entity CLA_GEN is
  port ( G, P: in bit_vector(3 downto 0);
         CIN: in bit;
         C: out bit_vector(2 downto 0);
         CGOUT, CPOUT: out bit);
end CLA_GEN;
architecture CLA of CLA_GEN is
begin
  C(0) <= G(0) or (P(0) and CIN) after 15 ns;
  C(1) <= G(1) or (P(1) and G(0)) or
         (P(1) and P(0) and CIN) after 15 ns;
  C(2) <= G(2) or (P(2) and G(1)) or
         (P(2) and P(1) and G(0)) or
         (P(2) and P(1) and P(0) and CIN) after 15 ns;
  CPOUT<= (P(3) and P(2) and P(1) and P(0)) after 15 ns;
  CGOUT<= G(3) or (P(3) and G(2)) or
         (P(3) and P(2) and G(1)) or
         (P(3) and P(2) and P(1) and G(0)) or
         (P(3) and P(2) and P(1) and P(0) and CIN) after 15 ns;
end CLA;
entity CLA_ADD is -- 4-bit CLA structural model: top entity
    port ( A, B: in bit_vector(3 downto 0);
          CIN: in bit;
          SUM: out bit_vector(3 downto 0);
          CGOUT, CPOUT: out bit);
end CLA_ADD;
architecture STRUKTUR of CLA_ADD is
component ADD_COMP -- component declaration full adder
    port ( A, B, CIN: in bit;
           SUM, CO, CG, CP: out bit);
end component;
component CLA_GEN -- component declaration CLA-generator
    port ( G, P: in bit_vector(3 downto 0);
           CIN: in bit;
           C: out bit_vector(2 downto 0);
           CGOUT, CPOUT: out bit);
end component;
signal CG, CP, CARRY: bit_vector(3 downto 0); -- local signals
begin
    CARRY(0) <= CIN;
    VA: for I in 0 to 3 generate -- 4 full adder cascading
        ADD:ADD_COMP port map (A(I), B(I), CARRY(I), SUM(I), open, CG(I), CP(I));
    end generate VA;
    CLA:CLA_GEN port map(CG,CP,CIN,CARRY(3 downto 1),CGOUT,CPOUT);-- CLA-generator
end STRUKTUR;
Structural VHDL

- Each **pre-compiled entity** can be used as a **component** within a higher level of hierarchy.

- For a VHDL **component** to be used with a structural VHDL hierarchy, it first has to be **declared**. This is done in the declaration part between **architecture** and **begin**. The port list in the component declaration must be identical to that in the component’s entity.

- The component is **instanced and linked** to other components and to the interface signals of the top entity using the **port map** command. Each **port map** has to be introduced by a label and the component name. Within the **port map** it is the order of signals which decides how the interconnection takes place. Types and range of component port (actuals) and local signals (locals) have to be identical. Not-used output signals can be left unconnected with the **open** keyword.

- If the **same component has to be instanced several times** in the same architecture, it may be effective to include port map commands in loop. This is done using the **for – generate** command. The loop subscript must not be declared explicitly and can be used to evaluate signal vector subscripts. Each repeated linking with **for – generate** has to be marked with a label which will usually be used by schematic generators to identify component blocks.
14.5 Combined Adder and Subtractor

1-bit full adder/subtractor (AS_COMP):
In order to simplify microprocessor hardware the subtractor carry signal will be designed as a low active signal: Carry = 1 means with subtraction no borrow but with the sum operation, a carry to the next adder stage.
A selection input bit SEL will control the logic function of the AS_COMP component:

<table>
<thead>
<tr>
<th>SEL</th>
<th>C_n</th>
<th>A_n</th>
<th>B_n</th>
<th>S_n</th>
<th>C_{n+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4-bit ripple-carry adder / subtractor

A[3:0] >
B[3:0] >
CIN >
SEL >

CIN SEL CIN SEL CIN SEL CIN SEL

CIN SEL CIN SEL CIN SEL CIN SEL

CIN SEL CIN SEL CIN SEL CIN SEL

CIN SEL CIN SEL CIN SEL CIN SEL

AS_COMP

S[0] S[0] S[0] S[0]


S[3:0] >

COUT >
Simulation waveforms of a 4-bit adder/subtractor
14.6 Arithmetic Operations

Several arithmetical operators are supported by synthesis tools. It is recommended that the data type \texttt{std_logic_vector} with arithmetic operations is used. Then the operators have to be defined in a package. This has been done in package \texttt{ieee.std_logic_unsigned} (unsigned numbers) and \texttt{ieee.std_logic_signed} (signed numbers with two’s complement representation):

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;  -- only unsigned number operations
use ieee.std_logic_signed.all;   -- only signed operations
use ieee.std_logic_arith.all;    -- both kind of numbers; data types
                                 -- signed and unsigned have to be used!!
entity XYZ is
...
```

These libraries also support the relational operators \(=, /=, <, \leq, \geq\) for \texttt{std_logic_vector} data.
### Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
<th>Example</th>
<th>Synthesizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>$Y \leq A + B$</td>
<td>Yes</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>$Y \leq A - B$</td>
<td>yes</td>
</tr>
<tr>
<td>abs</td>
<td>Magnitude calculation (absolute value)</td>
<td>$Y \leq \text{abs}(A)$</td>
<td>yes</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>$Y \leq A \times B$</td>
<td>supported by several synthesis tools</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>$Y \leq A / B$</td>
<td>mostly not supported</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>$Y \leq 2^{\text{A}}$</td>
<td>only powers of two allowed that can be done by shift left or right</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus of division $A/B$</td>
<td>$Y \leq A \mod B$</td>
<td>Synthesizable if B is a power of 2</td>
</tr>
<tr>
<td></td>
<td>$A \mod B = A - B \times n$; (n integer part of division)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sign of result is equal to the sign of B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rem</td>
<td>Remainder of division $A/B$.</td>
<td>$Y \leq A \text{rem} B$</td>
<td>Synthesizable if B is a power of 2</td>
</tr>
<tr>
<td></td>
<td>$A \text{rem} B = A - (A/B) \times B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sign of result is equal to the sign of A.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>