Chapter 1 – Basic Number Representations and Arithmetic Algorithms

Arithmetic Processing

- **AP** = (operands, operation, results, conditions, singularities)
  - Operands are:
    - Set of numerical values
    - Range $\frac{V_{\text{min}}}{V_{\text{max}}} \leq x \leq V_{\text{max}}$
    - Precision (number of bits)
    - Number Representation System (NRS)
  - Operands: +, -, *, /, etc.
  - Conditions: Values of results (zero, neg, etc.)
  - Singularities: Illegal results (overflow, NaN, etc.)

Number Representation

- Need to map numbers to bits
  - (or some other representation, but we’ll use bits)
- Representation you choose matters!
  - Complexity of arithmetic operations depends heavily on representation!
  - But, be careful of conversions
    - Arithmetic that’s easy in one representation may lose its advantage when you convert back and forth…

Basic Fixed Point NRS

- Number represented by ordered n-tuple of symbols
  - Symbols are digits
  - n-tuple is a digit-vector
  - Number of digits in digit-vector is precision
  - $X = (X_{n-1}, X_{n-2}, \ldots, X_1, X_0)$

Digit Values

- A set of numerical values for the digits
  - $D_i$ is the set of possible values for $X_i$
  - $|D_i|$ is the cardinality of $D_i$
  - Binary (cardinality 2) is $\{0,1\}$
  - Decimal (cardinality 10) is $\{0,1,2,3,4,5,6,7,8,9\}$
  - Balanced ternary (cardinality 3) is $\{-1,0,1\}$
  - Set of integers represented by a digit-vector with n digits is a finite set with max elements of
    $$K = \left\lfloor \frac{\sum_{i=0}^{n-1} |D_i|}{2^n} \right\rfloor$$

Rule of Interpretation

- Mapping of set of digit-vectors to numbers
  - “Thirteen”
    - (1,3)
  - Digit-Vectors
  - Numbers (N, Z, R, …)
Mappings...

Digit-vectors: N.R.Z,...
- **Nonredundant**
- **Redundant**
- **Not Useful!**

Digit-vectors: N.R.Z,...
- **Ambiguous**

Positional Weighted Systems

- Integer \( x \) represented by digit vector
  \[ X = (X_{n-1}, X_{n-2}, ..., X_1, X_0) \]

- Rule of interpretation
  \[ X = \sum_{i=0}^{n-1} X_i W_i \]

- Where weight vector is
  \[ W = (W_{n-1}, W_{n-2}, ..., W_1, W_0) \]

Radix Number Systems

- Weights are not arbitrary
  - they are related to a radix vector
    \[ R = (R_{n-1}, R_{n-2}, ..., R_1, R_0) \]
  - So that
    \[ W_0 = 1; \quad W_i = W_{i-1} \cdot R_{i-1} \quad (1 \leq i \leq n - 1) \]
  - Or
    \[ W_0 = 1; \quad W_i = \prod_{j=0}^{i-1} R_j \]

Fixed-Radix Systems

- In fixed-radix system all elements of the radix vector have the same value \( r \) (the radix)
  - Weight vector is
    \[ W = (r, r, ..., r, r) \]
  - So
    \[ x = \sum_{i=0}^{n} X_i \cdot r^i \]

  - Radix 2: \( W = (...) 16, 8, 4, 2, 1 \)
  - Radix 4: \( W = (...) 256, 64, 16, 4, 1 \)
  - Radix 10: \( W = (...) 100000, 10000, 1000, 100, 10, 1 \)

Mixed-Radix Systems

- Time is the most common...
  - Hours, Minutes, Seconds
    \[ R = (24, 60, 60) \]
  - \( W = (3600, 60, 1) \)

  - \( X = (5,37,43) = 20,263 \) seconds
    - \( 5 \times 3600 = 18,000 \)
    - \( 37 \times 60 = 2,220 \)
    - \( 43 \times 1 = 43 \)
    - Total = 20,263 seconds

Canonical Systems

- Canonical if \( D_r = \{0,1, ..., R_r \} \) with \( D_r = R_i \)
  - Binary = \( \{0,1\} \)
  - Octal = \( \{0,1,2,3,4,5,6,7\} \)
  - Decimal = \( \{0,1,2,3,4,5,6,7,8,9\} \)

- Range of values with \( n \) radix-\( r \) digits is
  \[ 0 \leq x \leq r^n - 1 \]
Non-Canonical Systems

- Digit set that is non canonical...
  - Non-canonical decimal $D_i = \{-4,-3,-2,-1,0,1,2,3,4,5\}$
  - Non-canonical binary $D_i = \{-1,0,1\}$ or $\{0,1,2\}$
  - Redundant if non-canonical $D_i$ s.t. $|D_i| > R_i$
  - I.e. binary system with $D_i = \{-1,0,1\}$
    - $(1,1,0,1)$ and $(1,1,1,-1)$ both represent “thirteen”

Conventional Number Systems

- A system with fixed positive radix $r$ and canonical set of digit values
  - Radix-$r$ conventional number system
  - After all this fuss, these are what we’ll mostly worry about…
    - Specifically binary (radix 2)
  - We’ll also see some signed-digit redundant binary (carry-save, signed-digit)

Aside – Residue Numbers

- Example of a non-radix number system
  - Weights are not defined recursively
  - Residue Number System (RNS) uses a set of pairwise relatively prime numbers $P = (P_1, \ldots, P_n)$
  - A positive integer $x$ is represented by a vector $x \text{ s.t. } X_i = x \mod P_i$
  - Can allow fast add and multiply
  - No notion of digits on left being more significant than digits on the right (i.e. no weighting)

Aside – Residue Numbers

- $P = (17, 13, 11, 7, 5, 3, 2)$
  - Digit-vector $(13 4 8 2 0 0 0)$
  - Number = “thirty”
    - $30 \mod 17 = 13$
    - $30 \mod 13 = 4$
    - $30 \mod 11 = 8$
    - Etc…

Lots of Choices…

- Non-negative integers – digits $\{0,1\}$
- Range with $n$ bits is $0 \leq x \leq 2^n - 1$
- Higher power of 2 radix – group bits in to groups with $\log_2 r$ bits
  - $X = (1,1,0,0,0,1,0,1,1,0,1,1,0,1)$
    - $(1,1), (0,0), (0,1), (1,1), (0,1)$
    - $(3,0,1,1,3,1)$ base 4 (quaternary)
    - $(1,1,0,0,0,1,1,1,0,1,1)$
    - $(6,1,3,5)$ base 8 (octal)
    - $(1,1,0,0,0,1,1,1,0,1,1)$
    - $(C,5,D)$ base 16 (hexadecimal)
Signed Integers

- Three choices for representing signed ints
  - Directly in the number system
    - Signed digit NRS, i.e. {-1, 0, 1}
  - Use extra symbol to represent the sign
    - Sign and Magnitude
  - Additional mapping on positive integers
    - True and Complement system
      - Signed integer X
      - Positive Integer X
  - Digit-Vector X

Transformation

- Transform signed numbers into unsigned, then use conventional systems

True and Complement System

- Signed integers in the range $-k \leq x \leq k$
  - Negative represented by $x_{\text{R}}$
  - Such that $x_{\text{R}} = x \mod C$
  - C is the Complementation constant
  - Unambiguous if $k < C/2$

Converse Mapping

- Convert back and forth

$$x_{\text{R}} = \begin{cases} x & \text{if } x \geq 0 \\ C - |x| & \text{if } x < 0 \end{cases}$$

$$x = \begin{cases} x_{\text{R}} & \text{if } x_{\text{R}} < C/2 \\ x_{\text{R}} - C & \text{if } x_{\text{R}} \geq C/2 \end{cases}$$

Boundary conditions

- If $x_{\text{R}} = C/2$, can be represented, you can assign it to either $x = -C/2$ or $x = C/2$
  - Representation is no longer symmetric
  - Not closed under sign change operation

- If $x_{\text{R}} = C$, can be represented, then there are two representations of 0

Two standard forms

- Range complement system $C = p^n$
  - Also called Radix Complement
  - Two’s complement in radix 2

- Digit Complement System $C = p^n - 1$
  - Also called Diminished Radix Complement
  - One’s complement in radix 2
Two's Compliment

- For \( n \) bits in the digit vector, \( C = 2^n \)
  - Example three bits: \( C = 8 \)
- \( x = C \) is outside the range
  - With 3 bits, you can't represent 8
  - So, only one representation of 0
- \( x = C/2 \) can be represented, so you have a choice. \( x \) is either \( 2^{-1} \) or \( x = -2^{-1} \)
  - Usually choose second for sign detection
  - Range is then \( -2^{-1} \leq x \leq 2^{-1} - 1 \)
  (asymmetric)

One's Compliment

- For \( n \) bits in the digit vector \( C = 2^n - 1 \)
  - Example three bits: \( C = 7 \)
- \( x = C \) is representable in 3 bits
  - Two representations of 0
  - \( x = 0 \): \( x = 0 \) and \( x = 2^n - 1 \)
  - \( x = C/2 \) cannot be represented
  - Symmetric range...
  - Range is then \( -(2^{n-1} - 1) \leq x \leq 2^{n-1} - 1 \)

Examples (n=3 bits)

- Two's compliment
  - \( C = 2^3 \) \( x = -3 \) so \( x = C - x = 8 - 3 = 5 \)
  - -3 represented as 101
  - \( x = 111 = 7 \), \( 7 > C/2 \), \( x = C - 7 = 7 - 1 = 6 \)
  - 111 represents -1
- One's compliment
  - \( C = 2^3 \) \( x = -3 \) so \( x = C - x = 8 - 3 = 4 \)
  - -3 represented as 100
  - \( x = 111 = 7 \), \( 7 > C/2 \), \( x = C - 7 = 7 - 7 = 0 \)

Range Comparison (3 bits)

<table>
<thead>
<tr>
<th>Decimal (signed)</th>
<th>Binary (sign &amp; magnitude)</th>
<th>Two's compliment</th>
<th>One's compliment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>011</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>-2</td>
<td>010</td>
<td>010</td>
<td>001</td>
</tr>
<tr>
<td>-1</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>000/100</td>
<td>000/111</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>111</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: 2's comp, n=4

Example: 1's comp, n=4
Converse Mapping (2’s comp)

- If $x_k < C/2$, $X_{n-1} = 0$
  \[ x = -y = 0 \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i \]

- If $x_k \geq C/2$, $X_{n-1} = 1$
  \[ x = -y - 1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i = -2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i \]

Most significant bit has negative weight, remaining have positive

Converse Mapping (1’s comp)

- Similar in one’s complement (case for $x_{n-1} = 1$)
  \[ x = y_c - C = 1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i 2^i - (2^n - 1) \]
  \[ = -2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} X_i 2^i = -(2^{n-1} - 1) + \sum_{i=0}^{n-2} X_i 2^i \]

  \[ x = -X_{n-1} (2^{n-1} - 1) + \sum_{i=0}^{n-2} X_i 2^i \]

Remember this! We’ll use it later when we need to adjust things in arrays of signed addition. Think of partial product arrays in multiplication….

Two’s Comp. Example

- Most significant bit has negative weight, remaining have positive

  \[ n=5: \quad X = 11011 = -16 + 8 + 0 + 2 + 1 = -5 \]
  \[ X = 01011 = 0 + 8 + 0 + 2 + 1 = 11 \]

One’s Comp. Example...

- Most significant bit has negative weight, remaining have positive. Weight of MSB is different because $C=2^n-1$. Intuition is that you have to add 1 to jump over the extra representation of 0.

  \[ n=5: \quad X = 11011 = -(16-1) + 8 + 0 + 2 + 1 = -4 \]
  \[ X = 01011 = 0 + 8 + 0 + 2 + 1 = 11 \]

Sign Bits

- Conveniently, sign is determined by high-order bit $X_{n-1}$
  - Because $|y| \leq C/2$

  \[ \text{sign}(x) = \begin{cases} 0 & \text{if } x_k < C/2 \\ 1 & \text{if } x_k \geq C/2 \end{cases} \]

  (Assuming $x_k = C/2$ is assigned to represent $x = C/2$)

Addition (unsigned)

- Adding two n-bit operands results in n+1 result bits
  - Usually call the n+1 bit Cout

  \[ z = (x + y + c_m) \mod r^n \]

  \[ c_{out} = \begin{cases} 1 & \text{if } (x + y + c_m) \geq r^n \\ 0 & \text{otherwise} \end{cases} \]

  (In terms of digit vectors
  - $C_{out} = \text{overflow}$
  - $c_{out}(z) = ADD(X, Y, c_m)$

  \[ \sum_{k=0}^{n-1} x_k 2^k + \sum_{k=0}^{n-1} y_k 2^k + c_m = \sum_{k=0}^{n+1} z_k 2^k \]
Addition (Signed)

- Assume no overflow for a moment…
  - \( z = x + y \) computed as \( z_n = (x_n + y_n) \mod C \)
- Use the property \((a \mod C) \mod C = a \mod C\)
  \[
  (x_n + y_n) \mod C = ((x \mod C) + (y \mod C)) \mod C = (x + y) \mod C = z \mod C = z_n
  \]

Addition: two’s comp.

- \( C = 2^n \), and \( 2^n \mod 2^n \) means ignore \( X_n \) (the carry out)!
- Makes addition simple — add the numbers and ignore the carry out \( z_n = (x_n + y_n) \mod 2^n \)

Addition: one’s comp.

- \( C = r-1 \), so mod operation is not as easy
  - \( z_n = w_n \mod (2^n - 1) \)
    - if \( X_n + Y_n < 2^n - 1 \) then \( w_n = 0 \) and \( W_n \mod (2^n - 1) = W_n \)
    - if \( X_n + Y_n < 2^n - 1 \) then \( w_n = 0 \) and \( W_n \mod (2^n - 1) = 0 \)
    - if \( 2^n - 1 < X_n + Y_n \neq 2(2^n - 1) \)
      - then \( w_n = 1 \) and \( W_n \mod (2^n - 1) = W_n - 2^n + 1 \)
- So:
  - if \( w_n = 0 \) \( \Rightarrow Z_n = W_n \)
  - if \( w_n = 1 \) \( \Rightarrow Z_n = W_n - 2^n + 1 = W_n + 1 \)

Change of Sign

- Start with bitwise negation
  - Flip every bit in the digit vector
  - Boolean style: \( \overline{A} = (r-1) - a \)
  - Fundamental property of n-digit radix-r
    - \( A + \overline{A} + 1 = r^n \)
      \[
      \begin{array}{cccc}
      0 & 1 & 1 & 0 \\
      + & 1 & 0 & 0 \\
      & 1 & 1 & 1 \\
      & 0 & 0 & 1 \\
      & 1 & 0 & 0
      \end{array}
      \]

Addition: one’s comp.

- If the \( c_{\text{out}} \) is 1, subtract \( 2^n \) (ignore \( c_{\text{out}} \)), and add 1 (end-around carry)

Addition: one’s comp.

- If the \( c_{\text{out}} \) is 1, subtract \( 2^n \) (ignore \( c_{\text{out}} \)), and add 1 (end-around carry)

Change of Sign

- One’s Complement: \( -A = \overline{A} \)
  \[
  \begin{cases}
  -A = C - A = r^n - 1 - A = \overline{A} & \text{if } A \geq 0
  \end{cases}
  \]
- Two’s Complement: \( -A = \overline{A} + 1 \)
  \[
  \begin{cases}
  -A = C - A = r^n - A = \overline{A} + 1 & \text{if } A \geq 0
  \end{cases}
  \]
Another two's comp. check

- Verify the property \( A + \overline{A} = -1 \)
- Use two's complement definition...

\[
\left( -X_{n-1}2^{n-1} + \sum_{i=1}^{n} X_i2^i \right) + \left( -\overline{X}_{n-1}2^{n-1} + \sum_{i=1}^{n} \overline{X}_i2^i \right) = \\
-\left( X_{n-1} + \overline{X}_{n-1} \right) \cdot 2^{n-1} + \sum_{i=1}^{n} (X_i + \overline{X}_i) \cdot 2^i = \\
-2^{n-1} + \sum_{i=1}^{n} 2^i = -1
\]

Two's comp subtract

- \( X - Y = X + (-Y) = X + \overline{Y} + 1 \)

![Diagram of two's complement subtract]

Two's comp add/subtract

- \( X - Y = X + (-Y) = X + \overline{Y} + 1 \)

![Diagram of two's complement add/subtract]

Overflow (unsigned)

- Overflow condition means that the result can't be represented in \( n \) bits
  - For unsigned addition, this simply means that the cout was 1
  - For \( n=4 \), this means the result was bigger than 15
  - \( 1010_2 \ (10_{10}) + 1100_2 \ (12_{10}) = 10110_2 \ (22_{10}) \)

Overflow (signed)

- Still the same definition — the result can't be represented in \( n \) bits
  - But, not as easy as looking at cout
  - For 4 bits, and two's comp, answer was smaller than \(-8\) or larger than 7
  - Overflow if (pos) + (pos) = (neg) \( 5 + 6 = 11 \) or (neg) + neg) = (pos) \(-5 + -6 = -11\)
  - Can you ever have overflow with (pos) + (neg)?
Example: 2’s comp, n=4

Overflow (signed)

- Overflow only possible if args are same sign
- Overflow if result is different sign

\[ OVF \neq X'_{n-1} \cdot Y'_{n-1} \cdot Z_{n-1} \land X_{n-1} \cdot Y_{n-1} \cdot Z'_{n-1} \]

Or, consider all possible cases around MSB…

<table>
<thead>
<tr>
<th>X_{n-1}</th>
<th>Y_{n-1}</th>
<th>C_{n-1}</th>
<th>C_n</th>
<th>Z_{n-1}</th>
<th>OVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

\[ OVF = c_n \oplus c_{n-1} \]

Implied Digits (unsigned)

- Unsigned numbers have an infinite number of leading 0’s
- Changing from n bits to m bits (m>n) is a simple matter of padding with 0’s to the left

<table>
<thead>
<tr>
<th>X_{n-1}</th>
<th>Y_{n-1}</th>
<th>C_{n-1}</th>
<th>C_n</th>
<th>Z_{n-1}</th>
<th>OVF</th>
</tr>
</thead>
</table>

Changing number of bits (signed)

- Signed numbers can be thought of as having infinite replicas of the sign bit to the left
  - Four bits: 1 1 0 1
    \[ -2^3 + 2^2 + 0 + 2^0 \rightarrow -3 \]
    \[ -8 + 4 + 1 = -3 \]
  - Eight bits: 1 1 1 1 1 0 1
    \[ -2^7 + 2^6 + 2^5 + 2^4 + 0 + 2^0 \rightarrow -3 \]
    \[ -128 + 64 + 32 + 16 + 8 + 4 + 1 = -3 \]

Shifting

- Shifting corresponds to multiply and divide by powers of 2
  - Left arithmetic shift
    \[ z = 2x \text{ or } z = 2^x x \text{ for an n-bit shift} \]
    \[ \text{Shift in } 0\text{'s in LSB, OVF if } X_{n-1} \neq X'_{n-2} \]
  - Right arithmetic shift
    \[ \text{Divide by 2 (integer result) (1-bit shift)} \]
    \[ z = 2^{-1} x - \varepsilon \text{, } |\varepsilon| < 1 \]
    \[ \text{Remember to copy the sign bit in empty MSB!} \]
Multiplication (unsigned)

- **Pencil and paper method**
  \[ p = x \times y = \sum_{i=0}^{n-1} x \cdot r^i \cdot Y_i \]
  - Compute \( n \) terms of \( x r^i Y_i \) and then sum them
  - The \( i \)th term requires an \( i \)-position shift, and a multiplication of \( x \) by the single digit \( Y_i \)
  - Requires \( n-1 \) adders

Instead of using \( n-1 \) adders, can iterate with 1
- \( p[0] = 0 \)
- \( p[j+1] = r^{-1} (p[j] + x \cdot r^i Y_i) \) for \( j = 0, 1, \ldots, n-1 \)
- \( p = p[n] \)
- Takes \( n \) steps for \( n \) bits
Multiplication (signed!)

- Remember that MSB has negative weight
- Add partial products as normal
- Subtract multiplicand in last step...

```
-5  1011
  × -5  1101
       00000
       11011
       00000
       1110111
       0110111
       0000111
```

Division (unsigned)

- x=qd+w (quotient, divisor, remainder)
- Consider 0<d, x<r (precludes /0 and OVF)
- Basic division is n iterations of the recurrence
  \[ w[0] = x \]
  \[ w[j+1] = rw[j] - dq_{j-1} \quad j = 0, \ldots, n - 1 \]
- where \( q = \sum q_i r^i \) and \( d^* = dr^* \)
- i.e. divisor is aligned with most-significant half of residual

```
Division (unsigned)

In each step of the iteration
- Get one digit of quotient \( q_{j+1} = \text{SEL}(w[j], d) \)
- Value of digit is bounded such that
  \[ 0 \leq w[j+1] < d^* \]
- This means you find the right digit such that the current remainder is less than (shifted) divisor
- In binary you only have to guess 1 or 0
- Guess 1 and fix if you're wrong (restoring)

```
Long Division

- Dividend
- Divisor
- Quotient
- Remainder

```
Restoring Division

1. Shift current result one bit left
2. Subtract divisor from this result
3. If the result of step 2 is neg, \( q=0 \), else \( q=1 \)
4. If the result of step 2 is neg, restore old value of result by adding divisor back
5. Repeat n times...

This is what the recurrence in the book says...

```
Restoring Division

Algorithm RD: Restoring Divide

1. \( w[0] = x \)
2. \( \text{for } j = 0 \ldots n - 1 \) \( \text{do} \)
   1. \( \text{if } w[j+1] \leq 0 \) then
      1. \( q_{j+1} = 0, u = w[j+1] \odot d^* \)
      2. \( \text{else} \)
         1. \( q_{j+1} = 1, u = w[j+1] - d'^* \)
   \( \text{end for} \)
```

In an implementation the tentative and the true residuals are stored in the same register. Thus a "restoration" operation \( \overline{w} + d^* \) is performed whenever the tentative residual is negative.
Restoring Division

Example:

Consider what happens:
- Result at each step is 2r - d (r is current result)
- If the result is negative, we restore by adding d back in
- But, if you store the result in a separate place and don’t update the result until you know if it’s negative, then you can save some restoring steps

Non-restoring Division

Consider again:
- At each step 2residual - d
- If it’s negative, restore to 2r by adding d back in
- Then shift to get 4r, then subtract getting 4r - d
- Suppose you don’t restore, but continued with the shift resulting in 4r - 2d
- Now add d instead of subtract resulting in 4r - d
- That’s what you wanted!
Whew!

- Basic number representation systems
  - Unsigned, signed
  - Conversions
- Basic addition, subtraction of signed numbers
- Multiplication of unsigned and signed
- Division of signed

- Now let's speed up the operations!