Processing Elements
Introduction

- Number Systems
- Basic arithmetic operations
  - Addition
  - Subtraction
  - Multiplication
- Compound operations
  - Multiple-constant multiplication
  - Vector multiplication
  - Multiplication involving complex values
Conventional Number Systems

Signed-magnitude representation

- MSB represents “sign”, other bits represent “magnitude”

\[ x = (1 - 2x_0) \sum_{i=1}^{W_d-1} x_i 2^{-i}, \quad -1 + 2^{-W_d+1} \leq x \leq 1 - 2^{-W_d+1} \]

1’s-complement representation

- \((+0.828125) = (0.110101)\)
- \((-0.828125) = (1.001010)\)

\[ x = -x_0 (1 - 2^{-W_d+1}) + \sum_{i=1}^{W_d-1} x_i 2^{-i}, \quad -1 + 2^{-W_d+1} \leq x \leq 1 - 2^{-W_d+1} \]
Conventional Number Systems (cont’d)

- **2’s-complement representation**
  \[ x = -x_0 + \sum_{i=1}^{W_d-1} x_i 2^{-i}, \quad -1 \leq x \leq 1 - 2^{-W_d+1} \]

- **Binary offset representation**
  \[ x = x_0 - 1 + \sum_{i=1}^{W_d-1} x_i 2^{-i}, \quad -1 \leq x \leq 1 - 2^{-W_d+1} \]
Show by an example that the sum of \( n \) (\( n \geq 3 \)) numbers in 2’s-complement representation can be added without regard to overflows of the partial sums if the final sum is within the number range.

Assume that the numbers are
\[
6/8 = (0.110)_{2c}, 4/8 = (0.100)_{2c}, \text{ and } -7/8 = (1.001)_{2c}
\]

We first add
\[
S_1 = 6/8 + 4/8 = (0.110)_{2c} + (0.100)_{2c} = (1.010)_{2c} = -6/8
\]
and then we add the third number (neglecting the temporal occurred and neglecting the second overflow)
\[
S_2 = -6/8 + (-7/8) = (1.010)_{2c} + (1.001)_{2c} = (0.011)_{2c} = 3/8
\]
yielding the correct result.

Figure illustrates the cyclic proper 2’s-complement representation.
In application-specific arithmetic units, it is often advantageous to use more unconventional, redundant number systems:

- Signed-digit code
- Canonic signed digit code
- On-line arithmetic
Signed-Digit Code

\[ x = \sum_{i=1}^{w_d-1} x_i 2^{-i}, \quad \text{where} \quad x_i = -1, 0, +1 \]

\[-2 + 2^{-W_d+1} \leq x \leq 2 - 2^{-W_d+1}\]

Not unique

\[(15/32)_{10} = (0.01111)_2C \quad \text{or} \quad (0.1000\bar{1})\]

Advantage:
addition is independent of word length
Canonic Signed Digit Code

- No two consecutive digits are nonzero
- Convert 2’s-complement to CSDC
  - $0111111100 \rightarrow 1000000100$
  - $(1, -1) \rightarrow (0, 1) (0, 1, 1) \rightarrow (1, 0, -1)$
  - Iteration
- Convert SDC to 2’s-complement
  - One part: 0 or 1,
    - Another part: -1
  - Subtract toe two part
On-Line Arithmetic

- Compute the i-th digit of the result using only the first (i+δ)-th digits
- Latency corresponds to δ digits
- Favorable in recursive
Residue Number Systems (RNS)

\[ x = q_i m_i + r_i = (r_1, r_2, \ldots, r_p) \]

(+, -, *) can be performed for each residue independently of all the other residues

- Modules set: 5, 3, 2
- \( 9+19 = (4, 0, 1)_{\text{RNS}} + (4, 1, 1)_{\text{RNS}} = (3, 1, 0)_{\text{RNS}} = 28 \)
Bit-Parallel Arithmetic

- Addition and Subtraction
- Bit-Parallel Multiplication
- Shift-and Add Multiplication
- Booth’s Algorithm
- Tree-Based Multipliers
- Array Multipliers
- Look-up Table Techniques
Ripple Carry Adder (CSA)

Carry propagation time: $O(W_d)$
Carry Look Ahead Adder (CLA)

- Compute all carries first
- Carry propagation time: $O(\log_2(W_d))$
Three or more operand are to be added
Reduce the number of operands by one for each stage
Pre-computation partial results for two possible values of the carry bit
Example

Fig. 8.7 Carry-propagate adder (CPA) and carry-save adder (CSA) functions in dot notation.

Fig. 8.9 Tree of carry-save adders reducing seven numbers to two.

Fig. 8.10 Addition of seven 6-bit numbers in dot notation.

Fig. 8.11 Adding seven k-bit numbers and the CSA/CPA widths required.
Carry Skip Adder

- Divided into RCA blocks
- If a block is in the propagate state,
  \[ \text{carryin} = \text{carrou}= \] carry skips the block
Conditional Sum Adder

- Similar to CSA
- Use modified half-adder to generate sum, carries
- Use multiplexers to combine neighboring bits into final sum
Bit-Parallel Multiplication

- Shift-and-Add multiplier
- Parallel multiplier
  - Partial product generation
  - Carry-free addition
  - Carry-propagation addition
- Array Multiplier
Shift-and-Add Multiplication

Use only one bit-parallel adder: $W_d \times W_d$
Booth’s Algorithm

\[ x \times y = \sum_{i=1}^{8} \left[ x_{2i-1} + x_{2i} - 2x_{2(i-1)} \right] y 2^{-2i+1} \]

-2y, -y, 0, y, 2y

Reduce the number of add/sub to \( W_d / 2 \)

<table>
<thead>
<tr>
<th>X_{2i-2}</th>
<th>X_{2i-1}</th>
<th>X_{2i}</th>
<th>f(i)</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Tree-Based Multipliers

- Full-adder $\rightarrow$ (3, 2) counter
- Wallace Tree
- $O(\log_2(W_d))$
Array Multipliers

Baugh-Wooley, 2W_d

\[ P = x \cdot y = (-x_0 + \sum_{i=1}^{W_d-1} x_i 2^{-i})(-y_0 + \sum_{i=1}^{W_d-1} y_i 2^{-i}) \]

\[ = 2 + 2^{-W_d+2} + x_0y_0 + \sum_{i=1}^{W_d-1} \sum_{j=1}^{W_d-1} x_i y_j 2^{-i-j} + \sum_{i=1}^{W_d-1} x_0 y_i 2^{-i} + \sum_{i=1}^{W_d-1} y_0 x_i 2^{-i} \]
Look-up Table Techniques

\[ x \cdot y = \frac{(x + y)^2}{4} - \frac{(x - y)^2}{4} \]

- One addition, two subtractions, two look-up table
- \(2^{W_d+W_c} \rightarrow 2^{W_d} \) words
Bit-Serial Arithmetic

- Advantage
  - Chip area reduction
    - Eliminate wide bus
    - Simplify wire routing
  - Support higher clock frequency
Bit-Serial Addition & Subtraction
Bit-Serial Multiplication

- Serial/Parallel Multiplier
- Transposed Serial/Parallel Multiplier
- S/P Multiplier-Accumulator
Clock cycle = AND + FA
Total clock cycles = \( W_d + W_c - 1 \)
For unsigned case!!!
Example

For signed multiplication

- Do sign-extension to avoid errors!
Example (cont’d)

\[ y = ax = a \{-x_0 + \sum_{i=1}^{5} x_i 2^{-i} \} \]

\[ y_{Mult} = a \{x_0 2^4 + x_0 2^3 + x_0 2^2 + x_0 2^1 + x_0 2^0 + \sum_{i=1}^{5} x_i 2^{-i} \} \]

\[ = a \{x_0 2^4 + x_0 2^3 + x_0 2^2 + x_0 2^1 + x_0 2^1 - x_0 + \sum_{i=1}^{5} x_i 2^{-i} \} \]

\[ = ax_0 2^5 + ax \]

Error term!
Serial/Parallel Multiplier

- First stage is a HA as subtractor
- No sign-extension
Transposed Serial/Parallel Multiplier

- Add bit-products column-wise

Diagram showing the transposed serial/parallel multiplier with the bit-products column-wise addition and a note to reverse the order.
S/P Multiplier-Accumulator
Bit-Serial Two-Port Adaptor

Latency = 2
S/P Converter with Overflow Correction

- \textit{sign=guard}
  - No action
- \textit{sign}=1, \textit{guard}=0
  - Positive overflow
- \textit{sign}=0, \textit{guard}=1
  - Negative overflow
Magnitude Truncation

- Quantization procedure
  - Truncate to W bits
  - Add sign bit to LSB
S/P Multipliers with Fixed Coefficients

[Diagram of S/P multipliers with fixed coefficients]
Example

Example 1:

- Number of FA = (number of 1’s in coefficient – 1)
- Number of D flip-flop = number of 1-bit positions between first and last bit position
Example (cont’d)

\[ A = (0.00111)_2 = (0.0100-1)_{\text{CSDC}} \]
Minimum Number of Basic Operations

Multiplication can be implemented by
- Addition
- Subtraction
- Shift
Example

\[ C_1 = 9 \times 8 + 3 = (1 + 2^3)2^3 + (1 + 2^1) \]

\[ C_2 = 5 \times 15 = (1 + 2^2)(-1 + 2^4) \]
Multiple-Constant Multiplication

Reduce number of basic operations by factoring out common factors in the coefficient
Example

\[ y_1 = 7x, \ y_2 = 106x \]
Bit-Serial Squarer

\[ x = \sum_{i=1}^{n} x_i 2^{-i} \]

\[ f_1(x) = \left( \sum_{i=1}^{n} x_i 2^{-i} \right)^2 = (x_1 2^{-1} + \sum_{i=2}^{n} x_i 2^{-i})^2 \]

\[ f_1(x) = x_1 2^{-2} + x_1 2^{0} \sum_{i=1}^{n} x_i 2^{-i} + f_1(\sum_{i=2}^{n} x_i 2^{-i}) \]

\[ f_j = \Delta_j + f_{j+1} \]

\[ \Delta_j = 2^{-2j} x_j + 2^{1-j} x_j \sum_{i=j+1}^{n} x_i 2^{-i} \]
Squarer for Unsigned Numbers
Squarer for Sign-Mag. Number
Squarer for 2’s-Complement
Serial/Serial Multipliers

\[ f_1 = \left( \sum_{i=1}^{n} x_i 2^{-i} \right) \left( \sum_{i=1}^{n} y_i 2^{-i} \right) \]

\[ = \left( x_1 2^{-1} + \sum_{i=2}^{n} x_i 2^{-i} \right) \left( y_1 2^{-1} + \sum_{i=2}^{n} y_i 2^{-i} \right) \]

\[ = x_1 y_1 2^{-2} + x_1 2^{-1} \sum_{i=2}^{n} y_i 2^{-i} + y_1 2^{-1} \sum_{i=2}^{n} x_i 2^{-i} + f_2 \]
Digit-Serial Arithmetic

- Digit-serial processing
  - Process several bits at a time

- Digit size
  - Number of bits processed in a clock cycle
CORDIC Algorithm

- Compute magnitude, phase, rotation of a vector iteratively
- For circular, linear, hyperbolic coordinate systems
- Digit-by-digit algorithm with linear convergence and sequential behavior
  - W iterations are needed for W-bit precision approximately
CORDIC Algorithm (cont’d)

\[ x_{n+1} = x_n + \sigma_n 2^{-S(m,n)} y_n \]
\[ y_{n+1} = y_n + \sigma_n 2^{-S(m,n)} x_n \]
\[ z_{n+1} = z_n - \sigma_n \alpha(m,n) \]

\[ m: \text{coordinate system} \]
\[ \sigma: \text{rotation direction} \]
\[ S: \text{shift sequence} \]
\[ \alpha: \text{rotation angle} \]
CORDIC Process

Concept

\[
\begin{bmatrix}
x_f \\
y_f
\end{bmatrix} = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
x_i \\
y_i
\end{bmatrix} = \cos \theta \begin{bmatrix}
1 & -\tan \theta \\
\tan \theta & 1
\end{bmatrix} \begin{bmatrix}
x_i \\
y_i
\end{bmatrix}
\]

For elementary rotation

\[
\begin{bmatrix}
x_1 \\
y_1
\end{bmatrix} = \cos \theta_0 \begin{bmatrix}
1 & -\tan \theta_0 \\
\tan \theta_0 & 1
\end{bmatrix} \begin{bmatrix}
x_0 \\
y_0
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_2 \\
y_2
\end{bmatrix} = \cos \theta_1 \begin{bmatrix}
1 & -\tan \theta_1 \\
\tan \theta_1 & 1
\end{bmatrix} \begin{bmatrix}
x_1 \\
y_1
\end{bmatrix} = \cos \theta_1 \cos \theta_0 \begin{bmatrix}
1 & -\tan \theta_0 \\
\tan \theta_0 & 1
\end{bmatrix} \begin{bmatrix}
x_0 \\
y_0
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_f \\
y_f
\end{bmatrix} = \cos \theta_{n-1} \ldots \cos \theta_0 \begin{bmatrix}
1 & -\tan \theta_{n-1} \\
\tan \theta_{n-1} & 1
\end{bmatrix} \ldots \begin{bmatrix}
x_0 \\
y_0
\end{bmatrix}
\]
CORDIC Algorithm (cont’d)

\[
\theta = \sum_{i=0}^{n-1} \mu_i a_{m(i)}
\]

\[
\mu_i = \pm 1 \quad \text{depends on modes}
\]

\[
a_{m(i)} = \text{shift right}(2^{-s(m,i)})
\]

\[
\text{depends on mode.}
\]

\[
\mu \text{ mode}
\]

\[
\text{for } i = 0 \text{ to } n-1 \text{ Do}
\]

\[
\begin{bmatrix}
  x(i+1) \\
  y(i+1)
\end{bmatrix} =
\begin{bmatrix}
  1 & -m\mu_i 2^{-s(m,i)} \\
  \mu_i 2^{-s(m,i)} & 1
\end{bmatrix}
\begin{bmatrix}
  x(i) \\
  y(i)
\end{bmatrix}
\]

\[
z(i+1) = z(i) - \mu_i a_{m(i)}
\]

end loop

\[
\begin{bmatrix}
  x_f \\
  y_f
\end{bmatrix} =
\sum_{i=0}^{n-1} \frac{1}{\sqrt{1 + m\mu_i^2 2^{-2s(m,i)}}}
\begin{bmatrix}
  x(n) \\
  y(n)
\end{bmatrix}
\]

or

\[
x(i+1) = x(i) - \text{shift } y(i)
\]

\[
y(i+1) = \text{shift } x(i) + y(i)
\]

\[
z(i+1) = z(i) - \mu_i a_{m(i)}
\]
CORDIC Process (cont’d)

\[ \mu_i = \text{sign of } z(i) \]

vector rotation mode

\[ = \text{sign of } x(i) y(i) \]

angle accumulation mode.

(a)

(b)

(c)
CORDIC Process (cont’d)

📖 Applications

♦ Linear Transformation
  ➢ DFT, DHT, FFT

♦ Digital Filter
  ➢ Lattice filter

♦ Matrix-based DSP Algorithm
  ➢ QR, Kalman filter, linear solver, SVD

📖 Ex:

\[
Y(k) = X(0)e^{-\frac{j2\pi 0k}{N}} + X(1)e^{-\frac{j2\pi k}{N}} + \ldots
\]

\[
X(m)e^{-\frac{j2\pi mk}{N}} + \ldots + X(N - 1)e^{-\frac{j2\pi (N-1)k}{N}}
\]

\[
\begin{bmatrix}
Y_r(m+1,k) \\
Y_i(m+1,k)
\end{bmatrix} = K_1(n) \begin{bmatrix}
\cos \frac{2\pi mk}{N} & -\sin \frac{2\pi mk}{N} \\
\cos \frac{2\pi mk}{N} & \sin \frac{2\pi mk}{N}
\end{bmatrix} \begin{bmatrix}
X_r(m) \\
X_i(m)
\end{bmatrix} + \begin{bmatrix}
Y_r(m,k) \\
Y_i(m,k)
\end{bmatrix}
\]

\[
Y_k = \frac{Y(N,K)}{K_1(n)}
\]
**Mode of Operation**

- **Vector mode**
  - Vectoring mode, Forward rotation mode
  - $\theta$ is given
  - To complete the final value $[x_f \ y_f]^t$

- **Rotation mode**
  - Y-reduction mode, backward rotation mode
  - Calculate the angle of $[x_0 \ y_0]^t$

**Advantages**

- Shift-and-add only
- $nN+s$ time
- Ease to pipelining or parallel
CORDIC Process (cont’d)

Each CORDIC iteration involves two shifts, a table lookup, and three additions.

Fig. 22.3  Hardware elements needed for the CORDIC method.