Computer Arithmetic

—

Number Representation

Paolo.Ienne@epfl.ch

EPFL – I&C – LAP

Vojin.Oklobdzija@epfl.ch

EPFL – CSDA and UC Davis – ACSEL

Why Representation?

- Need to map numbers to binary bits
  - ➔ Need a representation anyway…
- Representation is consequential
  - ➔ Complexity of arithmetic operations depends heavily on the representation

Number representation is the heart of computer arithmetic…

What Is a Number System?

- A number is represented as an ordered n-tuple of symbols *(digit vector)*

$$\langle a_{n-1}a_{n-2}\ldots a_2a_1a_0\rangle$$

- Each symbol is a *digit*

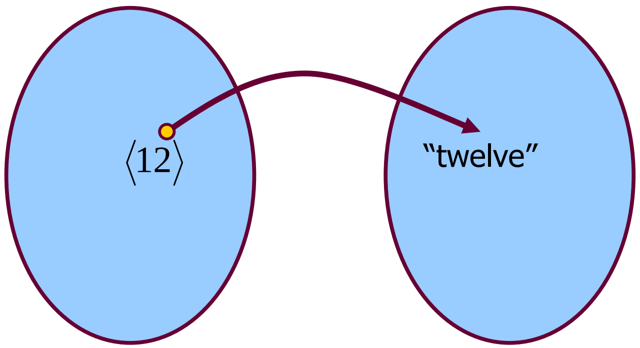
$$a_{n-1}, a_{n-2}, \ldots a_2, a_1, a_0$$

- Digits usually represent integers from a given set—e.g.,

$$a_i \in \{0, 1\} \text{ or } a_i \in \{0, 1, \ldots, 9\} \text{ or } a_i \in \{-1, 0, 1\}$$

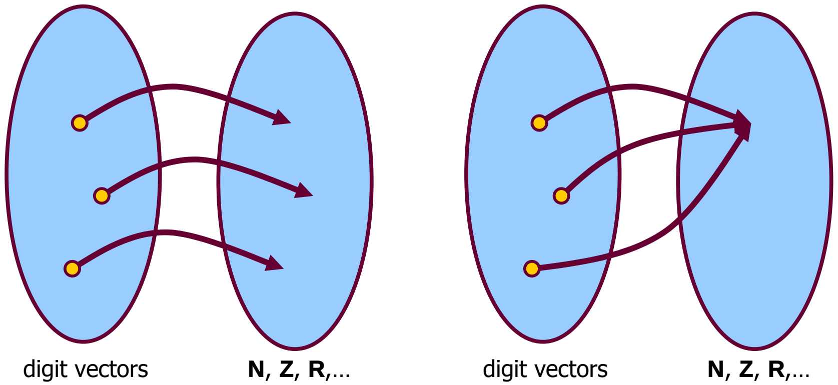
Rule of Interpretation

- Mapping from set of digit vectors to numbers (e.g., integers, reals)



digit vectors **N, Z, R**,…

Redundancy



Nonredundant　　　　Redundant

Positional Weighted Systems

- The rule of interpretation is a scalar product

$$A = \langle a_{n-1} a_{n-2} \ldots a_2 a_1 a_0 \rangle = \sum_{i=0}^{n-1} a_i w_i$$

where

$$W = (w_{n-1}, w_{n-2}, \ldots w_2, w_1, w_0)$$

is the *weight vector*

Radix Systems

- Weights are not arbitrary but related to a radix vector

$$R = (r_{n-1}, r_{n-2}, \ldots r_2, r_1, r_0)$$

in the following way

$$w_0 = 1 \qquad w_i = w_{i-1} \cdot r_{i-1} = \prod_{j=0}^{i-1} r_j$$

Mixed-Radix Systems

- *Fixed-radix* if all elements of *R* are identical

$$w_i = r^i \Rightarrow A = \sum_{i=0}^{n-1} a_i r^i$$

- A few *mixed-radix* systems are very common—e.g., *time*

$$R = (24, 60, 60)$$
$$W = (86400, 3600, 60, 1)$$

Common Decimal Notation

- It is *weighted*

$$A = \langle a_{n-1}a_{n-2}\ldots a_2 a_1 a_0 \rangle = \sum_{i=0}^{n-1} a_i w_i$$

- It is *positional*

w_i depends only on i

- It is *fixed-radix*

$$w_i = r^i$$

Common Decimal Notation

- It is *nonredundant* because *canonical*

$$a_i \in \{0, 1, \ldots, r-1\}$$

- 10^n possible digit vectors to represent 10^n values

Weighted, positional, fixed-radix, nonredundant
➔ also called *conventional* systems

Digit Set

- *Canonical* set

$$a_i \in \{0, 1, \ldots, r-1\}$$

A canonical system is nonredundant

- Any other choice is noncanonical

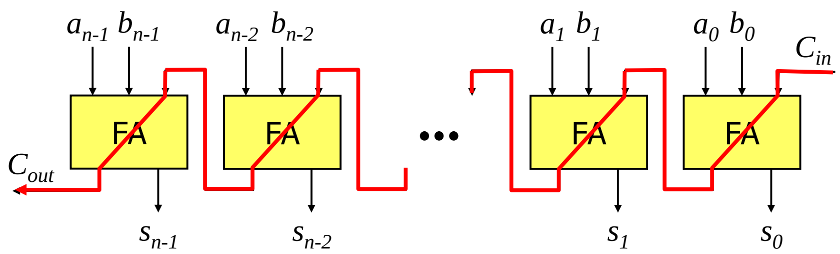
$$a_i \in \{-1, 0, 1\}$$

$$a_i \in \{-1, 0, 1, 2\}$$

$$a_i \in \{-(r/2-1), \ldots, 0, \ldots, r/2-1, r/2\}$$

Why Redundant Representations?

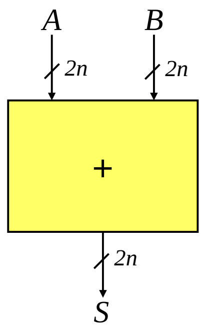
- Remember paper & pencil addition?

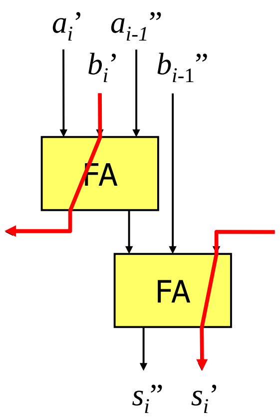


- In binary:
  - Slow—critical path contains *n* full-adders: naïve delay is *O(n)*
  - Must be performed right to left (LSB to MSB)

Redundancy May Simplify Some Operations

- Radix-2, noncanonical $\{0, 1, 2, 3\}$, 2 bits per digit





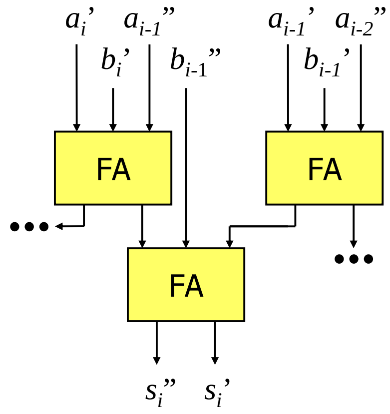
digit	encoding	
a_i	a_i"	a_i'
0	0	0
1	0	1
2	1	0
3	1	1

Critical path now contains only two full-adders: delay is now ***O(1)***

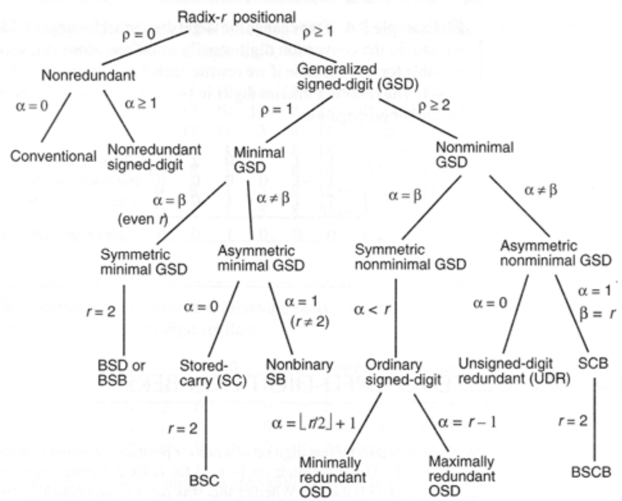
Redundancy and MSB-first?

- Now that the carry chain is broken, output digit i depends exclusively on input digits i, $i-1$, $i-2$
- Hence I can receive the operands in serial form and, when I have seen the first three input digits, I can produce the first output digit; afterwards, every new input digit, I can produce a new output digit

Online arithmetic or MSDF digit-serial arithmetic



Very Large Choice of Weighted Positional Number Systems



Source: Parhami, © Oxford 2000

Binary Representation and Negative Numbers

- Simplest signed system: **Sign-and-magnitude**
  - One bit represents the sign (typ. 1 → neg.)
  - Absolute value represented as unsigned

$$A = \langle s a_{n-2} \ldots a_2 a_1 a_0 \rangle = (-1)^s \cdot \sum_{i=0}^{n-2} a_i 2^i$$

  - Exactly as humans do in decimal

$$A = \langle \pm a_{n-2} \ldots a_2 a_1 a_0 \rangle = \pm \sum_{i=0}^{n-2} a_i 10^i$$

Of course, we could also write 0 for + and 9 for – …

Sign-and-Magnitude Representation

- Some advantages and disadvantages
  - ↗ Familiar for users
  - ↗ Simple naïve multiplication
  - ↘ Adders are not the most efficient
  - ↘ Redundant zero (+0 and –0) may cause some problems (e.g. testing for a variable =0)

Negative Numbers As Result of Paper & Pencil Subtraction

- Consider a normal paper-and-pencil subtraction

$$
\begin{array}{ccccccccccc}
 & & -1 & -1 & -1 & & & & -1 & & & \\
 & & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0}_2 & \rightarrow & 10_{10} \\
 & - & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & \rightarrow & 17_{10} \\
\hline
\dots & \dots & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & & & \\
 & & & & & \Downarrow & & & & & & \\
 & & \mathbf{-1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & & \\
 & & -2^7 & +2^6 & +2^5 & +2^4 & +2^3 & & & +2^0 & \rightarrow & -7_{10}
\end{array}
$$

Intuitive Derivation of the Two's Complement System

- Two's complement representation

$$A = \langle a_{n-1} a_{n-2} \dots a_2 a_1 a_0 \rangle = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Largest (positive): all ones except $a_{n-1} = 0$

$$A_{\max} = 2^{n-1} - 1$$

- Smallest (negative): all zeros except $a_{n-1} = 1$

$$A_{\min} = -2^{n-1}$$

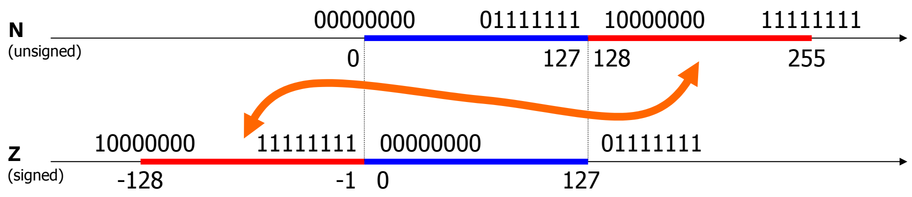
- Asymmetric range, non-redundant
- Standard in computers for signed integers nowadays

Comparison of Binary Representations on 3 Bits

Decimal	Binary (Unsigned)	Sign and Magnitude	Two's Complement
7	111		
6	110		
5	101		
4	100		
3	011	011	011
2	010	010	010
1	001	001	001
0	000	000 / 100	000
-1		101	111
-2		110	110
-3		111	101
-4			100

Two's Complement Representation of Negative Numbers

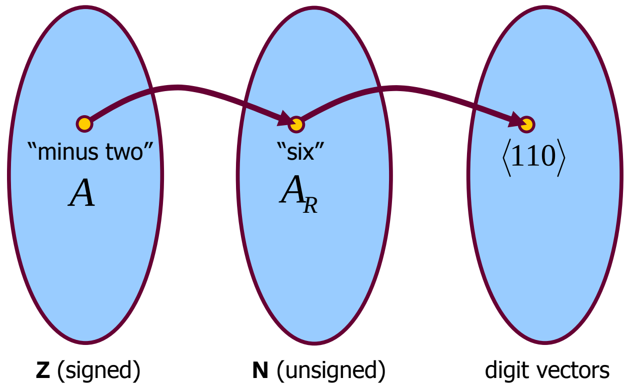
- It "shifts" the second half of the unsigned range into the negative by 2^n
- E.g., on 8 bits:



that is, $-A$ is represented as the unsigned integer $2^n - A$

True-and-Complement Systems for Signed Numbers

- Transform signed numbers in unsigned and then use conventional systems



Mapping from Signed to Unsigned

- A number A is represented through the unsigned number A_R defined through a *complementation constant*

$$A_R = A \bmod C$$

- This is equivalent (for $|A| < C$) to

$$A_R = \begin{cases} A & \text{if } A \geq 0 \\ C - |A| = C + A & \text{if } A < 0 \end{cases}$$

True forms

Complement forms

- For the representation to be unambiguous, it must be

$$|A| < C/2$$

or at most equal…

Inverse Mapping

- Inversely

$$A = \begin{cases} A_R & \text{if } A_R < C/2 \\ A_R - C & \text{if } A_R > C/2 \end{cases}$$

- If $A_R = C/2$ can be represented, it can be assigned either to $A = -C/2$ or to $A = C/2$, but the representation is then asymmetric (→ system not close under sign change operation)
- If $A_R = C$ is can be represented, there are two representations for zero

Range Complement and Digit Complement Systems

- Two particular cases of true-and-complement systems
  - Range complement: $C = r^n$
  - Digit complement: $C = r^n - 1$
- In radix-2
  - Two's complement: $C = 2^n$
  - One's complement: $C = 2^n - 1$

Two's complement is asymmetric and nonredundant
One's complement is symmetric with a redundant zero

Comparison of More Binary Representations on 3 Bits

Decimal	Binary (Unsigned)	Sign and Magnitude	Two's Complement	One's Complement
7	111			
6	110			
5	101			
4	100			
3	011	011	011	011
2	010	010	010	010
1	001	001	001	001
0	000	000 / 100	000	000 / 111
-1		101	111	110
-2		110	110	101
-3		111	101	100
-4			100	

Equivalence of the Two Two's Complement Definitions

- Using the true-and-complement definition, if $A_R < C/2$

$$\begin{cases} A = A_R \\ a_{n-1} = 0 \end{cases} \Rightarrow \quad A = \sum_{i=0}^{n-2} a_i 2^i$$

- If $A_R \geq C/2$

$$\begin{cases} A = A_R - C \\ a_{n-1} = 1 \end{cases} \Rightarrow \quad A = -2^n + \sum_{i=0}^{n-1} a_i 2^i = -2^n + 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i =$$

$$= -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Hence the property

$$A = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Similar Property for One's Complement

- Using the true-and-complement definition, if $A_R < C/2$

$$\begin{cases} A = A_R \\ a_{n-1} = 0 \end{cases} \Rightarrow \quad A = \sum_{i=0}^{n-2} a_i 2^i$$

- If $A_R \geq C/2$

$$\begin{cases} A = A_R - C \\ a_{n-1} = 1 \end{cases} \Rightarrow \quad A = -(2^n - 1) + \sum_{i=0}^{n-1} a_i 2^i = -2^n + 2^{n-1} + 1 + \sum_{i=0}^{n-2} a_i 2^i =$$

$$= -2^{n-1} + 1 + \sum_{i=0}^{n-2} a_i 2^i$$

- Hence the property

$$A = -a_{n-1}\left(2^{n-1} + 1\right) + \sum_{i=0}^{n-2} a_i 2^i$$

Sign

$$\operatorname{sign}(A) = \begin{cases} 0 & \text{if } A \geq 0 \\ 1 & \text{if } A \leq 0 \end{cases}$$

- For sign-and-magnitude is clearly trivial…
- For true-and-complement too, it is

$$\operatorname{sign}(A) = \begin{cases} 0 & \text{if } A_R < C/2, \text{ that is if } a_{n-1} = 0 \\ 1 & \text{if } A_R \geq C/2, \text{ that is if } a_{n-1} = 1 \end{cases}$$

- For all these representations

Sign bit

$$\boxed{\operatorname{sign}(A) = a_{n-1}}$$

Implied Digits in the Unsigned Representation

- Unsigned numbers (like naturals in decimal) have infinite leading zeros in front of them:

$$1\ 234_{10} \quad \rightarrow \quad \ldots 00\ 001\ 234_{10}$$

$$1\ 0101_2 \quad \rightarrow \quad \ldots 0\ 0000\ 0001\ 0101_2$$

- Changing the representation from n bits to m bits ($m > n$) is just a matter of padding the number with leading zeros

Changing the Number of Bits in One's and Two's Complement

- Signed numbers can be thought as having infinite replicas of the MSB/sign in front of them

				1	**1**	**0**	**1_2**		4 bits
				-2^3	$+2^2$		$+2^0$	$\rightarrow$	-3_{10}
1	**1**	**1**	**1**	**1**	**1**	**0**	**1_2**		8 bits
-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$		$+2^0$	$\rightarrow$	-3_{10}

- "Minus three" on 4 bits is **not** represented as "minus three" on 8 bits!

Sign Extension and Shift Right

- Two's complement representation on n bits

$$A^{(n)} = \langle a_{n-1} a_{n-2} \ldots a_2 a_1 a_0 \rangle = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Sign extended two's complement representation on m bits ($m > n$)

$$A^{(m)} = \langle a_{n-1} \ldots a_{n-1} a_{n-1} a_{n-2} \ldots a_2 a_1 a_0 \rangle =$$

$$= -a_{n-1} 2^{m-1} + \sum_{j=n-1}^{m-2} a_{n-1} 2^j + \sum_{i=0}^{n-2} a_i 2^i$$

- To extend from n to m bits, one adds (for positive #) zero or (for negative #)

$$\left(-2^{m-1} + 2^{m-2} + \ldots + 2^{n-1}\right) - \left(-2^{n-1}\right) \equiv 0$$

Addition in True-and-Complement

- If there is no overflow (that is, we assume the result of the sum to be representable), $S = A + B$ can be simply computed as

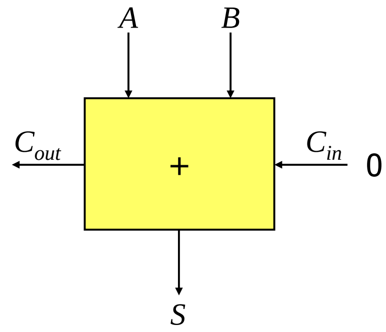
$$S_R = (A_R + B_R) \bmod C$$

- Easy to prove using $(X \bmod Y) \bmod Y = X \bmod Y$

$$(A_R + B_R) \bmod C = (A \bmod C + B \bmod C) \bmod C =$$
$$= (A + B) \bmod C = S \bmod C = S_R$$

Two's Complement Addition

- Particularly comfortable in two's complement, because $C = 2^n$ and performing $\bmod 2^n$ implies just to **ignore the carry out bit**



- Remember the intuitive derivation of the two's complement representation…

One's Complement Addition

- A little more complex in one's complement, because $C = r^n - 1$ and the modulo of $W_R = A_R + B_R$ is less easy

$w_n = 0$ and $W_R \bmod (r^n - 1) = W_R$ if $A_R + B_R < r^n - 1$

$w_n = 0$ and $W_R \bmod (r^n - 1) = 0$ if $A_R + B_R = r^n - 1$

$w_n = 1$ and $W_R \bmod (r^n - 1) = W_R - r^n + 1$ if $r^n - 1 < A_R + B_R \le 2(r^n - 1)$

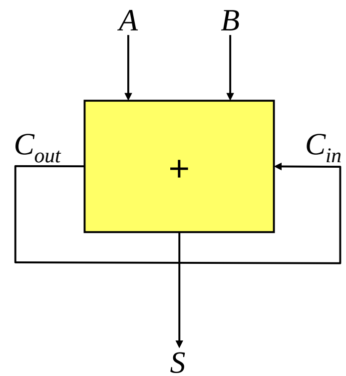
- Hence

If $w_n = 0 \Rightarrow S_R = W_R$

If $w_n = 1 \Rightarrow S_R = W_R + 1$, subtracting $r^n = 2^n$ simply by ignoring w_n

One's Complement Addition

- End-around carry



Negation

- Bitwise negation

$$\overline{A} = \text{not}(A) = \langle \overline{a_{n-1}}\ \overline{a_{n-2}} \ldots \overline{a_2}\ \overline{a_1}\ \overline{a_0} \rangle$$

where, as usual in binary,

$$\overline{a} = (r-1) - a$$

- Example in radix-2 on 4 bits:

$$\overline{6_{10}} = \text{not}(6_{10}) = \langle \overline{0}\,\overline{1}\,\overline{1}\,\overline{0} \rangle = \langle 1001 \rangle = 1001_2 = 9_{10}$$

Complement and Negation in True-and-Complement

- Fundamental property in n-digit r-radix systems

$$A + \overline{A} + 1 = r^n$$

because

	a_{n-1}	a_{n-2}	…	a_2	a_1	a_0	+	A
	$\overline{a_{n-1}}$	$\overline{a_{n-2}}$	…	$\overline{a_2}$	$\overline{a_1}$	$\overline{a_0}$	=	$\overline{A}$
	$r-1$	$r-1$	…	$r-1$	$r-1$	$r-1$	+	
						1	=	1
1	0	0	…	0	0	0		r^n

Change of Sign

- In one's complement

$$-A = \overline{A}$$

$$\begin{cases} -A = r^n - 1 - A = \overline{A} & \text{if } A \ge 0 \\ \overline{A} = \overline{r^n - 1 + A} = r^n - (r^n - 1 + A) - 1 = -A & \text{if } A < 0 \end{cases}$$

- In two's complement

$$-A = \overline{A} + 1$$

$$\begin{cases} -A = r^n - A = \overline{A} + 1 & \text{if } A \ge 0 \\ \overline{A} + 1 = \overline{r^n + A} + 1 = r^n - (r^n + A) - 1 + 1 = -A & \text{if } A < 0 \end{cases}$$

Change of Sign in Two's Complement

- Another way of verifying the property

$$A + \overline{A} = -1$$

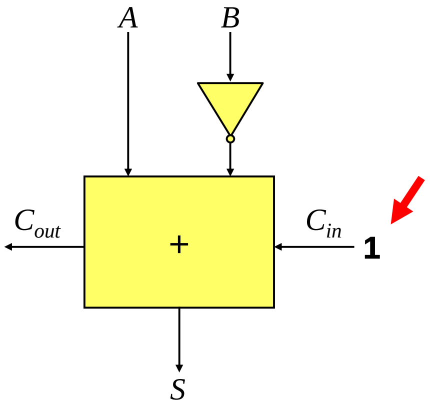
- Use two's complement property/definition

$$\left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) + \left(-\overline{a_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} 2^i\right) =$$

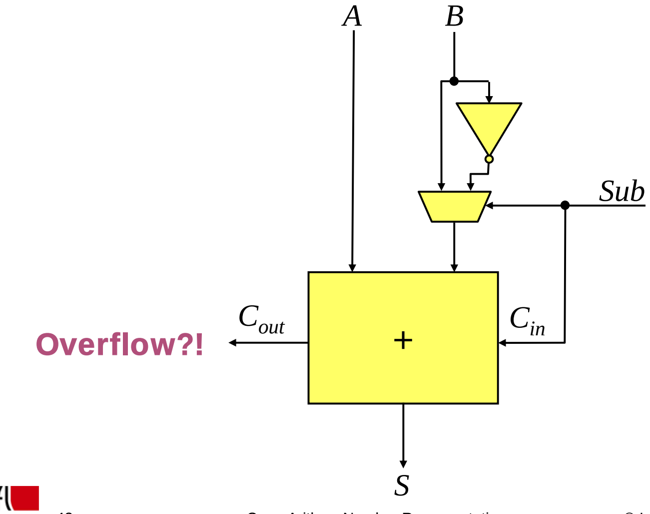
$$= -\left(a_{n-1} + \overline{a_{n-1}}\right) \cdot 2^{n-1} + \sum_{i=0}^{n-2} \left(a_i + \overline{a_i}\right) \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1$$

Two's Complement Subtracter

$$A - B = A + (-B) = A + \overline{B} + 1$$



Two's Complement Adder/Subtracter



Overflows for Unsigned Addition

- Limited range, thus sum of two numbers can exceed the range

	1	1	0	0	1	0	0	1_2	→	201_{10}
+	1	1	0	0	1	0	0	0_2	→	200_{10}
1	1	0	0	1	0	0	0	1_2	→	401_{10} ($>255_{10}$)

- **Overflows if carry out is 1**—that is, if the first bit which cannot be stored is not null

Overflows for Signed Addition

- The situation is somehow different and the same simple rule does not hold anymore

	1	1	0	0	1	0	0	1_2	→	-55_{10}
+	1	1	0	0	1	0	0	0_2	→	-56_{10}
1	1	0	0	1	0	0	0	1_2	→	-111_{10}
	1	0	0	1	0	0	0	1_2	→	-111_{10}

- **Carry out is 1**, but now it is ok because the result is negative (think of sign extension…)

Overflows for Signed Addition

- Overflows makes sum of positives look like negative or sum of negatives look like positive

$90_{10} = \underline{0}101\ 1010_2$

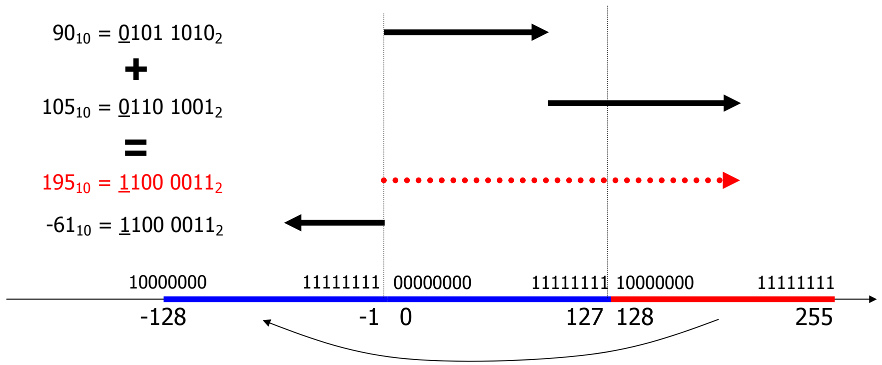
+

$105_{10} = \underline{0}110\ 1001_2$

=

$195_{10} = \underline{1}100\ 0011_2$

$-61_{10} = \underline{1}100\ 0011_2$



Overflows
Addition and Subtraction

- No overflow possible if adding operands of different sign
- Overflow only if

Op	A	B	Result
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

- Hence

$$OVF_+ = a_{n-1}\, b_{n-1}\, \overline{s_{n-1}} + \overline{a_{n-1}}\, \overline{b_{n-1}}\, s_{n-1}$$
$$OVF_- = a_{n-1}\, \overline{b_{n-1}}\, \overline{s_{n-1}} + \overline{a_{n-1}}\, b_{n-1}\, s_{n-1}$$

Overflows Addition

- Considering all possible cases around the last FA

a_{n-1}	b_{n-1}	c_{n-1}	c_n	s_{n-1}	Overflow
0	0	0	0	0	No
0	0	1	0	1	Yes
0	1	0	0	1	No
0	1	1	1	0	No
1	0	0	0	1	No
1	0	1	1	0	No
1	1	0	1	0	Yes
1	1	1	1	1	No

$$OVF_+ = c_n \oplus c_{n-1}$$

Another View on Overflow for Signed Addition

- Sign extend the operands and check if the additional result bit is equal to the pervious one (sign “unextension”…)

$$\begin{array}{rccccccccccl}
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & \rightarrow & -55_{10} \\
+ & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0}_2 & \rightarrow & -56_{10} \\
\hline
 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & \rightarrow & -111_{10} \\
 & & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1}_2 & \rightarrow & -111_{10}
\end{array}$$

$$OVF_+ = (a_n \oplus b_n \oplus c_n) \oplus (a_{n-1} \oplus b_{n-1} \oplus c_{n-1}) =$$
$$= (a_{n-1} \oplus b_{n-1} \oplus c_n) \oplus (a_{n-1} \oplus b_{n-1} \oplus c_{n-1}) =$$
$$= c_n \oplus c_{n-1}$$

Fixed Point

- In real life, one does not only need integers…
- If one adds a fractional point in a fixed position, hardware for integers works just as well

$$
\begin{array}{rccccccl}
 & 0 & 1 & 0 & 0 & 1_2 & \rightarrow & 9_{10} \\
+ & 0 & 0 & 0 & 1 & 1_2 & \rightarrow & 3_{10} \\
\hline
 & 0 & 1 & 1 & 0 & 0_2 & \rightarrow & 12_{10} \\
 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & &
\end{array}
\qquad
\begin{array}{rccccccl}
 & 0 & 1. & 0 & 0 & 1_2 & \rightarrow & 1.125_{10} \\
+ & 0 & 0. & 0 & 1 & 1_2 & \rightarrow & 0.375_{10} \\
\hline
 & 0 & 1. & 1 & 0 & 0_2 & \rightarrow & 1.500_{10} \\
 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & &
\end{array}
$$

- It's just a matter of representation!

Fixed Point Requires Attention from Programmers

- Multiplication often introduces the need of arithmetic shifts to the right

$$
\begin{array}{rccccccccccl}
 & & & & & 0. & 1 & 0 & 1 & 0_2 & \rightarrow & 0.625_{10} \\
\times & & & & & 0. & 0 & 1 & 1 & 0_2 & \rightarrow & 0.375_{10} \\
\hline
 & 0. & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0_2 & \rightarrow & 0.234375_{10} \\
 & & & & & & & & & & & \\
 & \rightarrow & \rightarrow & \rightarrow & \rightarrow & 0. & 0 & 0 & 1 & 1_2 & \rightarrow & 0.1875_{10}
\end{array}
$$

- Note the low accuracy!

References

- M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2004
- I. Koren, *Computer Arithmetic Algorithms*, Peters, 2002
- A. R. Omondi, *Computer Arithmetic Systems—Algorithms, Architecture and Implementation*, Prentice Hall, 1991
- B. Parhami, *Computer Arithmetic—Algorithms and Hardware Designs*, Oxford, 2000