Chapter 3
Numeric Basics

Representing and processing numeric data is a common requirement
- unsigned integers
- signed integers
- fixed-point real numbers
- floating-point real numbers
- complex numbers
Unsigned Integers

- Non-negative numbers (including 0)
  - Represent real-world data
    - e.g., temperature, position, time, ...
  - Also used in controlling operation of a digital system
    - e.g., counting iterations, table indices
- Coded using unsigned binary (base 2) representation
  - analogous to decimal representation

Binary Representation

- Decimal: base 10
  - $124_{10} = 1 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$
- Binary: base 2
  - $124_{10} = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    - $= 1111100_2$
  - In general, a number $x$ is represented using $n$ bits as $x_{n-1}, x_{n-2}, ..., x_0$, where
    $$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ... + x_02^0$$
Binary Representation

- Unsigned binary is a code for numbers
  - \( n \) bits: represent numbers from 0 to \( 2^n - 1 \)
    - 0: 0000...00; \( 2^n - 1 \): 1111...11
  - To represent \( x \): \( 0 \leq x \leq N - 1 \), need \( \lceil \log_2 N \rceil \) bits

- Computers use
  - 8-bit bytes: 0, ..., 255
  - 32-bit words: 0, ..., \(~4\) billion
- Digital circuits can use whatever size is appropriate

 Unsigned Integers in VHDL

- Package numeric_std provides unsigned vector type and arithmetic operations

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;
entity multiplexer_6bit_4_to_1 is
  port ( a0, a1, a2, a3 : in unsigned (5 downto 0);
        sel : in std_logic_vector (1 downto 0);
        z : out unsigned (5 downto 0) );
end entity multiplexer_6bit_4_to_1;
architecture eqn of multiplexer_6bit_4_to_1 is
begin
  with sel select z <= a0 when "00",
                  a1 when "01",
                  a2 when "10",
                  a3 when others;
end architecture eqn;
```
Octal and Hexadecimal

- Short-hand notations for vectors of bits
- Octal (base 8)
  - Each group of 3 bits represented by a digit
  - 0: 000, 1: 001, 2: 010, ..., 7: 111
  - $253_8 = 010\ 101\ 011_2$
  - $11001011_2 \Rightarrow 11\ 001\ 011_2 = 313_8$
- Hex (base 16)
  - Each group of 4 bits represented by a digit
  - 0: 0000, ..., 9: 1001, A: 1010, ..., F: 1111
  - $3CE_{16} = 0011\ 1100\ 1110_2$
  - $11001011_2 \Rightarrow 1100\ 1011_2 = CB_{16}$

Extending Unsigned Numbers

- To extend an $n$-bit number to $m$ bits
  - Add leading 0 bits
  - e.g., $72_{10} = 1001000 = 000001001000$

\[
\begin{array}{|c|c|}
\hline
x(0) & y(0) \\
\hline
x(1) & y(1) \\
\vdots & \vdots \\
x(n-1) & y(n-1) \\
\hline
\end{array}
\]

```
signal x : unsigned(3 downto 0);
signal y : unsigned(7 downto 0);
y <= "0000" & x;
y <= resize(x, 8);
```
**Truncating Unsigned Numbers**

- To truncate from \( m \) bits to \( n \) bits
  - Discard leftmost bits
  - Value is preserved if discarded bits are 0
  - Result is \( x \mod 2^n \)

\[
\begin{array}{c|c}
\text{y(0)} & x(0) \\
\text{y(1)} & x(1) \\
\vdots & \vdots \\
\text{y(n-1)} & x(n-1) \\
\text{y(n)} & \\
\vdots & \vdots \\
\text{y(m-2)} & \\
\text{y(m-1)} & \\
\end{array}
\]

\[
x \leftarrow y(3 \text{ downto } 0);
\]

\[
x \leftarrow \text{resize}(y, 4);
\]

---

**Unsigned Addition**

- Performed in the same way as decimal

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

Overflow

Carry bits
Addition Circuits

- **Half adder**
  - for least-significant bits
    \[ s_0 = x_0 \oplus y_0 \]
    \[ c_1 = x_0 \cdot y_0 \]

- **Full adder**
  - for remaining bits
    \[ s_i = (x_i \oplus y_i) \oplus c_i \]
    \[ c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</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>

Ripple-Carry Adder

- **Full adder for each bit,** \( c_0 = 0 \)

- **Worst-case delay**
  - from \( x_0, y_0 \) to \( s_n \)
  - carry must ripple through intervening stages, affecting sum bits
Improving Adder Performance

- **Carry kill:**  \( k_i = x_i \cdot \overline{y_i} \)

- **Carry propagate:**  \( p_i = x_i \oplus y_i \)

- **Carry generate:**  \( g_i = x_i \cdot y_i \)

**Adder equations**

\[
\begin{align*}
s_i &= p_i \oplus c_i  \\
c_{i+1} &= g_i + p_i \cdot c_i
\end{align*}
\]

Fast-Carry-Chain Adder

- **Also called Manchester adder**

Xilinx FPGAs include this structure
**Carry Lookahead**

\[ c_{i+1} = g_i + p_i \cdot c_i \]

\[ c_1 = g_0 + p_0 \cdot c_0 \]

\[ c_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \]

\[ c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]

\[ c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 
+ p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \]

**Carry-Lookahead Adder**

- Avoids chained carry circuit

- Use multilevel lookahead for wider numbers
Other Optimized Adders

- Other adders are based on other reformulations of adder equations
- Choice of adder depends on constraints
  - e.g., ripple-carry has low area, so is ok for low performance circuits
  - e.g., Manchester adder ok in FPGAs that include carry-chain circuits

Adders in VHDL

- Use operations from numeric_std

```vhdl
library ieee; use ieee.numeric_std.all;
...
signal x, y, s: unsigned(7 downto 0);
...
s <= a + b;

signal tmp_result : unsigned(8 downto 0);
signal c : std_logic;
...
tmp_result <= ('0' & a) + ('0' & b);
c <= tmp_result(8);
s <= tmp_result(7 downto 0);
```
**Unsigned Subtraction**

- As in decimal

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{array}
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 \\
\end{array}
\begin{array}{c}
0 \\
0 \\
1 \\
\end{array}
\]

**Subtraction Circuits**

- For least-significant bits
  \[d_0 = x_0 \oplus y_0\]
  \[b_1 = x_0 \cdot y_0\]

<table>
<thead>
<tr>
<th>(x_0)</th>
<th>(y_0)</th>
<th>(b_1)</th>
<th>(s_1)</th>
<th>(b_{i+1})</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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</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>
</tbody>
</table>

- For remaining bits
  \[d_i = (x_i \oplus y_i) \oplus b_i\]
  \[b_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot b_i\]
Adder/Subtractor Circuits

- Many systems add and subtract
  - Trick: use complemented borrows

  Addition
  \[ s_i = (x_i \oplus y_i) \oplus c_i \]

  Subtraction
  \[ d_i = (x_i \oplus \overline{y_i}) \oplus \overline{b_i} \]

  \[ c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i \] \[ \overline{b_{i+1}} = x_i \cdot \overline{y_i} + (x_i \oplus y_i) \cdot \overline{b_i} \]

- Same hardware can perform both
  - For subtraction: complement \( y \), set \( \overline{b_0} = 1 \)

Adder/Subtractor Circuits

- Adder can be any of those we have seen
  - depends on constraints
Subtraction in VHDL

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;
entity adder_subtractor is
  port ( x, y : in unsigned(11 downto 0);
         s : out unsigned(11 downto 0);
         mode : in std_logic;
         error : out std_logic);
end entity adder_subtractor;

architecture behavior of adder_subtractor is
  signal s_tmp : unsigned(12 downto 0);
  begin
    s_tmp <= ('0' & x) + ('0' & y) when mode = '0' else
             ('0' & x) - ('0' & y);
    s <= s_tmp(11 downto 0);
    error <= s_tmp(12);
  end architecture behavior;
```

Increment and Decrement

- Adding 1: set \( y = 0 \) and \( c_0 = 1 \)

\[
S_i = x_i \oplus c_i \quad c_{i+1} = x_i \cdot c_i
\]

- These are equations for a half adder

- Similarly for decrementing: subtracting 1
Increment/Decrement in VHDL

- Just add or subtract 1

```vhdl
signal x, s: unsigned(15 downto 0);
...
s <= x + 1;  -- increment x
s <= x - 1;  -- decrement x
```

- Note: 1 (integer), not '1' (bit)

Equality Comparison

- XNOR gate: equality of two bits
  - Apply bitwise to two unsigned numbers

```vhdl
eq <= '1' when x = y else '0';
```

- In VHDL, `x = y` gives a boolean result
  - false or true
  - can't assign to a std_logic signal
Inequality Comparison

- Magnitude comparator for \( x > y \)

\[
x_{n-1} > y_{n-1} \\
x_{n-1} = y_{n-1} \\
x_{n-2} > y_{n-2} \\
x_{n-2} = y_{n-2} \\
\vdots \\
x_1 > y_1 \\
x_1 = y_1 \\
x_0 > y_0 \\
\]

\[
\text{if } x_{n-1} > y_{n-1} \text{ then } \text{gt} \\
\text{if } x_{n-1} = y_{n-1} \text{ then } \text{eq} \\
\text{if } x_{n-2} > y_{n-2} \text{ then } \text{gt} \\
\text{if } x_{n-2} = y_{n-2} \text{ then } \text{eq} \\
\vdots \\
\text{if } x_1 > y_1 \text{ then } \text{gt} \\
\text{if } x_1 = y_1 \text{ then } \text{eq} \\
\text{if } x_0 > y_0 \text{ then } \text{gt} \\
\]

Comparison Example in VHDL

- Thermostat with target temperature
  - Heater or cooler on when actual temperature is more than 5° from target

```vhdl
entity thermostat is
  port ( target, actual : in unsigned(7 downto 0);
         heater_on, cooler_on : out std_logic);
end entity thermostat;

architecture rtl of thermostat is
begin
  heater_on <= '1' when actual < target - 5 else '0';
  cooler_on <= '1' when actual > target + 5 else '0';
end architecture rtl;
```
Scaling by Power of 2

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

\[ 2^k x = x_{n-1}2^{k+n-1} + x_{n-2}2^{k+n-2} + \cdots + x_02^k + (0)2^{k-1} + \cdots + (0)2^0 \]

- This is \( x \) shifted left \( k \) places, with \( k \) bits of 0 added on the right
  - *logical shift left* by \( k \) places
  - e.g., \( 00010110_2 \times 2^3 = 00010110000_2 \)
- Truncate if result must fit in \( n \) bits
  - overflow if any truncated bit is not 0

Scaling by Power of 2

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

\[ x / 2^k = x_{n-1}2^{n-1-k} + x_{n-2}2^{n-2-k} + \cdots + x_k2^0 + x_{k-1}2^{-1} + \cdots + x_{-k}2^{-k} \]

- This is \( x \) shifted right \( k \) places, with \( k \) bits truncated on the right
  - *logical shift right* by \( k \) places
  - e.g., \( 01110110_2 / 2^3 = 01110_2 \)
- Fill on the left with \( k \) bits of 0 if result must fit in \( n \) bits
Scaling in VHDL

- `shift_left` and `shift_right` operations
- Result is the same size as the operand

\[
s = 00010011_2 = 19_{10}
\]

\[
y \leftarrow \text{shift_left}(s, 2); \quad y = 01001100_2 = 76_{10}
\]

\[
s = 00010011_2 = 19_{10}
\]

\[
y \leftarrow \text{shift_right}(s, 2); \quad y = 000100_2 = 4_{10}
\]

Unsigned Multiplication

\[
xy = x(y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \cdots + y_02^0)
\]

\[
= y_{n-1}x2^{n-1} + y_{n-2}x2^{n-2} + \cdots + y_0x2^0
\]

- \(y_i x 2^i\) is called a partial product
  - if \(y_i = 0\), then \(y_i x 2^i = 0\)
  - if \(y_i = 1\), then \(y_i x 2^i\) is \(x\) shifted left by \(i\)
- Combinational array multiplier
  - AND gates form partial products
  - Adders form full product
Unsigned Multiplication

- Adders can be any of those we have seen
- Optimized multipliers combine parts of adjacent adders

Product Size

- Greatest result for \( n \)-bit operands:
  \[
  (2^n - 1)(2^n - 1) = 2^{2n} - 2^n - 2^n + 1 = 2^{2n} - (2^{n+1} - 1)
  \]
- Requires \( 2^{2n} \) bits to avoid overflow
- Adding \( n \)-bit and \( m \)-bit operands
  - requires \( n + m \) bits

```vhdl
signal x : unsigned(7 downto 0);
signal y : unsigned(13 downto 0);
signal p : unsigned(21 downto 0);
...
p <= x * y;
```
Other Unsigned Operations

- Division, remainder
  - More complicated than multiplication
  - Large circuit area, power
- Complicated operations are often performed sequentially
  - in a sequence of steps, one per clock cycle
  - cost/performance/power trade-off

Gray Codes

- Important for position encoders
  - Only one bit changes at a time

<table>
<thead>
<tr>
<th>Segment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
</tr>
<tr>
<td>5</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>0101</td>
</tr>
<tr>
<td>7</td>
<td>0100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1100</td>
</tr>
<tr>
<td>9</td>
<td>1101</td>
</tr>
<tr>
<td>10</td>
<td>1111</td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
</tr>
<tr>
<td>12</td>
<td>1010</td>
</tr>
<tr>
<td>13</td>
<td>1011</td>
</tr>
<tr>
<td>14</td>
<td>1001</td>
</tr>
<tr>
<td>15</td>
<td>1000</td>
</tr>
</tbody>
</table>

- See book for $n$-bit Gray code
Signed Integers

- Positive and negative numbers (and 0)
- \( n \)-bit signed magnitude code
  - 1 bit for sign: 0 ⇒ +, 1 ⇒ –
  - \( n – 1 \) bits for magnitude
- Signed-magnitude rarely used for integers now
  - circuits are too complex
- Use 2s-complement binary code

2s-Complement Representation

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \]

- Most-negative number
  - 1000...0 = \(-2^{n-1}\)
- Most-positive number
  - 0111...1 = \(+2^{n-1} \) – 1
- \( x_{n-1} = 1 \) ⇒ negative,
  \( x_{n-1} = 0 \) ⇒ non-negative
- Since \( 2^{n-2} + \cdots + 2^0 = 2^{n-1} – 1 \)
2's-Complement Examples

- 00110101
  - \( = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 53 \)
- 10110101
  - \( = -1 \times 2^7 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 \)
  - \( = -128 + 53 = -75 \)
- 00000000 = 0
- 11111111 = -1
- 10000000 = -128
- 01111111 = +127

Signed Integers in VHDL

- Type signed from numeric_std
  ```vhdl
  library ieee; use ieee.numeric_std.all;
  ...
  s : signed(15 downto 0);
  ```

- Types signed and unsigned are distinct
  ```vhdl
  signal s1 : unsigned(11 downto 0);
  signal s2 : signed(11 downto 0);
  ...
  s1 <= s2; -- illegal
  s1 <= unsigned(s2); -- s2 is known to be non-negative
  ...
  s2 <= signed(s1); -- s1 is known to be less than 2**11
  ```
Octal and Hex Signed Integers

- Don’t think of signed octal or hex
  - Just treat octal or hex as shorthand for a vector of bits
- E.g., \(844_{10}\) is \(001101001100\)
  - In hex: \(0011\ 0100\ 1100\ \Rightarrow\ 34C\)
- E.g., \(-42_{10}\) is \(1111010110\)
  - In octal: \(1\ 111\ 010\ 110\ \Rightarrow\ 1726\ (10\ bits)\)

Resizing Signed Integers

- To extend a non-negative number
  - Add leading 0 bits
  - e.g., \(53_{10}\) = \(00110101\) = \(000000110101\)
- To truncate a non-negative number
  - Discard leftmost bits, provided
    - discarded bits are all 0
    - sign bit of result is 0
  - E.g., \(41_{10}\) is \(00101001\)
    - Truncating to 6 bits: \(101001\ —\ error!\)
Resizing Signed Integers

- To extend a negative number
  - Add leading 1 bits
  - See textbook for proof
  - e.g., $-75_{10} = 10110101 = 111110110101$
- To truncate a negative number
  - Discard leftmost bits, provided
    - discarded bits are all 1
    - sign bit of result is 1

In general, for 2s-complement integers

- Extend by replicating sign bit
  - sign extension
- Truncate by discarding leading bits
  - Discarded bits must all be the same, and the same as
    the sign bit of the result

```vhdl
signal x : signed (7 downto 0);
signal y : signed (15 downto 0);
...
y <= resize(x, y'length);
...
x <= resize(y, x'length);
```
Signed Negation

- Complement and add 1
  - Note that $\bar{x}_i = 1 - x_i$

$$
\bar{x} + 1 = -(1 - x_{n-1})2^{n-1} + (1 - x_{n-2})2^{n-2} + \cdots + (1 - x_0)2^0 + 1 \\
= -2^{n-1} + x_{n-1}2^{n-1} + 2^{n-2} - x_{n-2}2^{n-2} + \cdots + 2^0 - x_02^0 + 1 \\
= -(x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0) \\
- 2^{n-1} + (2^{n-2} + \cdots + 2^0) + 1 \\
= -x - 2^{n-1} + 2^{n-1} = -x
$$

- E.g., 43 is 00101011
  so $-43$ is $11010100 + 1 = 11010101$

Signed Negation

- What about negating $-2^{n-1}$?
  - 1000...00 $\Rightarrow$ 0111...11 + 1 = 1000...00
  - Result is $-2^{n-1}$!
- Recall range of $n$-bit numbers is not symmetric
  - Either check for overflow, extend by one bit, or ensure this case can’t arise
- In VHDL: use $-$ operator
  - E.g., $y <= -x$;
Signed Addition

\[ x = -x_{n-1}2^{n-1} + x_{n-2} \ldots 0 \quad \quad y = -y_{n-1}2^{n-1} + y_{n-2} \ldots 0 \]

\[ x + y = -(x_{n-1} + y_{n-1})2^{n-1} + x_{n-2} \ldots 0 + y_{n-2} \ldots 0 \]

\[ \text{yields } c_{n-1} \]

- Perform addition as for unsigned
- Overflow if \( c_{n-1} \) differs from \( c_n \)
- See textbook for case analysis
- Can use the same circuit for signed and unsigned addition

Signed Addition Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Signed Numbers</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>72:</td>
<td>0 1 0 0 1 0 0 0</td>
<td>no overflow</td>
</tr>
<tr>
<td>49:</td>
<td>0 0 1 1 0 0 0 1</td>
<td>no overflow</td>
</tr>
<tr>
<td>121:</td>
<td>0 1 1 1 1 0 0 1</td>
<td>no overflow</td>
</tr>
<tr>
<td>-63:</td>
<td>1 1 0 0 0 0 0 1</td>
<td>no overflow</td>
</tr>
<tr>
<td>-32:</td>
<td>1 1 1 0 0 0 0 0</td>
<td>no overflow</td>
</tr>
<tr>
<td>-95:</td>
<td>1 0 1 0 0 0 0 1</td>
<td>no overflow</td>
</tr>
<tr>
<td>-42:</td>
<td>1 1 0 1 0 1 1 0</td>
<td>no overflow</td>
</tr>
<tr>
<td>-34:</td>
<td>1 1 0 1 1 1 1 0</td>
<td>no overflow</td>
</tr>
</tbody>
</table>

\[ 105: \quad 0 1 1 0 1 0 0 1 \]

<table>
<thead>
<tr>
<th>Example</th>
<th>Signed Numbers</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>72:</td>
<td>0 1 0 0 1 0 0 0</td>
<td>positive overflow</td>
</tr>
<tr>
<td>105:</td>
<td>0 1 1 0 1 0 0 1</td>
<td>negative overflow</td>
</tr>
</tbody>
</table>

\[ 96: \quad 0 1 0 0 1 0 0 0 \]

\[ 34: \quad 0 0 1 0 0 0 1 0 \]
### Signed Addition in VHDL

- **Result of + is same size as operands**

```vhdl
signal v1, v2 : signed(11 downto 0);
signal sum : signed(12 downto 0);
...
sum <= resize(v1, sum'length) + resize(v2 sum'length);
```

- **To check overflow, compare signs**

```vhdl
signal x, y, z: signed(7 downto 0);
signal ovf : std_logic;
...
z <= x + y;
ovf <= (not x(7) and not y(7) and z(7))
or (x(7) and y(7) and not z(7));
```

### Signed Subtraction

\[ x - y = x + (-y) = x + \overline{y} + 1 \]

- **Use a 2s-complement adder**
  - Complement \( y \) and set \( c_0 = 1 \)
Other Signed Operations

- Increment, decrement
  - same as unsigned
- Comparison
  - =, same as unsigned
  - >, compare sign bits using $x_{n-1} \cdot y_{n-1}$
- Multiplication
  - Complicated by the need to sign extend partial products
  - Refer to Further Reading

Scaling Signed Integers

- Multiplying by $2^k$
  - logical left shift (as for unsigned)
  - truncate result using 2s-complement rules
- Dividing by $2^k$
  - arithmetic right shift
  - discard $k$ bits from the right, and replicate sign bit $k$ times on the left
  - e.g., $s = "11110011" \rightarrow \frac{-13}{2^2}$
Fixed-Point Numbers

- Many applications use non-integers
  - especially signal-processing apps
- Fixed-point numbers
  - allow for fractional parts
  - represented as integers that are implicitly scaled by a power of 2
  - can be unsigned or signed

Positional Notation

- In decimal
  \[10.24_{10} = 1 \times 10^1 + 0 \times 10^0 + 2 \times 10^{-1} + 4 \times 10^{-2}\]
- In binary
  \[101.01_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 5.25_{10}\]
- Represent as a bit vector: 10101
  - binary point is implicit
Unsigned Fixed-Point

- $n$-bit unsigned fixed-point
  - $m$ bits before and $f$ bits after binary point
  \[ x = x_{m-1}2^{m-1} + \cdots + x_02^0 + x_{-1}2^{-1} + \cdots + x_{-f}2^{-f} \]
  - Range: 0 to $2^m - 2^{-f}$
  - Precision: $2^{-f}$
  - $m$ may be $\leq 0$, giving fractions only
    - e.g., $m = -2$: 0.0001001101

Signed Fixed-Point

- $n$-bit signed 2s-complement fixed-point
  - $m$ bits before and $f$ bits after binary point
  \[ x = -x_{m-1}2^{m-1} + \cdots + x_02^0 + x_{-1}2^{-1} + \cdots + x_{-f}2^{-f} \]
  - Range: $-2^{m-1}$ to $2^{m-1} - 2^{-f}$
  - Precision: $2^{-f}$
  - E.g., 111101, signed fixed-point, $m = 2$
    - $11.1101_2 = -2 + 1 + 0.5 + 0.25 + 0.0625 = -0.1875_{10}$
Choosing Range and Precision

- Choice depends on application
- Need to understand the numerical behavior of computations performed
  - some operations can magnify quantization errors
- In DSP
  - fixed-point range affects dynamic range
  - precision affects signal-to-noise ratio
- Perform simulations to evaluate effects

Fixed-Point in VHDL

- Use numeric_bit with implied scaling
- Use proposed fixed_pkg package
  - Currently being standardized by IEEE
  - Types ufixed and sfixed
  - Arithmetic operations, resizing, conversion

```vhdl
library ieee_proposed; use ieee_proposed.fixed_pkg.all;
entity fixed_converter is
  port ( input : in ufixed(5 downto -7);
          output : out sfixed(7 downto -7) );
end entity fixed_converter;
```
Fixed-Point Operations

- Just use integer hardware
  - e.g., addition:
    \[ x + y = (x \times 2^f + y \times 2^f) / 2^f \]
  - Ensure binary points are aligned

Floating-Point Numbers

- Similar to scientific notation for decimal
  - e.g., 6.02214199 \times 10^{23}, 1.60217653 \times 10^{-19}
- Allow for larger range, with same relative precision throughout the range

6.02214199 \times 10^{23}

mantissa \quad \text{radix} \quad \text{exponent}
### IEEE Floating-Point Format

<table>
<thead>
<tr>
<th>$e$ bits</th>
<th>$m$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>s exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>

$$x = M \times 2^E = (-1 \times s) \times 1.mantissa \times 2^{exponent - 2^{e-1} + 1}$$

- **s**: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- **Normalize**: $1.0 \leq |M| < 2.0$
  - $M$ always has a leading pre-binary-point 1 bit, so no need to represent it explicitly *(hidden bit)*
  - Exponent: excess representation: $E + 2^{e-1} - 1$

### Floating-Point Range

- **Exponents 000...0 and 111...1 reserved**
- **Smallest value**
  - exponent: 000...01 ⇒ $E = -2^{e-1} + 2$
  - mantissa: 0000...00 ⇒ $M = 1.0$
- **Largest value**
  - exponent: 111...10 ⇒ $E = 2^{e-1} - 1$
  - mantissa: 111...11 ⇒ $M \approx 2.0$
- **Range**: $2^{-2^{e-1}+2} \leq |x| < 2^{2^{e-1}}$
Floating-Point Precision

- Relative precision approximately $2^{-m}$
  - all mantissa bits are significant
- $m$ bits of precision
  - $m \times \log_{10}2 \approx m \times 0.3$ decimal digits

Example Formats

- IEEE single precision, 32 bits
  - $e = 8$, $m = 23$
  - range $\approx \pm1.2 \times 10^{-38}$ to $\pm1.7 \times 10^{38}$
  - precision $\approx 7$ decimal digits
- Application-specific, 22 bits
  - $e = 5$, $m = 16$
  - range $\approx \pm6.1 \times 10^{-5}$ to $\pm6.6 \times 10^{4}$
  - precision $\approx 5$ decimal digits
Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0
  \[ x = M \times 2^E = (-1 \times s) \times 0.mantissa \times 2^{-2^e+1} \]
- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Mantissa = 000...0
  \[ x = M \times 2^E = (-1 \times s) \times 0.0 \times 2^{-2^e+1} = \pm 0.0 \]

Infinities and NaNs

- Exponent = 111...1, mantissa = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, mantissa ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations
Floating-Point Operations

- Considerably more complicated than integer operations
  - E.g., addition
    - unpack, align binary points, adjust exponents
    - add mantissas, check for exceptions
    - round and normalize result, adjust exponent
- Combinational circuits not feasible
  - Pipelined sequential circuits

Floating-Point in VHDL

- Use proposed float_pkg package
  - Currently being standardized by IEEE
  - Types float, float32, float64, float128
  - Arithmetic operations, resizing, conversion
- Not likely to be synthesizable
  - Rather, use to verify results of hand-optimized circuits
Summary

- Unsigned: \( x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \)
- Signed: \( x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_02^0 \)
- Octal and Hex short-hand
- Operations: resize, arithmetic, compare
- Arithmetic circuits trade off speed/area/power
- Fixed- and floating-point non-integers
- Gray codes for position encoding