Chapter 3 – Arithmetic
Chapter Contents

3.1 Fixed Point Addition and Subtraction
3.2 Fixed Point Multiplication and Division
3.3 Floating Point Arithmetic
3.4 High Performance Arithmetic
Computer Arithmetic

• Using number representations from Chapter 2, we will explore four basic arithmetic operations: addition, subtraction, multiplication, and division.

• Significant issues include: fixed point vs. floating point arithmetic, overflow and underflow, handling of signed numbers, and performance.

• We look first at fixed point arithmetic, and then at floating point arithmetic.
Binary Numbers can be Represented on a Number Circle

- Numbers are added or subtracted on the number circle by traversing clockwise for addition and counterclockwise for subtraction.

- Unlike the number line (a) where overflow never occurs, overflow occurs when a transition is made from +3 to -4 while proceeding around the number circle when adding, or from -4 to +3 while subtracting.
Overflow

• Overflow occurs when the addition of two positive numbers produces a negative result, or when the addition of two negative numbers produces a positive result. Adding operands of unlike signs never produces an overflow.

• Notice that discarding the carry out of the most significant bit during two’s complement addition is a normal occurrence, and does not by itself indicate overflow.

• As an example of overflow, consider adding \((80 + 80 = 160)_{10}\), which produces a result of \(-96_{10}\) in an 8-bit two’s complement format:

\[
\begin{align*}
01010000 & = 80 \\
+ 01010000 & = 80 \\
\hline
10100000 & = -96\ (not\ 160\ because\ the\ sign\ bit\ is\ 1.)
\end{align*}
\]
Sign Extension

• The leftmost bit is assigned to the sign. What happens to the sign bit if we place a number into a larger or smaller word? For positive numbers, all that we need to do is pad the left side with 0s:

\[
\begin{align*}
8\text{-bit } 12_{10} & \quad \text{becomes} \quad 16\text{-bit } 12_{10} \\
00001100 & \quad \Rightarrow \quad 0000000000001100 \\
\uparrow & \quad \uparrow \\
\text{Sign} = \text{“+”} & \quad \text{Sign} = \text{“+”}
\end{align*}
\]

• For negative numbers, we cannot simply pad the left side with 0s because that would change the value of the number:

\[
\begin{align*}
8\text{-bit } -12_{10} & \quad \text{incorrectly} \quad \text{becomes} \quad 16\text{-bit } 244_{10} \\
11110100 & \quad \Rightarrow \quad 00000000011110100 \\
\uparrow & \quad \uparrow \\
\text{Sign} = \text{“-”} & \quad \text{Sign} = \text{“+”}
\end{align*}
\]
Sign Extension (continued)

- As it turns out, all that we need to do is copy the sign bit for as many places as there are to the left and the number will be correctly extended, regardless of the value of the sign. This is known as sign extension:

\[ \begin{align*}
\text{8-bit } &+12_{10} \quad \text{becomes} \quad 16\text{-bit } +12_{10} \\
00001100 \quad &\rightarrow \quad 00000000 \quad 00001100 \\
\uparrow \quad &\uparrow \\
\text{Sign } = \text{“+”} \quad &\text{New sign bit } = \text{“+”} \quad \text{Old sign bit is copied to the left}
\end{align*} \]

\[ \begin{align*}
\text{8-bit } &-12_{10} \quad \text{becomes} \quad 16\text{-bit } -12_{10} \\
11110100 \quad &\rightarrow \quad 11111111 \quad 11110100 \\
\uparrow \quad &\uparrow \\
\text{Sign } = \text{“-”} \quad &\text{New sign bit } = \text{“-”} \quad \text{Old sign bit is copied to the left}
\end{align*} \]
Sign Extension (continued)

• If we want to reduce the size of the word, we can simply remove bits on the left and the resulting sign will be correct, as long as the number can be represented in the remaining bits:

16-bit \(-12_{10}\) becomes 8-bit \(-12_{10}\)

\[\begin{array}{c}
111111111110100 \\
\uparrow \\
\text{Sign bit = “+”}
\end{array}\quad \rightarrow \quad \begin{array}{c}
11110100 \\
\uparrow \\
\text{New sign bit = “-”}
\end{array}\]

These 8 bits will be deleted
Ripple Carry Adder

- Two binary numbers $A$ and $B$ are added from right to left, creating a sum and a carry at the outputs of each full adder for each bit position.
Constructing Larger Adders

- A 16-bit adder can be made up of a cascade of four 4-bit ripple-carry adders.
Full Subtractor

• Truth table and schematic symbol for a ripple-borrow subtractor:

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>$bor_i$</th>
<th>$diff_i$</th>
<th>$bor_{i+1}$</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>

(bor_i) (a_i - b_i)
Ripple-Borrow Subtractor

- A ripple-borrow subtractor can be composed of a cascade of full subtractors.
- Two binary numbers $A$ and $B$ are subtracted from right to left, creating a difference and a borrow at the outputs of each full subtractor for each bit position.
Combined Adder/Subtractor

- A single ripple-carry adder can perform both addition and subtraction, by forming the two’s complement negative for $B$ when subtracting. (Note that +1 is added at $c_0$ for two’s complement.)
One’s Complement Addition

- An example of one’s complement integer addition with an end-around carry:

\[
\begin{align*}
10011 & \quad (-12)_{10} \\
+01101 & \quad (+13)_{10} \\
\hline
100000 & \quad \text{End-around carry} \\
+1 & \\
\hline
00001 & \quad (+1)_{10}
\end{align*}
\]
Number Circle (Revisited)

- Number circle for a three-bit signed one’s complement representation. Notice the two representations for 0.
End-Around Carry for Fractions

• The end-around carry complicates one’s complement addition for non-integers, and is generally not used for this situation.

• The issue is that the distance between the two representations of 0 is 1.0, whereas the rightmost fraction position is less than 1.

\[
\begin{align*}
0 & 1 & 0 & 1 & . & 1 & ( +5.5 )_{10} \\
+ & 1 & 1 & 1 & 0 & . & 1 & ( -1.0 )_{10} \\
\hline
1 & 0 & 1 & 0 & 0 & . & 0
\end{align*}
\]

\[
\begin{align*}
+ & & & & & & . & 1
\hline
0 & 1 & 0 & 0 & . & 1 & ( +4.5 )_{10}
\end{align*}
\]
Multiplication Example

- Multiplication of two 4-bit unsigned binary integers produces an 8-bit result.

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

\( (13)_{10} \quad \text{Multiplicand M} \quad (11)_{10} \quad \text{Multiplier Q} \quad (143)_{10} \quad \text{Product P} \)

- Multiplication of two 4-bit signed binary integers produces only a 7-bit result (each operand reduces to a sign bit and a 3-bit magnitude for each operand, producing a sign-bit and a 6-bit result).
A Serial Multiplier

Multiplicand (M)

\[ m_3 \ m_2 \ m_1 \ m_0 \]

4-Bit Adder

Add

Shift Right

C

A Register

4

\[ q_3 \ q_2 \ q_1 \ q_0 \]

Multiplier (Q)

Shift and Add Control Logic

4

\[ q_0 \]
Example of Multiplication Using Serial Multiplier

**Multiplicand (M):**

Initial values

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 0 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

0 1 1 0 1 1 0 1 1   Add M to A
0 0 1 1 0 1 1 0 1   Shift
1 0 0 1 1 1 1 0 1   Add M to A
0 1 0 0 1 1 1 0   Shift
0 0 1 0 0 1 1 1   Shift (no add)
1 0 0 0 1 1 1 1   Add M to A
0 1 0 0 0 1 1 1   Shift

**Product**
Example of Base 2 Division

- $(7 / 3 = 2)_{10}$ with a remainder $R$ of 1.
- Equivalently, $(0111 / 11 = 10)_{2}$ with a remainder $R$ of 1.

```
  0 0 1 0  R 1
1 1 | 0 1 1 1
  1 1
  0 1
```
Serial Divider

Divisor (M)

0 \[ m_3 \ m_2 \ m_1 \ m_0 \]

5-Bit Adder

5

Add / Sub

Shift Left

Shift and Add / Sub Control Logic

q_0

A Register

Dividend (Q)

a_4 \ a_3 \ a_2 \ a_1 \ a_0

q_3 \ q_2 \ q_1 \ q_0

Computer Architecture and Organization by M. Murdocca and V. Heuring © 2007 M. Murdocca and V. Heuring
Division Example Using Serial Divider

Divisor (M):

\[
\begin{array}{c}
\text{A} \\
0 0 0 0 0 \\
0 0 0 1 1
\end{array}
\quad \begin{array}{c}
\text{Q} \\
0 1 1 1 1
\end{array}
\]

Initial values

<table>
<thead>
<tr>
<th>A</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>0 1 1 1 1</td>
</tr>
</tbody>
</table>

- Shift left
- Subtract M from A
- Restore A (Add M to A)
- Clear q₀

\[
\begin{array}{c}
\text{A} \\
0 0 0 0 1 \\
0 0 0 1 1
\end{array}
\quad \begin{array}{c}
\text{Q} \\
1 1 0 0 0
\end{array}
\]

- Shift left
- Subtract M from A
- Restore A
- Clear q₀

\[
\begin{array}{c}
\text{A} \\
0 0 0 1 1 \\
0 0 0 0 0
\end{array}
\quad \begin{array}{c}
\text{Q} \\
1 0 0 0 0
\end{array}
\]

- Shift left
- Subtract M from A
- Set q₀

\[
\begin{array}{c}
\text{A} \\
0 0 0 0 1 \\
0 0 0 0 0
\end{array}
\quad \begin{array}{c}
\text{Q} \\
0 0 1 0 1
\end{array}
\]

- Shift left
- Subtract M from A
- Restore A
- Clear q₀

Remainder: 0 0 1 0
Quotient: 0 0 1 0
Multiplication of Signed Integers

- Sign extension to the target word size is needed for the negative operand(s).

- A target word size of 8 bits is used here for two 4-bit signed operands, but only a 7-bit target word size is needed for the result.

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\times & 0 & 0 & 0 & 1 \\
\hline
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & (15)_{10}
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\times & 0 & 0 & 0 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & (1)_{10}
\end{array}
\]

Incorrect; result should be \(-1\)
Carry-Lookahead Addition

\[ s_i = \overline{a_i} \overline{b_i} c_i + \overline{a_i} b_i \overline{c_i} + a_i \overline{b_i} c_i + a_i b_i c_i \]

\[ c_{i+1} = b_i c_i + a_i c_i + a_i b_i \]

\[ c_{i+1} = a_i b_i + (a_i + b_i)c_i \]

\[ c_{i+1} = G_i + P_i c_i \]

\[ G_i = a_i b_i \quad \text{and} \quad P_i = a_i + b_i \]

\[ c_0 = 0 \]

\[ c_1 = G_0 \]

\[ c_2 = G_1 + P_1 G_0 \]

\[ c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 \]

\[ c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \]

- Carries are represented in terms of \( G_i \) (generate) and \( P_i \) (propagate) expressions.
Carry Lookahead Adder

- Maximum gate delay for the carry generation is only 3. The full adders introduce two more gate delays. Worst case path is 5 gate delays.
Floating Point Arithmetic

• Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands.

• The exponents of the operands must be made equal for addition and subtraction. The fractions are then added or subtracted as appropriate, and the result is normalized.

• Ex: Perform the floating point operation: 
  \[(.101 \times 2^3 + .111 \times 2^4)_2\]

• Start by adjusting the smaller exponent to be equal to the larger exponent, and adjust the fraction accordingly. Thus we have \(.101 \times 2^3 = .010 \times 2^4\), losing \(.001 \times 2^3\) of precision in the process.

• The resulting sum is \((.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5\), and rounding to three significant digits, \(.100 \times 2^5\), and we have lost another \(0.001 \times 2^4\) in the rounding process.
Floating Point Arithmetic (Cont’)

• If we simply added the numbers using as much precision as we needed and then applied rounding only in the final normalization step, then the calculation would go like this:

\[
.101 \times 2^3 + .111 \times 2^4 = \\
.0101 \times 2^4 + .111 \times 2^4 = \\
1.0011 \times 2^4.
\]

• Normalizing yields \( .10011 \times 2^5 \), and rounding to three significant digits using the round to nearest even method yields \( .101 \times 2^5 \).

• Which calculation is correct \( .100 \times 2^5 \) or \( .101 \times 2^5 \)?

• According to the IEEE 754 standard, the final result should be the same as if the maximum precision needed is used before applying the rounding method, and so the correct result is \( .101 \times 2^5 \). So what do we do?
Guard, Round, and Sticky Bits

• This raises the issue of how to compute the intermediate results with sufficient accuracy and without requiring too much hardware, and for this we use guard, round, and sticky bits.

• For the previous example, applying guard (g) and round (r) bits with the round toward nearest even method, we have:

\[
\begin{align*}
0.101 \times 2^3 & + 0.111 \times 2^4 \\
\Rightarrow & \quad + 0.0101 \times 2^4 \\
& \quad + 0.111 \times 2^4 \\
& = 1.00110 \times 2^4
\end{align*}
\]

• Only one extra bit is needed for this intermediate result, the guard bit (g), but we also show the round bit (r = 0) to locate its position. As we shift the number to normalize, we set a sticky bit (s) if any of the shifted out bits are nonzero. For this case, there are no nonzero bits to the right of the r bit and so s = 0: (see next slide)
Guard, Round, and Sticky Bits (Cont’)

• Now for the rounding step: simply append the sticky bit to the right of the result before rounding. There is no tie as there would be for .100100 and so we round up, otherwise we would have rounded down to the closest even number (.100):

\[ 1.0011 \times 2^4 = 1.00110 \times 2^4 \]

\[ gr \]

0 is shifted out, so \( s = 0 \)

• For this case, the guard, round, and sticky bits changed our previous result. Note that if \( r \) is 0 instead of 1, so that the \( grs \) combination is 100, we would have rounded down to .100 because .100 is even whereas .101 is not.

\[ .100110 \times 2^5 \equiv .101 \times 2^5 \]

\[ grs \]
Floating Point Multiplication/Division

- Floating point multiplication/division are performed in a manner similar to floating point addition/subtraction, except that the sign, exponent, and fraction of the result can be computed separately.

- Like/unlike signs produce positive/negative results, respectively. Exponent of result is obtained by adding exponents for multiplication, or by subtracting exponents for division. Fractions are multiplied or divided according to the operation, and then normalized.

- Ex: Perform the floating point operation: \((+.110 \times 2^5) / (+.100 \times 2^4)_2\)

- The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is \(5 - 4 = 1\).

- We divide fractions, producing the result: \(110/100 = 1.10\).

- Putting it all together, the result of dividing \((+.110 \times 2^5)\) by \((+.100 \times 2^4)\) produces \((+1.10 \times 2^1)\). After normalization, the final result is \((+.110 \times 2^2)\).
The Booth Algorithm

- Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.

- Positive and negative numbers treated alike.

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 \\
\times & 0 & +1 & 0 & 0 & -1 & 0 \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[(21)_{10} \text{ Multiplicand} \]
\[(14)_{10} \text{ Multiplier} \]
\[(-21 \times 2)_{10} \]
\[(21 \times 16)_{10} \]
\[(294)_{10} \text{ Product} \]
A Worst Case Booth Example

A worst case situation in which the simple Booth algorithm requires twice as many additions as serial multiplication.

\[
\begin{array}{cccccc}
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 \\
\times & +1 & -1 & +1 & -1 & +1 & -1 \\
\hline
& \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \text{Subtract} & \text{Add} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & (-14 \times 1)_{10} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & (14 \times 2)_{10} \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & (-14 \times 4)_{10} \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & (14 \times 8)_{10} \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & (-14 \times 16)_{10} \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & (14 \times 32)_{10} \\
\hline
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & (294)_{10} & \text{Product}
\end{array}
\]
**Bit-Pair Recoding (Modified Booth Algorithm)**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>×</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ (21)_{10} \quad \text{Multiplicand} \]
\[ (14)_{10} \quad \text{Multiplier} \]
\[ \text{Booth recoded multiplier} \]
\[ \text{Bit pair recoded multiplier} \]
\[ (14 \times 1)_{10} \]
\[ (14 \times 4)_{10} \]
\[ (14 \times 16)_{10} \]
\[ (294)_{10} \quad \text{Product} \]
# Coding of Bit Pairs

<table>
<thead>
<tr>
<th>Booth pair $(i + 1, i)$</th>
<th>Recoded bit pair $(i)$</th>
<th>Corresponding multiplier bits $(i + 1, i, i - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>= 0</td>
<td>000 or 111</td>
</tr>
<tr>
<td>0 +1</td>
<td>= +1</td>
<td>001</td>
</tr>
<tr>
<td>0 −1</td>
<td>= −1</td>
<td>110</td>
</tr>
<tr>
<td>+1 0</td>
<td>= +2</td>
<td>011</td>
</tr>
<tr>
<td>+1 +1</td>
<td>= __</td>
<td></td>
</tr>
<tr>
<td>+1 −1</td>
<td>= +1</td>
<td>010</td>
</tr>
<tr>
<td>−1 0</td>
<td>= −2</td>
<td>100</td>
</tr>
<tr>
<td>−1 +1</td>
<td>= −1</td>
<td>101</td>
</tr>
<tr>
<td>−1 −1</td>
<td>= __</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Pipelined Array Multiplier
Newton’s Iteration for Zero Finding

• The goal is to find where the function $f(x)$ crosses the $x$ axis by starting with a guess $x_i$ and then using the error between $f(x_i)$ and zero to refine the guess.

• A three-bit lookup table for computing $x_0$:

<table>
<thead>
<tr>
<th>B = First three bits of $b$</th>
<th>Actual base 10 value of 1/B</th>
<th>Corresponding lookup table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>.100</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>.101</td>
<td>1 3/5</td>
<td>01</td>
</tr>
<tr>
<td>.110</td>
<td>1 1/3</td>
<td>01</td>
</tr>
<tr>
<td>.111</td>
<td>1 1/7</td>
<td>01</td>
</tr>
</tbody>
</table>

• The division operation $a/b$ is computed as $a \times 1/b$. Newton’s iteration provides a fast method of computing $1/b$. 
Residue Arithmetic

- Implements carryless arithmetic (thus fast!), but comparisons are difficult without converting to a weighted position code.

- Representation of the first twenty decimal integers in the residue number system for the given moduli:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Residue 5794</th>
<th>Decimal</th>
<th>Residue 5794</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>10</td>
<td>0312</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>11</td>
<td>1423</td>
</tr>
<tr>
<td>2</td>
<td>2222</td>
<td>12</td>
<td>2530</td>
</tr>
<tr>
<td>3</td>
<td>3333</td>
<td>13</td>
<td>3641</td>
</tr>
<tr>
<td>4</td>
<td>4440</td>
<td>14</td>
<td>4052</td>
</tr>
<tr>
<td>5</td>
<td>0551</td>
<td>15</td>
<td>0163</td>
</tr>
<tr>
<td>6</td>
<td>1662</td>
<td>16</td>
<td>1270</td>
</tr>
<tr>
<td>7</td>
<td>2073</td>
<td>17</td>
<td>2381</td>
</tr>
<tr>
<td>8</td>
<td>3180</td>
<td>18</td>
<td>3402</td>
</tr>
<tr>
<td>9</td>
<td>4201</td>
<td>19</td>
<td>4513</td>
</tr>
</tbody>
</table>
Examples of Addition and Multiplication in the Residue Number System

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Residue 5794</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>4121</td>
</tr>
<tr>
<td>27</td>
<td>2603</td>
</tr>
<tr>
<td>56</td>
<td>1020</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Residue 5794</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0312</td>
</tr>
<tr>
<td>17</td>
<td>2381</td>
</tr>
<tr>
<td>170</td>
<td>0282</td>
</tr>
</tbody>
</table>
16-bit Group Carry Lookahead Adder

- A 16-bit GCLA is composed of four 4-bit CLAs, with additional logic that generates the carries between the four-bit groups.

\[ GG_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \]
\[ GP_0 = P_3P_2P_1P_0 \]
\[ c_4 = GG_0 + GP_0c_0 \]
\[ c_8 = GG_1 + GP_1c_4 = GG_1 + GP_1GG_0 + GP_1GP_0c_0 \]
\[ c_{12} = GG_2 + GP_2c_8 = GG_2 + GP_2GG_1 + GP_2GP_1GG_0 + GP_2GP_1GP_0c_0 \]
\[ c_{16} = GG_3 + GP_3c_{12} = GG_3 + GP_3GG_2 + GP_3GP_2GG_1 + GP_3GP_2GP_1GG_0 + GP_3GP_2GP_1GP_0c_0 \]
16-Bit Group Carry Lookahead Adder

- Each CLA has a longest path of 5 gate delays.

- In the GCLL section, $GG$ and $GP$ signals are generated in 3 gate delays; carry signals are generated in 2 more gate delays, resulting in 5 gate delays to generate the carry out of each GCLA group and 10 gates delays on the worst case path (which is $s_{15}$ – not $c_{16}$).