ETH *Eidgenössische*
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme   Integrated Systems Laboratory

Lecture notes on

Computer Arithmetic: Principles, Architectures, and VLSI Design

March 16, 1999

Reto Zimmermann

Integrated Systems Laboratory
Swiss Federal Institute of Technology (ETH)
CH-8092 Zürich, Switzerland
zimmermann@iis.ee.ethz.ch

`http://www.iis.ee.ethz.ch/ zimmi/publications/comp_arith_notes.ps.gz`

Contents

1 Introduction and Conventions

1.1 Outline

- Basic *principles* of computer arithmetic [1, 2, 3, 4, 5, 6, 7]

- *Circuit architectures* and *implementations* of main arithmetic operations

- Aspects regarding *VLSI design* of arithmetic units

1.2 Motivation

- Arithmetic units are, among others, core of every *data path* and *addressing unit*

- Data path is core of :

  ○ *microprocessors* (CPU)

  ○ *signal processors* (DSP)

  ○ data-processing *application specific ICs* (ASIC) and *programmable ICs* (e.g. FPGA)

- Standard arithmetic units available from *libraries*

- *Design* of arithmetic units necessary for :

  ○ non-standard operations

  ○ high-performance components

  ○ library development

1.3 Conventions

Naming conventions

Signal buses : A (1-D), A_i (2-D), $a_{i:k}$ (subbus, 1-D)

Signals : a, a_i (1-D), $a_{i,k}$ (2-D), $A_{i:k}$ (group signal)

Circuit complexity measures : A (area), T (cycle time, delay), AT (area-time product), L (latency, # cycles)

Arithmetic operators : $+, -, \cdot, /, \log$ ($= \log_2$)

Logic operators : $+$ (or), $\cdot$ (and), $\oplus$ (xor), $\odot$ (xnor), $^-$ (not)

Circuit complexity measures

Unit-gate model ($\sim$ gate-equivalents (GE) model) :

- *Inverter, buffer* : $\boxed{A = 0 \,,\; T = 0}$ (i.e. ignored)

- *Simple monotonic 2-input gates* (AND, NAND, OR, NOR) : $\boxed{A = 1 \,,\; T = 1}$

- *Simple non-monotonic 2-input gates* (XOR, XNOR) : $\boxed{A = 2 \,,\; T = 2}$

- *Complex gates* : composed from simple gates

$\Rightarrow$ *Simple m-input gates* : $\boxed{A = m - 1 \,,\; T = \lceil \log m \rceil}$

- *Wiring* not considered (acceptable for comparison purposes, local wiring, multilevel metallization)

- Only *estimations* given for complex circuits

1.4 Recursive Function Evaluation

- Given : inputs a_i, outputs z_i, function f (graph sym. : ●)

Non-recursive functions (n.)

- Output z_i is a function of input a_i (or $a_{j+m:j}$, m const.)

$$\boxed{z_i = f(a_i, x) \;;\; i = 0, \ldots, n - 1}$$

$\Rightarrow$ *parallel* structure :

$$\boxed{A = O(n) \,,\; T = O(1)}$$
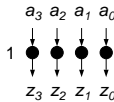
Recursive functions (r.)

- Output z_i is a function of all inputs a_k , $k \leq i$

a) with *single* output $z = z_{n-1}$ (r.s.) :

$$\boxed{\begin{array}{l} t_i = f(a_i, t_{i-1}) \;;\; i = 0, \ldots, n - 1 \\ t_{-1} = 0/1 \,,\; z = t_{n-1} \end{array}}$$

1. f is *non-associative* (**r.s.n.**)
   $\Rightarrow$ *serial* structure :
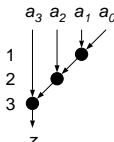
$$\boxed{A = O(n) \,,\; T = O(n)}$$

2. f is *associative* (**r.s.a.**)
   $\Rightarrow$ serial or *single-tree* structure :

$$\boxed{A = O(n) \,,\; T = O(\log n)}$$
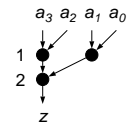
b) with *multiple* outputs z_i (r.m.) ($\Rightarrow$ prefix problem) :

$$\boxed{z_i = f(a_i, z_{i-1}) \;;\; i = 0, \ldots, n - 1 \,,\; z_{-1} = 0/1}$$

1. f is *non-associative* (**r.m.n.**)
   $\Rightarrow$ *serial* structure :

$$\boxed{A = O(n) \,,\; T = O(n)}$$
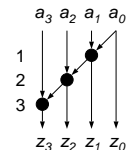
2. f is *associative* (**r.m.a.**)
   $\Rightarrow$ serial or *multi-tree* structure :

$$\boxed{A = O(n^2) \,,\; T = O(\log n)}$$
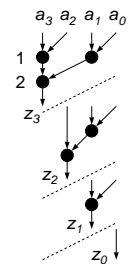
$\Rightarrow$ or *shared-tree* structure :

$$\boxed{A = O(n \log n) \,,\; T = O(\log n)}$$

2 Arithmetic Operations

2.1 Overview

⟶ based on operation
⋯⋯ related operation

fixed-point | floating-point

<< , >>

= , <   +1 , -1   +/-   + , -   + , -

·   ·

/   sqrt (x)

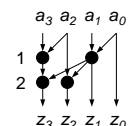(same as on the left for floating-point numbers)

exp (x)

log (x)   trig (x)   hyp (x)

complexity
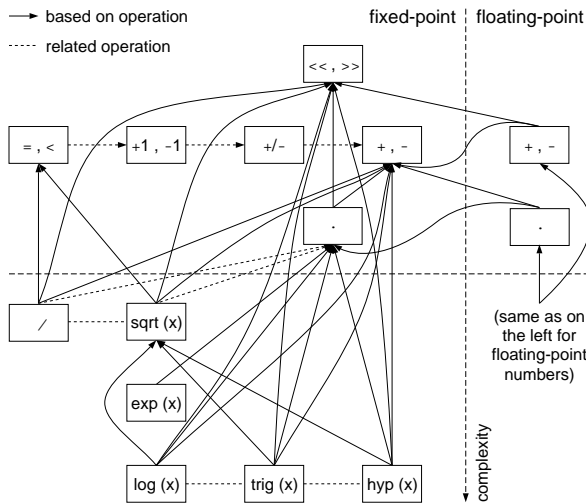
1	shift/extension	7	division
2	comparison	8	square root extraction
3	increment/decrement	9	exponential function
4	complement	10	logarithm function
5	addition/subtraction	11	trigonometric functions
6	multiplication	12	hyperbolic functions

2.2 Implementation Techniques

Direct implementation of dedicated units :

- *always* : $1 - 5$
- *in most cases* : 6
- *sometimes* : 7, 8

Sequential implementation using simpler units and several clock cycles ($\Rightarrow$ decomposition) :

- *sometimes* : 6
- *in most cases* : 7, 8, 9

Table look-up techniques using ROMs :

- *universal* : simple application to all operations
- *efficient* only for single-operand operations of high complexity ($8 - 12$) and small word length (note: ROM size $= 2^n \times n$)

Approximation techniques using simpler units : 7–12

- *taylor series* expansion
- *polynomial* and *rational* approximations
- convergence of *recursive equation systems*
- *CORDIC* (COordinate Rotation DIgital Computer)

---

3 Number Representations

3.1 Binary Number Systems (BNS)

- *Radix-2, binary* number system (BNS) : irredundant, weighted, positional, monotonic [1, 2]

- n-bit number is *ordered sequence* of **bits** (**b**inary dig**its**) :
  $$A = (a_{n-1}, a_{n-2}, \ldots, a_0)_2 \,, \; a_i \in \{0, 1\}$$

- Simple and efficient implementation in *digital circuits*

- *MSB/LSB* (most-/least-significant bit) : a_{n-1} / a_0

- Represents an *integer* or *fixed-point* number, **exact**

- *Fixed-point* numbers : $(\underbrace{a_{m-1}, \ldots, a_0}_{m\text{-bit integer}} \cdot \underbrace{a_{-1}, \ldots, a_{m-n}}_{(n-m)\text{-bit fraction}})$

Unsigned : *positive* or *natural* numbers

$Value$: $A = a_{n-1}2^{n-1} + \cdots + a_1 2 + a_0 = \sum_{i=0}^{n-1} a_i 2^i$

$Range$: $[0, \; 2^n - 1]$

Two's (2's) complement : standard representation of *signed* or *integer* numbers

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

$Range$: $[-2^{n-1}, \; 2^{n-1} - 1]$

$Complement$: $-A = 2^n - A = \overline{A} + 1$, where $\overline{A} = (\overline{a}_{n-1}, \overline{a}_{n-2}, \ldots, \overline{a}_0)$

$Sign$: a_{n-1}

$Properties$: asymmetric range, compatible with unsigned numbers in many arithmetic operations (i.e. same treatment of positive and negative numbers)

One's (1's) complement : similar to 2's complement

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

$Range$: $[-(2^{n-1} - 1), \; 2^{n-1} - 1]$

$Complement$: $-A = 2^n - A - 1 = \overline{A}$

$Sign$: a_{n-1}

$Properties$: double representation of zero, symmetric range, modulo $(2^n - 1)$ number system

Sign-magnitude : alternative representation of signed numbers

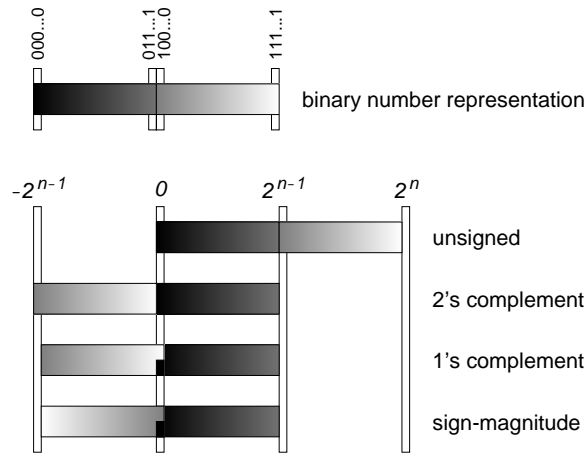$Value$: $A = (-1)^{a_{n-1}} \cdot \sum_{i=0}^{n-2} a_i 2^i$

$Range$: $[-(2^{n-1} - 1), \; 2^{n-1} - 1]$

$Complement$: $-A = (\overline{a}_{n-1}, a_{n-2}, \ldots, a_0)$

$Sign$: a_{n-1}

Properties : double representation of zero, symmetric range, different treatment of positive and negative numbers in arithmetic operations, no MSB toggles at sign changes around 0 ($\Rightarrow$ low power)

Graphical representation

Conventions

- 2's complement used for *signed numbers* in these notes
- *Unsigned* and *signed* numbers can be treated equally in most cases, exceptions are mentioned

---

3.2 Gray Numbers

- *Gray numbers* (*code*) : binary, irredundant, non-weighted, non-monotonic
- + *Property* : unit-distance coding (i.e. exactly one bit toggles between adjacent numbers)
- *Applications* : counters with *low output toggle rate* (low-power signal buses), representation of continuous signals for *low-error sampling* (no false numbers due to switching of different bits at different times)
- – *Non-monotonic* numbers : difficult arithmetic operations, e.g. addition, comparison :

$$\frac{g_1 g_0 \quad g_1' g_0' \quad\quad g_0 \quad g_0'}{0 \ \underline{0} < 0 \ \underline{1} \ \text{ and } \ 0 < 1}$$
$$1 \ \underline{1} < 1 \ \underline{0} \ \text{ but } \ 1 > 0$$

- *binary* $\rightarrow$ *Gray* :

$$\boxed{g_i = b_{i+1} \oplus b_i \ , b_n = 0 \ ; \\ i = 0, \dots, n-1 \quad (\text{n.})}$$

- *Gray* $\rightarrow$ *binary* :

$$\boxed{b_i = b_{i+1} \oplus g_i \ , b_n = 0 \ ; \\ i = n-1, \dots, 0 \quad (\text{r.m.a.})}$$

	binary $b_3\,b_2\,b_1\,b_0$	Gray $g_3\,g_2\,g_1\,g_0$
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0

3.3 Redundant Number Systems

- *Non-binary*, *redundant*, *weighted* number systems [1, 2]
- *Digit set* larger than radix (typically radix 2) $\Rightarrow$ *multiple representations* of same number $\Rightarrow$ *redundancy*
- + No *carry-propagation* in adders $\Rightarrow$ more efficient impl. of *adder-based* units (e.g. multipliers and dividers)
- – Redundancy $\Rightarrow$ no direct implementation of *relational* operators $\Rightarrow$ conversion to irredundant numbers
- – Several bits used to represent one digit $\Rightarrow$ higher *storage requirements*
- – Expensive *conversion* into irredundant numbers (not necessary if redundant input operands are allowed)

Delayed-carry of **half-adder** number representation :
- $r_i \in \{0, 1, 2\} \ , \ c_i, s_i, a_i, b_i \in \{0, 1\} \ ,$
  $r_i = (c_{i+1}, s_i) = 2c_{i+1} + s_i = a_i + b_i \ , \underline{c_{i+1}s_i = 0}$
- $R = \sum_{i=0}^{n-1} r_i 2^i = (C, S) = C + S = A + B$
- 1 digit holds sum of *2 bits* (no carry-out digit)
- example : $\underline{(00, 10)} = 00 + 10 = 01 + 01 = \underline{(10, 00)}$
- irredundant representation of -1 [8], since
  $c_{i+1}s_i = 0 \ \& \ C + S = -1 \ \rightarrow \ S = -1, C = 0$

Carry-save number representation :
- $r_i \in \{0, 1, 2, 3\} \ , \ c_i, s_i, a_i, b_i, d_i \in \{0, 1\} \ ,$
  $r_i = (c_{i+1}, s_i) = 2c_{i+1} + s_i = a_i + b_i + d_i = a_i + r_i'$
- $R = \sum_{i=0}^{n-1} r_i 2^i = (C, S) = C + S = A + R'$

- 1 digit holds sum of *3 bits* or *1 digit + 1 bit* (no carry-out digit, i.e. carry is **saved**)
- *standard* redundant number system for fast addition

Signed-digit (SD) or **redundant digit** (RD) number representation :

- $r_i, s_i, t_i \in \{-1, 0, 1\} \equiv \{\overline{1}, 0, 1\} \ , \ R = \sum_{i=0}^{n-1} r_i 2^i$
- *no* carry-propagation in $S = R + T$:
  - ○ $r_i + t_i = (c_{i+1}, u_i) = 2c_{i+1} + u_i \ , \ c_{i+1}, u_i \in \{\overline{1}, 0, 1\}$
  - ○ (c_{i+1}, u_i) is *redundant* (e.g. $0 + 1 = 01 = 1\overline{1}$)
  - ○ $\forall i \ \exists (c_i, u_i) \mid c_i + u_i = s_i \in \{\overline{1}, 0, 1\}$
- 1 digit holds sum of *2 digits* (no carry-out digit)
- *minimal SD* representation : minimal number of non-zero digits, $\boxed{\cdots 011\{1\}10\cdots \rightarrow \cdots 100\{0\}\overline{1}0\cdots}$
  - ○ *applications* : sequential multiplication (less cycles), filters with constant coefficients (less hardware)
  - ○ *example* :
  $$7 = (0111 \mid 1\overline{1}11 \mid 10\overline{1}1 \mid \overbrace{100\overline{1}}^{\text{minimal}} \mid 1\overline{1}111 \mid \cdots)$$
- *canonical SD* repres.: minimal SD + not two non-zero digits in sequence, $\boxed{\cdots 01\{1\}10\cdots \rightarrow \cdots 10\{0\}\overline{1}0\cdots}$
- $SD \rightarrow binary$: carry-propagation necessary ($\Rightarrow$ adder)
- *other applications* : high-speed multipliers [9]
- similar to *carry-save*, simple use for *signed* numbers

3.4 Residue Number Systems (RNS)

- *Non-binary*, *irredundant*, *non-weighted* number system [1]

+ *Carry-free* and *fast* additions and multiplications

– *Complex* and *slow* other arithmetic operations
  (e.g. comparison, sign and overflow detection) because
  digits are *not weighted*, conversion to weighted
  mixed-radix or binary system required

- Codes for *error detection* and *correction* [1]

- Possible *applications* (but hardly used) :

  ○ *digital filters* : fast additions and multiplications

  ○ *error detection* and *correction* for arithmetic operations
    in conventional and residue number systems

- *Base* is n-tuple of integers $(m_{n-1}, m_{n-2}, \ldots, m_0)$,
  residues (or *moduli*) m_i pairwise relatively prime

  ○ $A = (a_{n-1}, a_{n-2}, \ldots, a_0)_{m_{n-1}, m_{n-2}, \ldots, m_0}$,
    $a_i \in \{0, 1, \ldots, m_i - 1\}$

  ○ Range: $M = \prod\limits_{i=0}^{n-1} m_i$, anywhere in $\mathbb{Z}$

  ○ $a_i = A \bmod m_i = |A|_{m_i}$, $A = m_i \cdot q_i + a_i$

  ○ $|A|_M = \left| \sum\limits_{i=0}^{n-1} C_i a_i \right|_M$, $C_i = (\ldots, 0, \underbrace{1}_{i}, 0, \ldots)$

- *Arithmetic operations* : (each digit computed separately)

  ○ $z_i = |Z|_{m_i} = |f(A)|_{m_i} = \left| f(|A|_{m_i}) \right|_{m_i} = |f(a_i)|_{m_i}$

  ○ $|A + B|_{m_i} = \left| |A|_{m_i} + |B|_{m_i} \right|_{m_i} = |a_i + b_i|_{m_i}$

  ○ $|A \cdot B|_{m_i} = \left| |A|_{m_i} \cdot |B|_{m_i} \right|_{m_i} = |a_i \cdot b_i|_{m_i}$

  ○ $|-a_i|_{m_i} = |m_i - a_i|_{m_i}$

  ○ $\left| a_i^{-1} \right|_{m_i} = \left| a_i^{m_i-2} \right|_{m_i}$　　(Fermat's theorem)

- Best moduli m_i are 2^k and $(2^k - 1)$:

  ○ high *storage efficiency* with k bits

  ○ simple *modular addition* : 2^k : k-bit adder without c_{out},
    $2^k - 1$: k-bit adder with end-around carry ($c_{in} = c_{out}$)

- Example : $(m_1, m_0) = (3, 2)$, $M = 6$

A	$\cdots$	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	$\cdots$
a_1	$\cdots$	2	0	1	2	0	1	2	0	1	2	0	1	2	$\cdots$
a_0	$\cdots$	0	1	0	1	0	1	0	1	0	1	0	1	0	$\cdots$

possible range

$$|5|_6 = A = (a_1, a_0) = (|5|_3, |5|_2) = (2, 1)$$

$$|4 + 5|_6 = (1, 0) + (2, 1) =$$
$$= (|1 + 2|_3, |0 + 1|_2) = (0, 1) = |3|_6$$

$$|4 \cdot 5|_6 = (1, 0) \cdot (2, 1) =$$
$$= (|1 \cdot 2|_3, |0 \cdot 1|_2) = (2, 0) = |2|_6$$

3.5 Floating-Point Numbers

- Larger *range*, smaller *precision* than fixed-point
  representation, **inexact**, *real* numbers [1, 2]

- *Double-number* form $\Rightarrow$ *discontinuous* precision

S	*biased exponent E*	*unsigned norm. mantissa M*

- $F = (-1)^S \cdot M \cdot \beta^E = (-1)^S \cdot 1.M \cdot 2^{E-bias}$

- Basic *arithmetic operations* :

  ○ $A \cdot B = (-1)^{S_A \oplus S_B} \cdot \underline{M_A \cdot M_B} \cdot \beta^{\underline{E_A + E_B}}$

  ○ $A + B = \left((-1)^{S_A} \cdot M_A + \right.$
  $$\left. (-1)^{S_B} \cdot \left(M_B \gg (E_A - E_B) \right) \right) \cdot \beta^{E_A}$$

  ○ base on *fixed-point* add, multiply, and shift operations

  ○ *postnormalization* required ($1/\beta \le M < 1$)

- *Applications* :
  processors : *"real"* floating-point formats (e.g. IEEE
    standard), *large* range due to universal use
  ASICs : usually *simplified* floating-point formats with
    small exponents, smaller range, used for *range
    extension* of normal fixed-point numbers

- *IEEE floating-point format* :

precision	n	n_M	n_E	bias	range	precision
single	32	23	8	127	$3.8 \cdot 10^{38}$	10^{-7}
double	64	52	11	1023	$9 \cdot 10^{307}$	10^{-15}

3.6 Logarithmic Number System

- Alternative representation to floating-point (i.e. mantissa
  + integer exponent $\to$ only *fixed-point exponent*) [1]

- *Single-number* form $\Rightarrow$ *continuous* precision $\Rightarrow$ higher
  accuracy, more reliable

S	*biased fixed-point exponent E*

- $L = (-1)^S \cdot \beta^E = (-1)^S \cdot 2^{E-bias}$　　(*signed-logarithmic*)

- Basic *arithmetic operations* :

  ○ $(A < B) = (E_A < E_B)$　　(additionally consider sign)

  ○ $A + B$: by *approximation* or addition in *conventional*
    number system and *double conversion*

  ○ $A \cdot B = (-1)^{S_A \oplus S_B} \cdot \beta^{E_A + E_B}$

  ○ $A^y = (-1)^{S_A} \cdot \beta^{y \cdot E_A}$, $\sqrt[y]{A} = (-1)^{S_A} \cdot \beta^{E_A/y}$

+ *Simpler* multiplication/exponent., *more complex* addition

– Expensive *conversion* : (anti)logarithms (table look-up)

- *Applications* : real-time digital filters

3.7 Antitetrational Number System

- *Tetration* (t. $x = \underbrace{2^{2^{\cdot^{\cdot^2}}}}_{x\times}$) and *antitetration* (a.t. x) [10]

- Larger *range*, smaller *precision* than logarithmic repres.,
  otherwise analogous (i.e. $2^x \to$ t. x , $\log x \to$ a.t. x)

3.8 Composite Arithmetic

- Proposal for a *new standard* of number representations [10]

- Scheme for storage and display of *exact* (primary: *integer*, secondary: *rational*) and *inexact* (primary: *logarithmic*, secondary: *antitetrational*) numbers

- *Secondary* forms used for numbers not representable by *primary* ones ($\Rightarrow$ no over-/underflow handling necessary)

- *Choice* of number representation hidden from user, i.e. software/compiler selects format for highest accuracy

- Number representations :

	tag	value	
integer :	00	2's complement integer	
rational :	$01\pm$	slash	denominator \ numerator
logarithmic :	$10\pm$	log integer	log fraction
antitetrational :	$11\pm$	a.t. integer	a.t. fraction

- *Rational* numbers : slash position (i.e. size of numerator/ denominator) is *variable* and stored (floating slash)

- *Storage form sizes* : 32-bit (short), 64-bit (normal), 128-bit (long), 256-bit (extended)

- *Implementation* : mixed hardware/software solutions

- Hardware proposal : *long accumulator* (4096 bits) holds any floating-point number in fixed-point format $\Rightarrow$ higher accurary $\Rightarrow$ large hardware/software overhead

---

3.9 Round-Off Schemes

- Intermediate results with d *additional* lower bits ($\Rightarrow$ higher accuracy) : $A = (a_{n-1}, \ldots, a_0, a_{-1}, \ldots, a_{-d})$

- *Rounding* : keeping error ϵ small during final *word length reduction* : $R = (r_{n-1}, \ldots, r_0) = A - \epsilon$

- *Trade-off* : numerical accuracy vs. implementation cost

Truncation : $\boxed{R_{TRUNC} = (a_{n-1}, \ldots, a_0)}$

- $bias = -\frac{1}{2} + \frac{1}{2^{d+1}}$ (= average error ϵ)

Round-to-nearest (i.e. normal *rounding*) :

$$\boxed{R_{ROUND} = (a'_{n-1}, \ldots, a'_0) , \ A' = A + \frac{1}{2} = A + 0.1_2}$$

- $bias = \frac{1}{2^{d+1}}$ (nearly symmetric)

- "$+ 0.1_2$" can often be included in previous operation

Round-to-nearest-even/-odd :

$$\boxed{R_{ROUND-EVEN} = \begin{cases} R_{ROUND} & \text{if } (a'_{-1}, \ldots, a'_{-d}) \neq 0 \cdots 0 \\ (a'_{n-1}, \ldots, a'_1, 0) & \text{otherwise} \end{cases}}$$

- $bias = 0$ (symmetric)

- *mandatory* in IEEE floating-point standard

- 3 *guard bits* for rounding after floating-point operations : *guard* bit G (postnormalization), *round* bit R (round-to-nearest), *sticky* bit S (round-to-nearest-even)

---

4 Addition

4.1 Overview
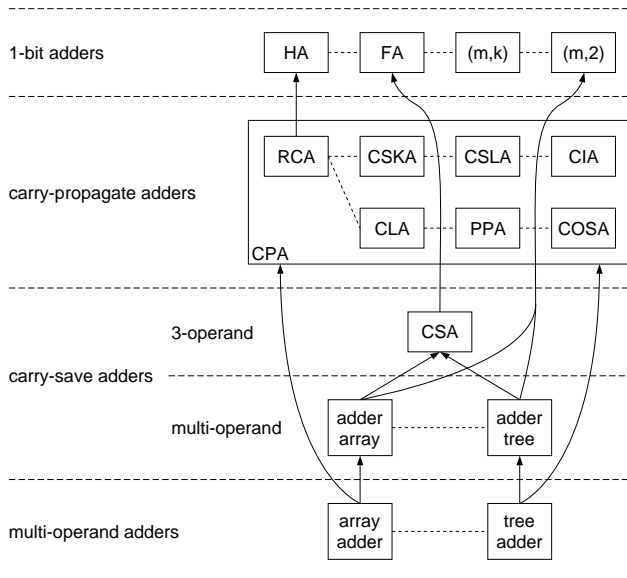
Legend:

HA: half-adder	CPA: carry-propagate adder	CLA: carry-lookahead adder
FA: full-adder	RCA: ripple-carry adder	PPA: parallel-prefix adder
(m,k): (m,k)-counter	CSKA: carry-skip adder	COSA: conditional-sum adder
(m,2): (m,2)-compressor	CSLA: carry-select adder	
	CIA: carry-increment adder	CSA: carry-save adder

→ based on component   ---- related component

4.2 1-Bit Adders, (m, k)-Counters

- *Add* up m bits of *same magnitude* (i.e. 1-bit numbers)

- Output *sum* as k-bit number ($k = \lfloor \log m \rfloor + 1$)

- or : **count** 1's at inputs $\Rightarrow$ *(m, k)-counter* [3] (*combinational* counters)

Half-adder (HA), (2, 2)-counter

$$\boxed{(c_{out}, s) = 2c_{out} + s = a + b} \qquad \boxed{A = 3 , \ T = 2 \ (1)}$$

$$\boxed{\begin{aligned} s &= a \oplus b \quad \text{(sum)} \\ c_{out} &= ab \quad \text{(carry-out)} \end{aligned}}$$

(reference)
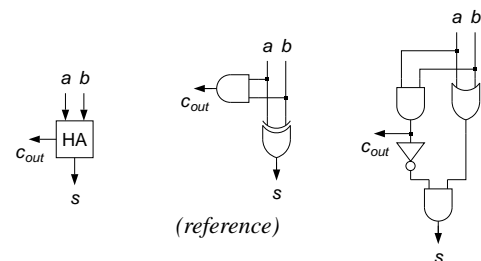
Full-adder (FA), (3, 2)-counter

$$\boxed{(c_{out}, s) = 2c_{out} + s = a + b + c_{in}} \qquad \boxed{A = 7 \, , \; T = 4 \; (2)}$$

$$\begin{array}{ll}
g = ab & \text{(generate)} \quad\big|\quad c^0 = ab \\
p = a \oplus b & \text{(propagate)} \quad\big|\quad c^1 = a + b \\
s = a \oplus b \oplus c_{in} = p \oplus c_{in} & \\
\end{array}$$
$$\begin{aligned}
c_{out} &= ab + ac_{in} + bc_{in} = ab + (a \oplus b)c_{in} \\
&= g + pc_{in} = \overline{p}g + pc_{in} = \overline{p}a + pc_{in} \\
&= \overline{c}_{in}c^0 + c_{in}c^1
\end{aligned}$$

(reference)

(m, k)-counters

$$\boxed{\begin{aligned}
(s_{k-1}, \ldots, s_0) &= \\
\sum_{j=0}^{k-1} s_j 2^j &= \sum_{i=0}^{m-1} a_i
\end{aligned}}$$
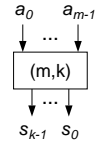
- Usually built from *full-adders*
- *Associativity* of addition allows conversion from *linear* to *tree structure* $\Rightarrow$ *faster* at same number of FAs

$$\boxed{\begin{aligned}
&A = 7 \sum_{k=1}^{\log m} \lfloor m2^{-k} \rfloor \approx 7(m - \log m) \, , \\
&T_{LIN} = 4m + 2\lfloor \log m \rfloor \, , \; T_{TREE} = 4\lceil \log_3 m \rceil + 2\lfloor \log m \rfloor
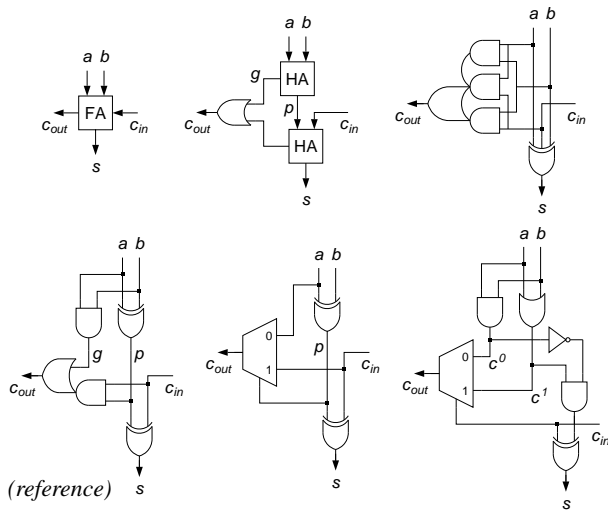\end{aligned}}$$

- Example : *(7, 3)-counter*

$$\boxed{A = 28 \, , \; T = 14} \qquad \boxed{A = 28 \, , \; T = 10}$$

linear structure   *tree structure*
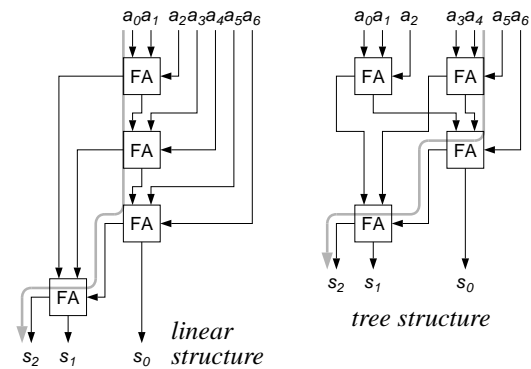
4.3 Carry-Propagate Adders (CPA)

- *Add* two n-bit operands A and B and an optional carry-in c_{in} by performing **carry-propagation** [1, 2, 11]
- *Sum* (c_{out}, S) is *irredundant* $(n + 1)$-bit number

$$\boxed{(c_{out}, S) = c_{out}2^n + S = A + B + c_{in}}$$

$$\boxed{\begin{aligned}
&2c_{i+1} + s_i = a_i + b_i + c_i \, ; \\
&\qquad i = 0, 1, \ldots, n - 1 \\
&c_0 = c_{in} \, , \; c_{out} = c_n \; \text{(r.m.a.)}
\end{aligned}}$$
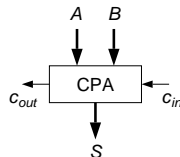
Ripple-carry adder (RCA)

- *Serial arrangement* of n full-adders
- *Simplest*, *smallest*, and *slowest* CPA structure

$$\boxed{A = 7n \, , \; T = 2n \, , \; AT = 14n^2}$$
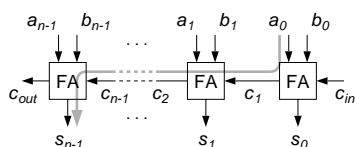
Carry-propagation speed-up techniques

a) Concatenation of *partial CPAs* with fast $c_{in} \rightarrow c_{out}$
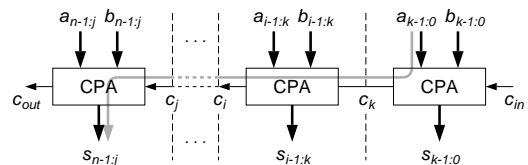
a) Fast *carry look-ahead* logic for entire range of bits

preprocessing

carry propagation

postprocessing
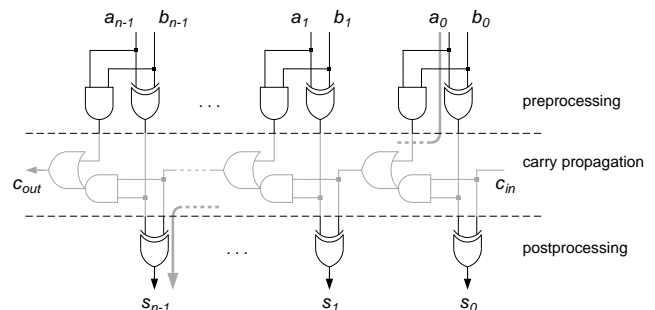
Carry-skip adder (CSKA)

- Type a) : partial CPA with fast $c_k \to c_i$

$$c_i = \overline{P_{i-1:k}} c_i' + P_{i-1:k} c_k \quad \text{(bit group } (a_{i-1}, \dots, a_k))$$
$$P_{i-1:k} = p_{i-1} p_{i-2} \cdots p_k \quad \text{(group propagate)}$$

- 1) $P_{i-1:k} = 0$: $c_k \not\to c_i'$ and c_i' selected ($c_i' \to c_i$)
  2) $P_{i-1:k} = 1$: $c_k \to c_i'$ but c_i' **skipped** ($c_i' \not\to c_i$)

$\Rightarrow$ path $c_k \to c_i' \to c_i$ *never sensitized* $\Rightarrow$ fast $c_k \to c_i$
$\Rightarrow$ *false* path $\Rightarrow$ inherent *logic redundancy* $\Rightarrow$ problems in circuit optimization, timing analysis, and testing

- *Variable* group sizes (faster) : larger groups in the *middle* (minimize delays $a_0 \to c_k \to s_{i-1}$ and $a_k \to c_i \to s_{n-1}$)
- Partial CPA typ. is RCA or CSKA ($\Rightarrow$ *multilevel* CSKA)
- *Medium* speed-up at *small* hardware overhead (+ AND/bit + MUX/group)

$$A \approx 8n , \; T \approx 4n^{1/2} , \; AT \approx 32n^{3/2}$$

Carry-select adder (CSLA)

- Type a) : partial CPA with fast $c_k \to c_i$ and $c_k \to s_{i-1:k}$

$$s_{i-1:k} = \overline{c}_k s_{i-1:k}^0 + c_k s_{i-1:k}^1$$
$$c_i = \overline{c}_k c_i^0 + c_k c_i^1$$

- *Two CPAs* compute two possible results ($c_{in} = 0/1$), group carry-in c_k **selects** correct one afterwards
- *Variable* group sizes (faster) : larger groups at *end* (MSB) (balance delays $a_0 \to c_k$ and $a_k \to c_i^0$)
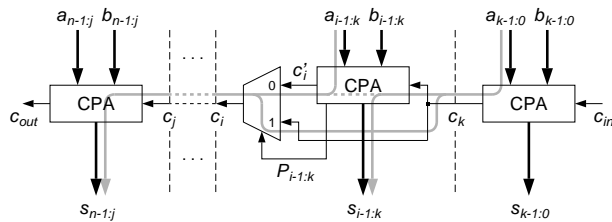- Part. CPA typ. is RCA, CSLA ($\Rightarrow$ *multil.* CSLA), or CLA
- *High* speed-up at *high* hardware overhead (+ MUX/bit + (CPA + MUX)/group)

$$A \approx 14n , \; T \approx 2.8n^{1/2} , \; AT \approx 39n^{3/2}$$
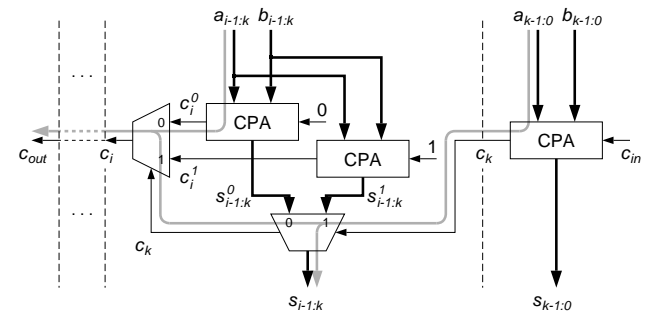
Carry-increment adder (CIA)

- Type a) : partial CPA with fast $c_k \to c_i$ and $c_k \to s_{i-1:k}$

$$s_{i-1:k} = s_{i-1:k}' + c_k , \; c_i = c_i' + P_{i-1:k} c_k$$
$$P_{i-1:k} = p_{i-1} p_{i-2} \cdots p_k \quad \text{(group propagate)}$$

- Result is **incremented** after addition, if $c_k = 1$ [12, 11]
- *Variable* group sizes (faster) : larger groups at *end* (MSB) (balance delays $a_0 \to c_k$ and $a_k \to c_i'$)
- Part. CPA typ. is RCA, CIA ($\Rightarrow$ *multilevel* CIA) or CLA
- *High* speed-up at *medium* hardware overhead (+ AND/bit + (incrementer + AND-OR)/group)
- Logic of CPA and incrementer can be *merged* [11]

$$A \approx 10n , \; T \approx 2.8n^{1/2} , \; AT \approx 28n^{3/2}$$
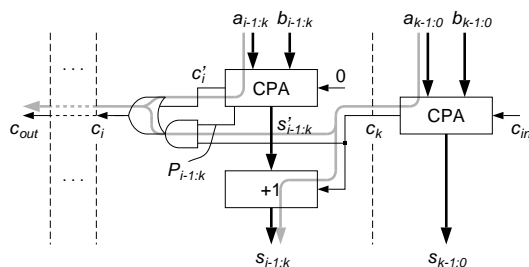
- Example : gate-level schematic of *carry-incr.* adder (CIA)
  ○ only 2 different logic cells (*bit-slices*) : IHA and IFA

T	4	6	10	12	14	16	18	20	22	24	26	28	...	38
max n_{group}			2	3	4	5	6	7	8	9	10	11	...	16
n	1	2	4	7	11	16	22	29	37	46	56	67	...	137
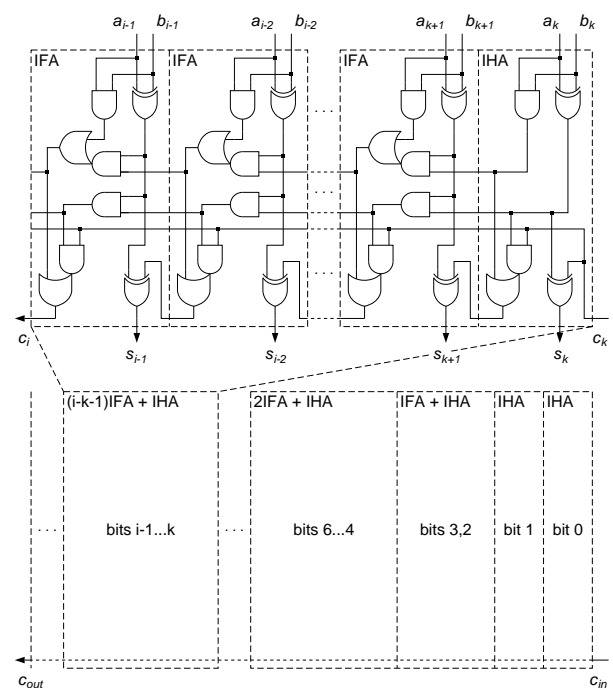
Conditional-sum adder (COSA)

- Type a) : optimized *multilevel CSLA* with $(\log n)$ levels (i.e. double CPAs are merged at higher levels)

- Correct sum bits ($s^0_{i-1:k}$ or $s^1_{i-1:k}$) are (**conditionally**) selected through $(\log n)$ levels of multiplexers

- Bit groups of size 2^l at level l

- Higher *parallelism*, more *balanced* signal paths

- *Highest* speed-up at *highest* hardware overhead ($2\,\text{RCA}$ + more than $(\log n)$ MUX/bit)

$$A \approx 3n \log n \, , \ T \approx 2 \log n \, , \ AT \approx 6n \log^2 n$$
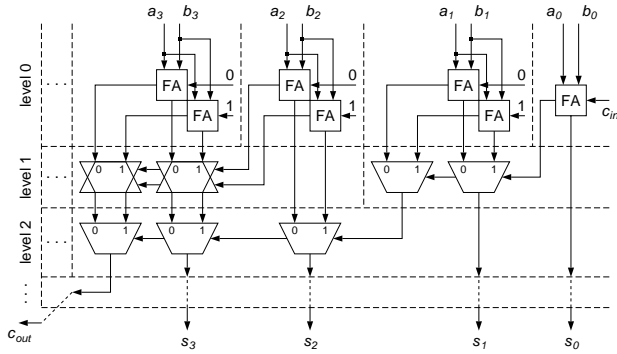
Carry-lookahead adder (CLA), traditional

- Type b) : carries **looked ahead** before sum bits computed

- Typically *4-bit blocks* used (e.g. standard IC SN74181)

$$
\begin{aligned}
c_0 &= c'_0 \\
c_1 &= g_0 + p_0 c'_0 \\
c_2 &= g_1 + p_1 g_0 + p_1 p_0 c'_0 \\
c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c'_0 \\
g'_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \\
p'_3 &= p_3 p_2 p_1 p_0
\end{aligned}
$$
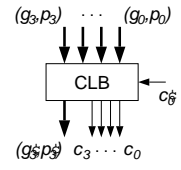
- *Hierarchical* arrangement using $(\frac{1}{2} \log n)$ levels : (g'_3, p'_3) passed up, c'_0 passed down between levels

- *High* speed-up at *medium* hardware overhead

$$A \approx 14n \, , \ T \approx 4 \log n \, , \ AT \approx 56n \log n$$

+ preprocessing : $g_i = a_i b_i$, $p_i = a_i \oplus b_i$
+ postprocessing : $s_i = p_i \oplus c_i$
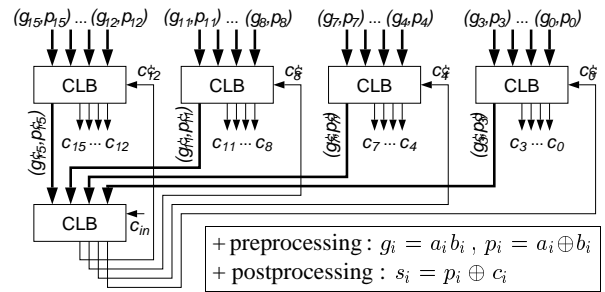
Parallel-prefix adders (PPA)

- Type b) : *universal* adder architecture comprising RCA, CIA, CLA, and more (i.e. entire range of *area-delay trade-offs* from slowest RCA to fastest CLA)

- *Preprocessing*, *carry-lookahead*, and *postprocessing* step

- Carries calculated using **parallel-prefix** algorithms

+ *High regularity* : suitable for synthesis and layout

+ *High flexibility* : special adders, other arithmetic operations, exchangeable prefix algorithms (i.e. speeds)

+ *High performance* : smallest and fastest adders

$$A \approx 5n + 3A_\bullet \, , \ T = 4 + 2T_\bullet$$

preprocessing:

$$g_i = a_i b_i$$
$$p_i = a_i \oplus b_i$$

carry-lookahead: prefix algorithm

postprocessing:

$$s_i = p_i \oplus c_i$$
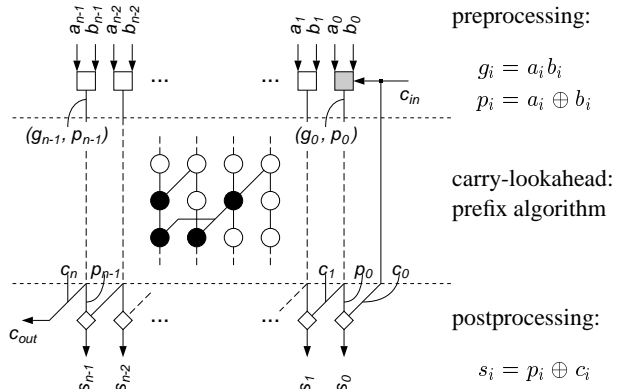
Prefix problem

- Inputs $(x_{n-1}, \ldots, x_0)$, outputs $(y_{n-1}, \ldots, y_0)$, associative binary operator $\bullet$ [11, 13]

$$(y_{n-1}, \ldots, y_0) = (x_{n-1} \bullet \cdots \bullet x_0, \ \ldots, \ x_1 \bullet x_0, \ x_0) \quad \text{or}$$
$$y_0 = x_0 \, , \ y_i = x_i \bullet y_{i-1} \, ; \ i = 1, \ldots, n-1 \quad \text{(r.m.a.)}$$

- Associativity of $\bullet$ $\Rightarrow$ *tree structures* for evaluation :

$$x_3 \bullet (x_2 \bullet \underbrace{(x_1 \bullet x_0)}_{y_1 = Y^1_{1:0}}) = \underbrace{(x_3 \bullet x_2)}_{Y^1_{3:2}} \bullet \underbrace{(x_1 \bullet x_0)}_{y_1 = Y^1_{1:0}} \, , \text{ but } y_2 \, ?$$

- *Group variables* $Y^l_{i:k}$: covers bits $(x_k, \ldots, x_i)$ at level l

- *Carry-propagation* is prefix problem : $Y^l_{i:k} = (G^l_{i:k}, P^l_{i:k})$

$$
\begin{aligned}
(G^0_{i:i}, P^0_{i:i}) &= (g_i, p_i) \\
(G^l_{i:k}, P^l_{i:k}) &= (G^{l-1}_{i:j+1}, P^{l-1}_{i:j+1}) \bullet (G^{l-1}_{j:k}, P^{l-1}_{j:k}) \, ; \ k \le j \le i \\
&= (G^{l-1}_{i:j+1} + P^{l-1}_{i:j+1} G^{l-1}_{j:k}, P^{l-1}_{i:j+1} P^{l-1}_{j:k}) \\
c_{i+1} &= G^m_{i:0} \, ; \ i = 0, \ldots, n-1 \, , \ l = 1, \ldots, m
\end{aligned}
$$

- *Parallel-prefix* algorithms [11] :
  - *multi-tree structures* $(T = O(n) \to O(\log n))$
  - *sharing subtrees* $(A = O(n^2) \to O(n \log n))$
  - different algorithms trading *area* vs. *delay* (influences also from *wiring* and maximum *fan-out* FO_{max})

Prefix algorithms

- Algorithms visualized by *directed acyclic graphs* (DAG) with array structure (n bits $\times$ m levels)

- Graph *vertex* symbols :

$(G_{i:j+1}^{l-1}, P_{i:j+1}^{l-1})$ $(G_{j:k}^{l-1}, P_{j:k}^{l-1})$ $(G_{i:k}^{l-1}, P_{i:k}^{l-1})$

$(G_{i:k}^{l}, P_{i:k}^{l})$ $(G_{i:k}^{l}, P_{i:k}^{l})$ $(G_{i:k}^{l}, P_{i:k}^{l})$ $(G_{i:k}^{l}, P_{i:k}^{l})$

(contains logic for •)   *(contains no logic)*
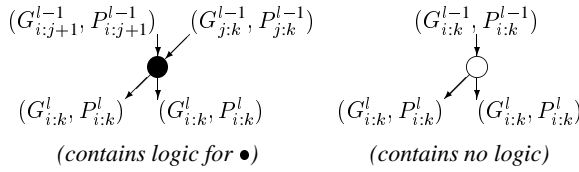
- Performance measures :

  $A_\bullet$: graph *size* (number of black nodes)

  $T_\bullet$: graph *depth* (number of black nodes on critical path)

- *Serial*-prefix algorithm ($\Rightarrow$ RCA)

$$\boxed{A_\bullet = n - 1 \;,\; T_\bullet = n - 1 \;,\; FO_{max} = 2}$$
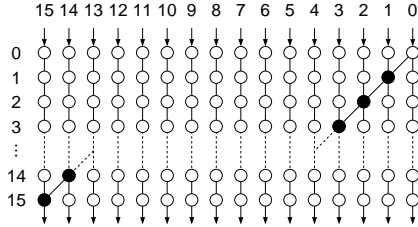
- *Sklansky* parallel-prefix algorithm ($\Rightarrow$ PPA-SK)

  ○ *Tree-like* collection, *parallel* redistribution of carries

$$\boxed{A_\bullet \approx \tfrac{1}{2}n \log n \;,\; T_\bullet = \lceil \log n \rceil \;,\; FO_{max} \approx \tfrac{1}{2}n}$$
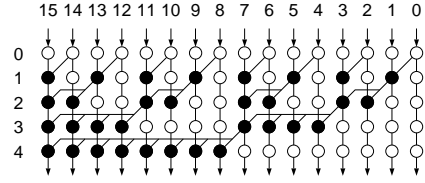
- *Brent-Kung* parallel-prefix algorithm ($\Rightarrow$ PPA-BK)

  ○ *Traditional CLA* is PPA-BK with 4-bit groups
  ○ *Tree-like* redistribution of carries (fan-out tree)

$$\boxed{\begin{array}{c} A_\bullet = 2n - \lceil \log n \rceil - 2 \;,\; T_\bullet = 2\lceil \log n \rceil - 2 \\ FO_{max} \approx \log n \end{array}}$$
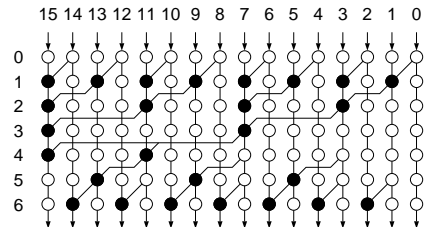
- *Kogge-Stone* parallel-prefix algorithm ($\Rightarrow$ PPA-KS)

  ○ very high *wiring* requirements

$$\boxed{A_\bullet \approx n \log n - n + 1 \;,\; T_\bullet = \lceil \log n \rceil \;,\; FO_{max} = 2}$$
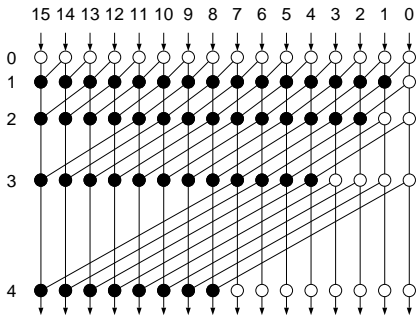
- *Carry-increment* parallel-prefix algorithm ($\Rightarrow$ CIA)

$$\boxed{A_\bullet \approx 2n - 1.4n^{1/2} \;,\; T_\bullet \approx 1.4n^{1/2} \;,\; FO_{max} \approx 1.4n^{1/2}}$$
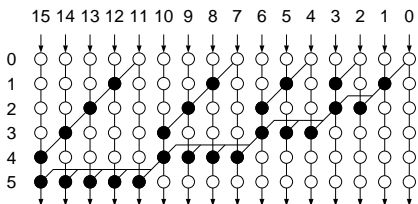
- *Mixed serial/parallel*-prefix algorithm ($\Rightarrow$ RCA + PPA)

  ○ linear *size-depth trade-off* using parameter k :

$$\boxed{0 \le k \le n - 2\lceil \log n \rceil + 2}$$

  ○ $k = 0$: serial-prefix graph
  $k = n - 2\lceil \log n \rceil + 1$: Brent-Kung parallel-prefix graph

  ○ *fills gap* between RCA and PPA-BK (i.e. CLA) in steps of single •-operations

$$\boxed{A_\bullet = n - 1 + k \;,\; T_\bullet = n - 1 - k \;,\; FO_{max} = \text{var.}}$$
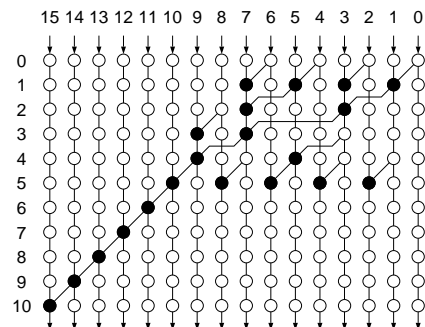
- Example : 4-bit *parallel-prefix* adder (PPA-SK)
  - efficient *AND-OR-prefix* circuit for the generate and *AND-prefix* circuit for the propagate signals
  - *optimization*: alternatingly AOI-/OAI- resp. NAND-/ NOR-gates (inverting gates are smaller and faster)
  - can also be realized using two *MUX-prefix* circuits
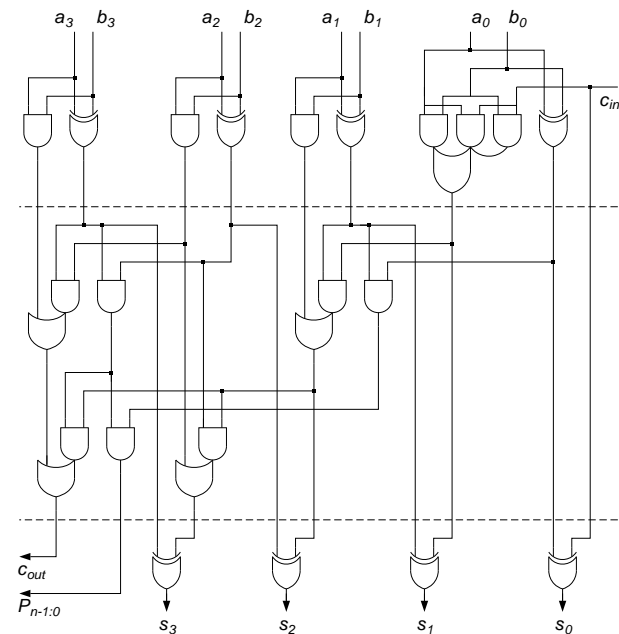
Prefix adder synthesis

- Local prefix graph *transformation* :
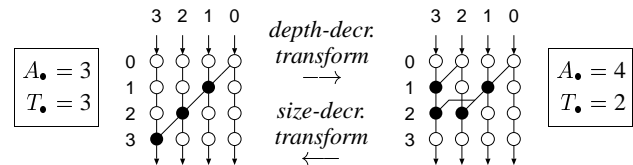
- Repeated (local) prefix transformations result in *overall minimization* of graph depth or size $\Rightarrow$ which sequence ?

- *Goal:* minimal size (area) at given depth (delay)

- Simple *algorithm* for sequence of applied transforms :

  Step 1 : *prefix graph compression* (depth minimization) : depth-decr. transforms in *right-to-left bottom-up* order

  Step 2 : *prefix graph expansion* (size minimization) : size-decreasing transforms in *left-to-right top-down* order, if allowed depth not exceeded

- *Prefix adder synthesis* : 1) generate serial-prefix graph, 2) graph compression, 3) depth-controlled graph expansion, 4) generate pre-/postprocessing and prefix logic

- + Generates *all* previous prefix graphs (except PPA-KS)

- + *Universal adder synthesis* algorithm : generates area-optimal adders for any given timing constraints [11] (including non-uniform signal arrival times)

Multilevel adders

- *Multilevel* versions of adders of type a) possible (CSKA, CSLA, and CIA; notation: 2-level CIA = CIA-2L)

- + *Delay* is $O(n^{1/(m+1)})$ for m levels

- – *Area* increase small for CSKA and CIA, high for CSLA ($\Rightarrow$ COSA)

- Difficult computation of optimal *group sizes*

Hybrid adders

- Arbitrary *combinations* of speed-up techniques possible $\Rightarrow$ hybrid/mixed adder architectures

- *Often used* combinations : CLA and CSLA [14]

- – *Pure* architectures usually perform best (at gate-level)

Transistor-level adders

- Influence of *logic styles* (e.g. dynamic logic, pass-transistor logic $\Rightarrow$ faster)

- + Efficient *transistor-level* implementation of ripple-carry chains (Manchester chain) [14]

- + *Combinations* of speed-up techniques make sense

- – Much higher *design effort*

- Many *efficient* implementations exist and published

Self-timed adders

- *Average* carry-propagation length : $\log n$

- + *RCA* is fast in average case ($\tilde{T} = O(\log n)$), slow in worst case $\Rightarrow$ suitable for *self-timed asynchronous* designs [15]

- – *Completion detection* is not trivial

Adder performance comparisons

- *Standard-cell* implementations, $0.8\mu m$ process
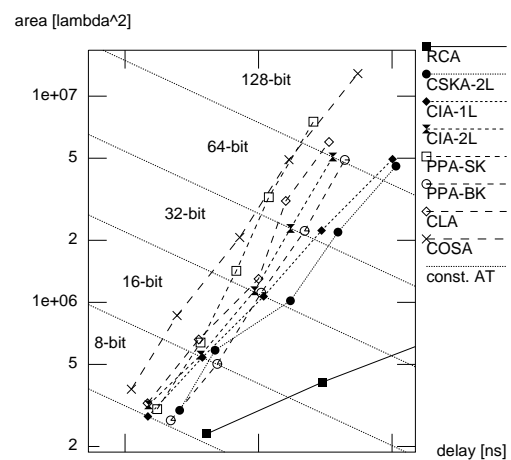
- Complexity *comparison* under the unit-gate model

adder	A	T	AT	opt.[1]	syn.[2]
RCA	$7n$	$2n$	$14n^2$	aaa	$\checkmark$
CSKA-1L	$8n$	$4n^{1/2}$	$32n^{3/2}$	aat [3]	
CSKA-2L	$8n$	$xn^{1/3}$ [4]	$xn^{4/3}$ [4]	—	
CSLA-1L	$14n$	$2.8n^{1/2}$	$39n^{3/2}$	—	
CIA-1L	$10n$	$2.8n^{1/2}$	$28n^{3/2}$	att	$\checkmark$
CIA-2L	$10n$	$3.6n^{1/3}$	$36n^{4/3}$	att	$\checkmark$
CIA-3L	$10n$	$4.4n^{1/4}$	$44n^{5/4}$	—	$\checkmark$
PPA-SK	$\frac{3}{2}n \log n$	$2 \log n$	$3n \log^2 n$	ttt	$\checkmark$
PPA-BK	$10n$	$4 \log n$	$40n \log n$	att	$\checkmark$
PPA-KS	$3n \log n$	$2 \log n$	$6n \log^2 n$	—	
CLA [5]	$14n$	$4 \log n$	$56n \log n$	—	$(\checkmark)$
COSA	$3n \log n$	$2 \log n$	$6n \log^2 n$	—	

[1]   optimality regarding area and delay
    aaa : smallest area, longest delay
    aat : small area, medium delay
    att : medium area, short delay
    ttt : large area, shortest delay
    — : not optimal
[2] obtained from prefix adder synthesis
[3] automatic logic optimization not possible (redundancy)
[4] exact factors not calculated
[5] corresponds to 4-bit PPA-BK

4.4 Carry-Save Adder (CSA)

a) *Adds* three n-bit operands A_0, A_1, A_2 performing *no* carry-propagation (i.e. carries are **saved**) [1]

$$(C, S) = C + S = A_0 + A_1 + A_2$$

$$2c_{i+1} + s_i = a_{0,i} + a_{1,i} + a_{2,i} \; ; \\ i = 0, 1, \ldots, n-1 \quad \text{(n.)}$$
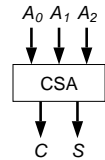
b) *Adds* one n-bit operand to an n-digit carry-save operand

$$(C, S)_{out} = A + (C, S)_{in}$$

– Result is in redundant *carry-save* format (n digits), represented by two n-bit numbers S (sum bits) and C (carry bits)

+ *Parallel* arrangement of n full-adders, *constant* delay

$$A = 7n \, , \; T = 4$$
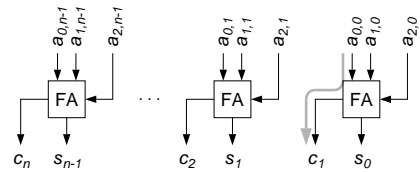
- Multi-operand carry-save adders ($m > 3$)
  $\Rightarrow$ *adder array* (linear arrangement), *adder tree* (tree arr.)

4.5 Multi-Operand Adders

- Add *three* or *more* ($m > 2$) n-bit operands, yield $(n + \lceil \log m \rceil)$-bit result in *irredundant* number rep. [1, 2]

Array adders

- Realization by **array adders** : (see figures on next page)
  a) linear arrangement of *CPAs*
  b) linear arr. of *CSAs* (**adder array**) and *final CPA*

- a) and b) differ in *bit arrival times* at final CPA :
  $\Rightarrow$ if CPA = RCA : a) and b) have same overall delay
  $\Rightarrow$ if fast final CPA : uniform bit arrival times required
                  $\Rightarrow$ CSA array (b)

- *Fast implementation* : CSA array + fast final CPA
  (note: array of fast CPAs *not* efficient/necessary)

$$A = (m-2)A_{CSA} + A_{CPA} \\ T = (m-2)T_{CSA} + T_{CPA}$$

CPA = RCA : $\quad \dfrac{A = O(mn + n)}{T = O(m + n)}$

Fast CPA : $\quad \dfrac{A = O(mn + n \log n)}{T = O(m + \log n)}$

---
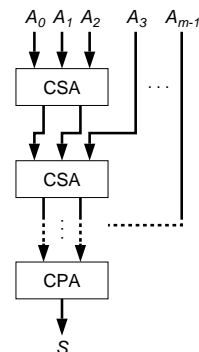
a) 4-operand *CPA (RCA) array* :
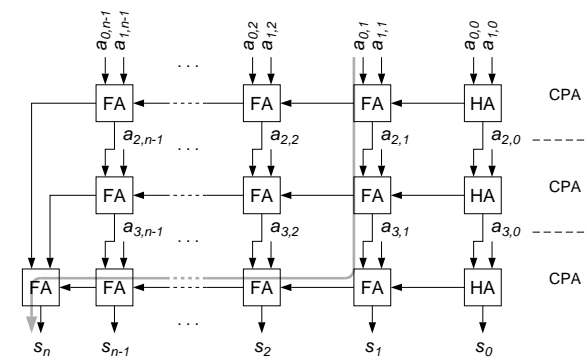
b) 4-operand *CSA array* with *final CPA* (RCA) :
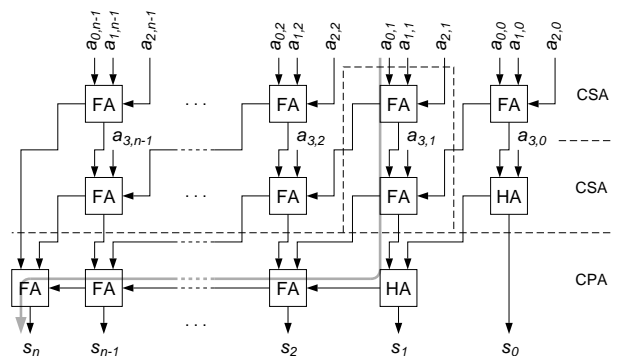
(m, 2)-compressors

$$2\left(c + \sum_{l=0}^{m-4} c_{out}^l\right) + s = \sum_{i=0}^{m-1} a_i + \sum_{l=0}^{m-4} c_{in}^l$$
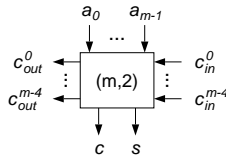
- *1-bit adders* (similar to (m, k)-counters) [16]

- **Compresses** m bits down to 2 by forwarding $(m-3)$ *intermediate carries* to next higher bit position

- Is *bit-slice* of multi-operand CSA array (see prev. page)

- + *No* horizontal carry-propagation (i.e. $c_{in}^l \rightarrow c_{out}^k$, $\underline{k > l}$)

- Built from *full-adders* ($= (3, 2)$-compressor) or *(4, 2)-compressors* arranged in linear or tree structures

- Example : 4-operand adder using *(4, 2)-compressors*
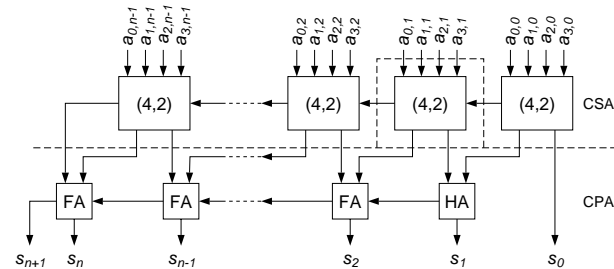
$$A = 7(m - 2)$$
$$T_{LIN} = 4(m - 2) , \ T_{TREE} = 6(\lceil \log m \rceil - 1)$$

- Optimized *(4, 2)-compressor* :

  ○ 2 full-adders *merged* and *optimized* (i.e. XORs arranged in *tree structure*)

$$\boxed{A = 14 , \ T = 8} \qquad \boxed{A = 14 , \ T = 6}$$

with full-adders   *optimized*
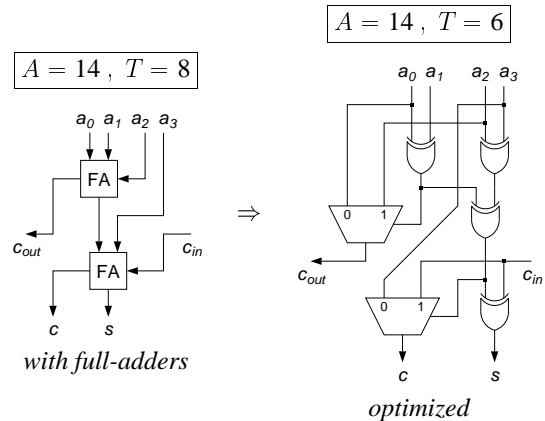
- + *same* area, 25% *shorter* delay

- ○ *SD-FA* (signed-digit full-adder) is similar to $(4, 2)$-compressor regarding structure and complexity

---

- *Advantages* of $(4, 2)$-compressors over FAs for realizing (m, 2)-compressors :

  ○ higher *compression rate* (4:2 instead of 3:2)

  ○ *less deep* and *more regular* trees

tree depth		0	1	2	3	4	5	6	7	8	9	10
# operands	FA	2	3	4	6	9	13	19	28	42	63	94
	(4,2)	2	4	8	16	32	64	128	$\cdots$			

- Example : *(8, 2)-compressor*

$$\boxed{A = 42 , \ T = 16} \qquad \boxed{A = 42 , \ T = 12}$$

full-adder tree   *(4, 2)-compressor tree*
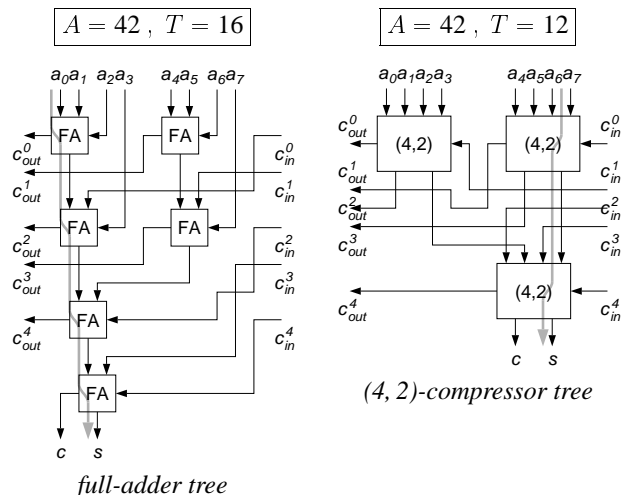
Tree adders (Wallace tree)

- **Adder tree** : n-bit m-operand *carry-save adder* composed of n tree-structured *(m, 2)-compressors* [1, 17]

- **Tree adders** : fastest multi-operand adders using an *adder tree* and a *fast final CPA*

$$A = A_{(m,2)} \cdot n + A_{CPA} = O(mn + n \log n)$$
$$T = T_{(m,2)} + T_{CPA} = O(\log m + \log n)$$

Adder arrays and adder trees revisited

- Some FA can often be *replaced by HA* or *eliminated* (i.e. redundant due to constant inputs)

- Number of (irredundant) FA *does not* depend on adder structure, but number of HA *does*

- An m-operand adder accomodates $(m - 1)$ *carry inputs*

- Adder trees ($T = O(\log n)$) are *faster* than adder arrays ($T = O(n)$) at same amount of gates ($A = O(mn)$)

- Adder trees are *less regular* and have *more complex routing* than adder arrays $\Rightarrow$ larger area, difficult layout (i.e. limited use in layout generators)

4.6 Sequential Adders

Bit-serial adder : Sequential n-*bit adder*

$$A = A_{FA} + A_{FF}$$
$$T = T_{FA} + T_{FF}$$
$$L = n$$
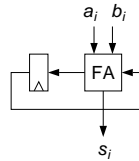
Accumulators : Sequential m-*operand adders*

- With *CPA*

$$A = A_{CPA} + A_{REG}$$
$$T = T_{CPA} + T_{REG}$$
$$L = m$$
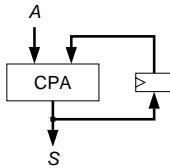
- With *CSA* and *final CPA*
  - Allows *higher* clock rates
  - Final CPA *too slow* :
    $\Rightarrow$ *pipelining* or *multiple cycles* for evaluation

$$A = A_{CSA} + A_{CPA} + 4A_{REG}$$
$$T = T_{CSA} + T_{REG}$$
$$L = m$$
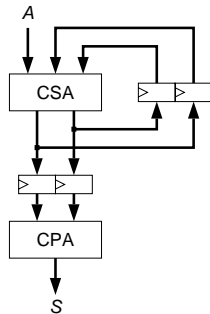
- *Mixed CSA/CPA* : CSA with partial CPAs (i.e. fewer carries saved), trade-off between speed and register size

---

5 Simple / Addition-Based Operations

5.1 Complement and Subtraction

2's complementer (negation)

$$-A = \overline{A} + 1$$
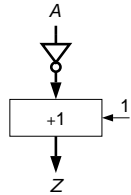
2's complement subtractor

$$A - B = A + (-B)$$
$$= A + \overline{B} + 1$$
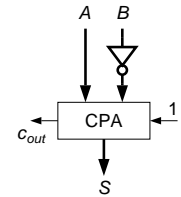
2's complement adder/subtractor

$$A \pm B = A + (-1)^{sub} B$$
$$= A + (B \oplus sub) + sub$$
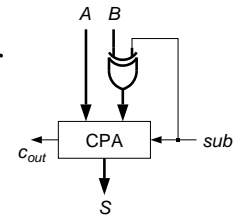
1's complement adder

$$A + B \pmod{2^n - 1}$$
$$= A + B + c_{out}$$
$$(\textit{end-around carry})$$
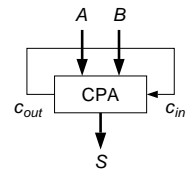
5.2 Increment / Decrement

Incrementer

- Adds a *single bit* c_{in} to an n-bit operand A

$$(c_{out}, Z) = c_{out}2^n + Z = A + c_{in}$$

$$z_i = a_i \oplus c_i$$
$$c_{i+1} = a_i c_i \; ; \; i = 0, \ldots, n-1$$
$$c_0 = c_{in} \, , \; c_{out} = c_n \quad (\text{r.m.a.})$$
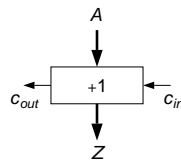
- Corresponds to *addition* with $B = 0$ ($\Rightarrow$ FA $\rightarrow$ HA)
- Example : *Ripple-carry* incrementer using half-adders

$$A = 3n \, , \; T = n + 1 \, , \; AT \approx 3n^2$$
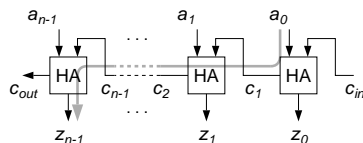
or using *incrementer slices* (= half-adder)
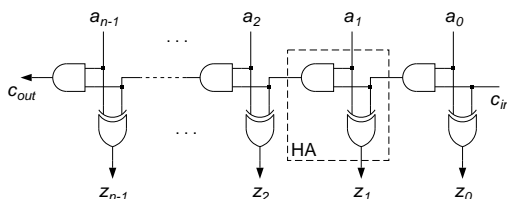
- *Prefix problem* : $C_{i:k} = C_{i:j+1}C_{j:k}$ $\Rightarrow$ *AND-prefix* struct.

$$A \approx \tfrac{1}{2}n \log n + 2n \, , \; T = \lceil \log n \rceil + 2 \, , \; AT \approx \tfrac{1}{2}n \log^2 n$$

Decrementer $(c_{out}, Z) = A - c_{in}$
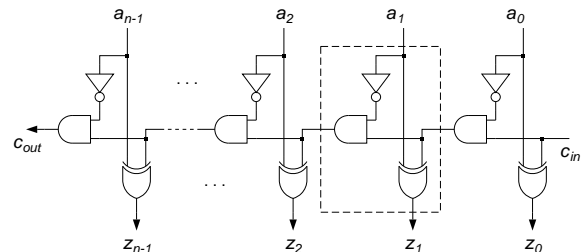
Incrementer-decrementer

$$(c_{out}, Z) = A \pm c_{in} = A + (-1)^{dec} c_{in}$$
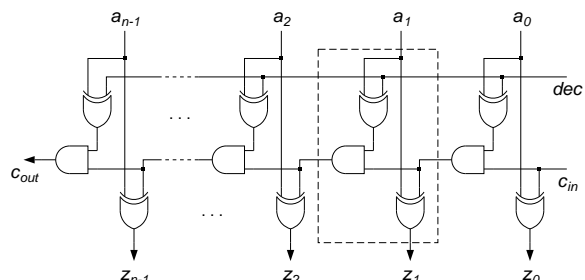
Fast incrementers

- 4-bit incrementer using *multi-input gates* :

a_3 a_2 a_1 a_0

c_{in}

c_{out}

z_3 z_2 z_1 z_0
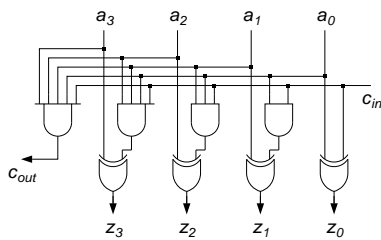
- 8-bit *parallel-prefix* incrementer (Sklansky AND-prefix structure) :

a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0

c_{in}

c_{out} z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0
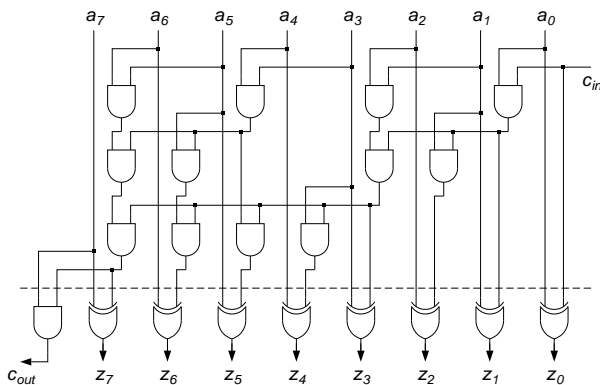
Gray incrementer

- Increments in *Gray number system*

$$
\begin{aligned}
c_0 &= a_{n-1} \oplus a_{n-2} \oplus \cdots \oplus a_0 \ \text{(parity)} \\
c_{i+1} &= \overline{a_i}\, c_i \ ; \ i = 0, \ldots, n-3 \ \text{(r.m.a.)} \\
z_0 &= \overline{a_0} \oplus c_0 \\
z_i &= a_i \oplus a_{i-1}\, c_{i-1} \ ; \ i = 1, \ldots, n-2 \\
z_{n-1} &= a_{n-1} \oplus c_{n-2}
\end{aligned}
$$

- *Prefix problem* $\Rightarrow$ AND-prefix structure

5.3 Counting

- *Count* clock cycles $\Rightarrow$ **counter**,
  divide clock frequency $\Rightarrow$ **frequency divider** (c_{out})

Binary counter

- *Sequential* in-/decrementer
- Incrementer *speed-up techniques* applicable
- *Down-* and *up-down-counters* using decrementers / incrementer-decrementers

c_{out} +1 c_{in}

clk

Q

- Example : *Ripple-carry* up-counter using *counter slices* (= HA + FF), c_{in} is count enable

c_{out} c_{in}

$\cdots$

q_{n-1} q_2 q_1 q_0
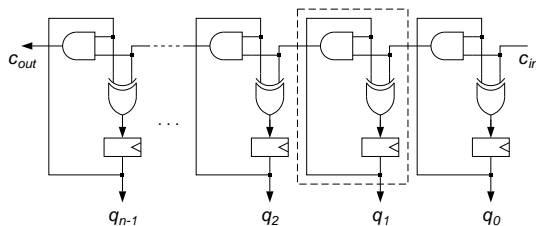
- *Asynchronous* counter using toggle-flip-flops (lower toggle rate $\Rightarrow$ lower power)

T $\cdots$ T T T clk

q_{n-1} q_2 q_1 q_0
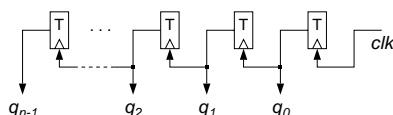
- *Fast* divider ($T = O(1)$) using *delayed-carry* numbers (irredundant carry-save representation of -1 allows using fast carry-save incrementer) [8]

Gray counter

- Counter using *Gray incrementer*

Ring counters

- *Shift register* connected to **ring** :

q_{n-1} q_2 q_1 q_0
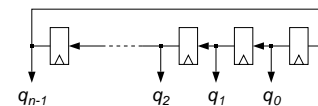
- State is *not encoded* $\Rightarrow$ n FF for counting n states
- Must be *initialized* correctly (e.g. $00\cdots01$)
- Applications:
  - *fast dividers* (no logic between FF)
  - state counter for *one-hot coded* FSMs
- *Johnson / twisted-ring* counter (inverted feed-back) :

q_{n-1} q_2 q_1 q_0
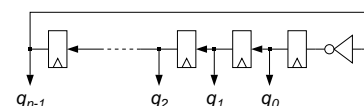
  - n FF for counting $2n$ states

5.4 Comparison, Coding, Detection

Comparison operations

$$EQ = (A = B) \qquad \text{(equal)}$$
$$NE = (A \neq B) = \overline{EQ} \qquad \text{(not equal)}$$
$$GE = (A \geq B) \qquad \text{(greater or equal)}$$
$$LT = (A < B) = \overline{GE} \qquad \text{(less than)}$$
$$GT = (A > B) = GE \cdot \overline{EQ} \qquad \text{(greater than)}$$
$$LE = (A \leq B) = \overline{GT} = \overline{GE} + EQ \qquad \text{(less or equal)}$$

Equality comparison

$$EQ = (A = B)$$

$$eq_{i+1} = (a_i = b_i)\, eq_i$$
$$= (a_i \odot b_i)\, eq_i \,;$$
$$i = 0, \ldots, n-1$$
$$eq_0 = 1 \,,\; EQ = eq_n \quad \text{(r.s.a.)}$$
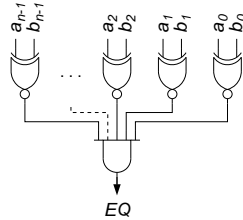
Magnitude comparison

$$GE = (A \geq B)$$

$$ge_{i+1} = (a_i > b_i) + (a_i = b_i)\, ge_i$$
$$= a_i \overline{b_i} + (a_i \odot b_i)\, ge_i \,;\; i = 0, \ldots, n-1$$
$$ge_0 = 1 \,,\; GE = ge_n \quad \text{(r.s.a.)}$$

Comparators

- *Subtractor* $(A - B)$:

$$GE = c_{out}$$
$$EQ = P_{n-1:0}$$
$$\text{(for } free \text{ in PPA)}$$
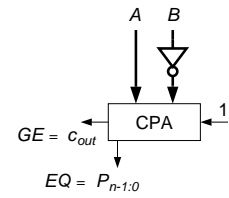
$$A_{RCA} = 7n \,,\; T_{RCA} = 2n \quad \text{or}$$
$$A_{PPA-KS} \approx \tfrac{3}{2} n \log n \,,\; T_{PPA-KS} \approx 2 \log n$$

- *Optimized* comparator :
  - removing *redundancies* in subtractor (unused s_i)
  - *single-tree* structure $\Rightarrow$ speed-up at *no* cost :

$$A = 6n \,,\; T_{LIN} = 2n \,,\; T_{TREE} \approx 2 \log n$$

  - example : ripple comparator using *comparator slices*
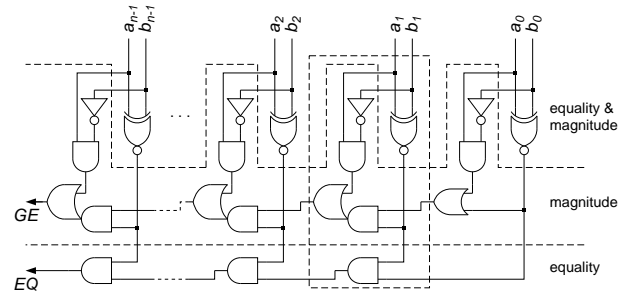
Decoder

- *Decodes* binary number $A_{n-1:0}$ to vector $Z_{m-1:0}$ $(m = 2^n)$

$$z_i = \begin{cases} 1 & \text{if } A = i \\ 0 & \text{else} \end{cases} \;;\; i = 0, \ldots, m-1 \qquad \boxed{Z = 2^A}$$
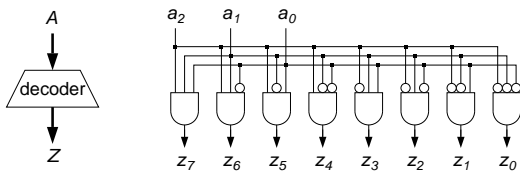
$$A = (n-1)2^n \,,\; T = \lceil \log n \rceil$$

Encoder

- *Encodes* vector $A_{m-1:0}$ to binary number $Z_{n-1:0}$ $(m = 2^n)$
  (condition: $\exists i \, \forall k \mid$ if $k = i$ then $a_k = 1$ else $a_k = 0$)

$$Z = i \;\text{if}\; a_i = 1 \,;\; i = 0, \ldots, m-1 \qquad \boxed{Z = \log_2 A}$$
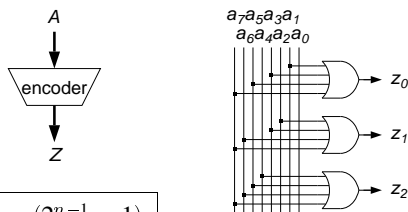
$$A = n(2^{n-1} - 1)$$
$$T = n - 1$$

(note: connections according to PPA-SK)

Detection operations

- *All-zeroes* detection : $\boxed{z = \overline{a_{n-1} + a_{n-2} + \cdots + a_0}}$

  All-ones detection : $\boxed{z = a_{n-1} a_{n-2} \cdots a_0}$ (r.s.a.)

$$A = n \,,\; T = \log n$$

- *Leading-zeroes* detection (LZD) :
  - for *scaling*, *normalization*, *priority encoding*
  - a) *non-encoded* output :

$$\{0\}1\{0|1\} \rightarrow \{0\}1\{0\}$$
$$(\text{e.g. } 000101 \rightarrow 000100)$$

$$A = 2n \,,\; T = n$$
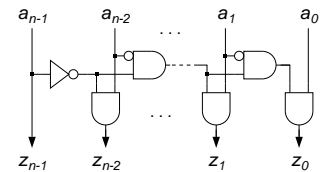
  - *prefix problem* (r.m.a.) $\Rightarrow$ AND-prefix structure

  - b) *encoded* output : + encoder

  - *signed* numbers : + *leading-ones detector* (LOZ)

5.5 Shift, Extension, Saturation

Shift :   a)   *shift* n-bit vector by k bit positions

        b)   *select* n out of more bits at position k

   • also: *logical* (= unsigned), *arithmetic* (= signed)

Rotation by k bit positions, n constant (logic operation)

Extension of *word lengths* by k bits ($n \rightarrow n + k$)

   (i.e. *sign-extension* for signed numbers)

Saturation to highest/lowest value after *over-/underflow*

shift a)	un-	l.	$a_{n-2}, \ldots, a_0, \underline{0}$	sll
	signed	r.	$\underline{0}, a_{n-1}, \ldots, a_1$	srl
	signed	l.	$a_{n-1}, \quad\quad a_{n-3}, \ldots, a_0, \underline{0}$	sla
		r.	$a_{n-1}, \underline{a_{n-1}}, a_{n-2}, \ldots, a_1$	sra
shift b)	unsigned		$a_{n+k-1}, \ldots, a_k$	
	signed		$a_{2n-1}, \quad a_{n+k-2}, \ldots, a_k$	
rotate		l.	$a_{n-2}, \ldots, a_0, \underline{a_{n-1}}$	rol
		r.	$\underline{a_0}, a_{n-1}, \ldots, a_1$	ror
extend	un-	l.	$\underline{0}, a_{n-1}, \ldots, a_0$	
	signed	r.	$a_{n-1}, \ldots, a_0, \underline{0}$	
	signed	l.	$a_{n-1}, \underline{a_{n-1}}, a_{n-2}, \ldots, a_0$	
		r.	$a_{n-1}, \quad\quad a_{n-2}, \ldots, a_0, \underline{0}$	
saturate	unsigned		$a_{n-1}, \ldots, a_{n-1}$	
	signed		$a_{n-1}, \quad \overline{a}_{n-1}, \ldots, \overline{a}_{n-1}$	

• *Applications* :

   ○ adaption of *magnitude* (shift a)) or *word length* (extension) of operands (e.g. for addition)

   ○ *multiplication/division* by multiples of 2 (shift)

   ○ *logic* bit/byte operations (shift, rotation)

   ○ *scaling* of numbers for *word-length reduction* (i.e. ignore leading zeroes, shift b)) or *normalization* (e.g. of floating-point numbers, shift a)) using LZD

   ○ *reducing error* after over-/underflow (saturation)

• *Implementation* of shift/extension/rotation by

   ○ constant values : *hard-wired*

   ○ variable values : *multiplexers*

   ○ n possible values : n–by–n *barrel-shifter/rotator*

• Example : 4–by–4 *barrel-rotator*

$$A = O(n^2)$$
$$T = O(\log n)$$

multiplexers   *tristate buffers*
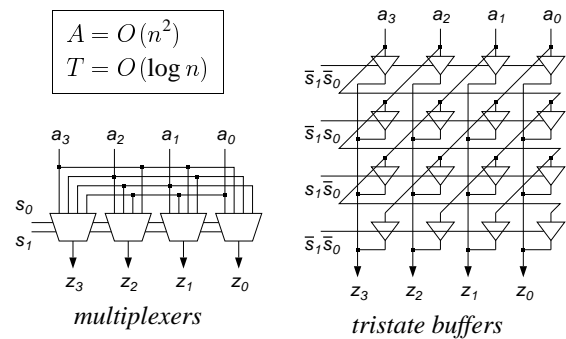
5.6 Addition Flags

flag	formula	description
C	c_n	*carry* flag
V	$c_n \oplus c_{n-1}$ $a_n b_n \overline{s}_n + \overline{a}_n \overline{b}_n s_n$	signed *overflow* flag
Z	$\forall i : s_i = 0$	*zero* flag
N	s_{n-1}	*negative* flag, sign

Implementation of adder with flags

C, N : for *free*

V : fast c_n, c_{n-1} computed by e.g. PPA $\Rightarrow$ very *cheap*

Z : a) $\underline{c_{in} = 1}$ (subtract.) : $Z = (A = B) = P_{n-1:0}$ (of PPA)

    b) $\underline{c_{in} = 0/1}$:

      1) $\boxed{Z = \overline{s_{n-1} + s_{n-2} + \cdots + s_0}}$ (r.s.a.)

         $\boxed{A = A_{CPA} + n \,,\; T_Z = T_{CPA} + \lceil \log n \rceil}$

      2) •   faster *without* final sum (i.e. carry prop.) [18]

         •   *example* :   01001 | 1 | 00 0

                     + 10110 | 1 | 00

                     = 00000 | 0 | 00

$$z_0 = ((a_0 \oplus b_0) \odot c_{in})$$
$$z_i = ((a_i \oplus b_i) \odot (a_{i-1} + b_{i-1}))$$
$$Z = z_{n-1} z_{n-2} \cdots z_0 \,;\; i = 0, \ldots, n-1 \;\; \text{(r.s.a.)}$$

$$\boxed{A = A_{CPA} + 3n \,,\; T_Z = 4 + \lceil \log n \rceil}$$

Basic and derived condition flags

condition	flag	formula	
		unsigned	signed
operation: $S = A + B$ (+) or $S = A - B$ (−)			
$S = 0$	zero	Z	Z
$S < 0$	negative	—	N
$S \geq 0$	positive	—	$\overline{N}$
$S > max$	overflow	C (+)	$V\overline{C}$
$S < min$	underflow	$\overline{C}$ (−)	VC
operation: $A - B$			
$A = B$	EQ	Z	Z
$A \neq B$	NE	$\overline{Z}$	$\overline{Z}$
$A \geq B$	GE	C	$\overline{N}\,\overline{V} + NV$
$A > B$	GT	$C\overline{Z}$	$(\overline{N}\,\overline{V} + NV)\overline{Z}$
$A < B$	LT	$\overline{C}$	$N\overline{V} + \overline{N}V$
$A \leq B$	LE	$\overline{C} + Z$	$N\overline{V} + \overline{N}V + Z$

• *Unsigned* and *signed* addition/subtraction only differ with respect to the *condition flags*

5.7 Arithmetic Logic Unit (ALU)
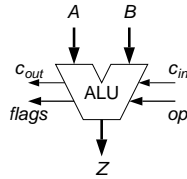
ALU operations

arithmetic	add	$A + B + c_{in}$	sub	$A - B - c_{in}$
	inc	$A + 1$	dec	$A - 1$
	pass	A	neg	$-A$
logic	and	$a_i b_i$	nand	$\overline{a_i b_i}$
	or	$a_i + b_i$	nor	$\overline{a_i + b_i}$
	xor	$a_i \oplus b_i$	xnor	$a_i \odot b_i$
	pass	a_i	not	$\overline{a_i}$
shift/ rotate	sll *	$A \ll 1$	srl *	$A \gg 1$
	sla *	$A \ll_a 1$	sra *	$A \gg_a 1$
	rol *	$A \ll_r 1$	ror *	$A \gg_r 1$

* s/ro : shift/rotate ; l/r : left/right ;

  l/a : logic (unsigned) / arithmetic (signed)

- Logic of adder/subtractor can partly be *shared* with logic operations

---

6 Multiplication

6.1 Multiplication Basics

- *Multiplies* two n-bit operands A and B [1, 2]
- *Product* P is $(2n)$-bit unsigned number or $(2n - 1)$-bit signed number
- Example : *unsigned* multiplication

$$P = A \cdot B = \sum_{i=0}^{n-1} a_i 2^i \cdot \sum_{j=0}^{n-1} b_j 2^j = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j 2^{i+j} \quad \text{or}$$

$$P_i = a_i \cdot B , \ P = \sum_{i=0}^{n-1} P_i 2^i ; \ i = 0, \ldots, n - 1 \ \text{(r.s.a.)}$$

Algorithm

1) Generation of n *partial products* P_i
2) *Adding up* partial products :

  a) *sequentially* (sequential shift-and-add),

  b) *serially* (combinational shift-and-add), or

  c) *in parallel*

Speed-up techniques

- *Reduce* number of partial products
- *Accelerate* addition of partial products

Sequential multipliers :
partial products generated and added *sequentially* (using *accumulator*)

$$A = O(n) , \ T = O(\log n) , \ L = n$$
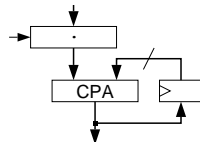
Array multipliers :
partial products generated and added *simultaneously* in linear array (using *array adder*)

$$A = O(n^2) , \ T = O(n)$$
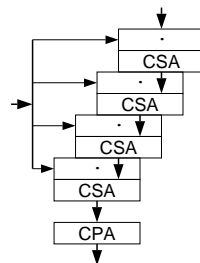
Parallel multipliers :
partial products generated in *parallel* and added *subsequently* in multi-operand adder (using *tree adder*)

$$A = O(n^2) , \ T = O(\log n)$$
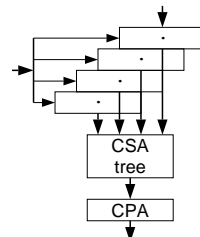
Signed multipliers :

  a) *complement* operands before and result after multiplication $\Rightarrow$ *unsigned* multiplication

  b) *direct* implementation (dedicated multiplier structure)

6.2 Unsigned Array Multiplier

- *Braun multiplier* : array multiplier for *unsigned* numbers

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

$$A = 8n^2 - 11n$$
$$T = 6n - 9$$

			$a_0 b_3$	$a_0 b_2$	$a_0 b_1$	$a_0 b_0$
		$a_1 b_3$	$a_1 b_2$	$a_1 b_1$	$a_1 b_0$	
	$a_2 b_3$	$a_2 b_2$	$a_2 b_1$	$a_2 b_0$		
$+$ $a_3 b_3$	$a_3 b_2$	$a_3 b_1$	$a_3 b_0$			
p_7 p_6	p_5	p_4	p_3	p_2	p_1	p_0

6.3 Signed Array Multipliers

Modified Braun multiplier

- *Subtract* bits with negative weight $\Rightarrow$ special FAs [1]

1 neg. bit :	$-a + b + c_{in} = \quad 2c_{out} - s$
2 neg. bits :	$a - b - c_{in} = -2c_{out} + s$

- *Replace* FAs in regions $\textcircled{1}$, $\textcircled{2}$, and $\textcircled{3}$ by : (input a at mark •)

  $$s = a \oplus b \oplus c_{in}$$
  $$c_{out} = \overline{a}b + \overline{a}c_{in} + bc_{in}$$

- Otherwise exactly *same structure* and *complexity* as Braun multiplier $\Rightarrow$ efficient and flexible

Baugh-Wooley multiplier

- *Arithmetic transformations* yield the following partial products (two additional ones) :

$$
\begin{array}{rrrrrrrr}
 & & & & \overline{a}_0b_3 & a_0b_2 & a_0b_1 & a_0b_0 \\
 & & & \overline{a}_1b_3 & a_1b_2 & a_1b_1 & a_1b_0 \\
 & & \overline{a}_2b_3 & a_2b_2 & a_2b_1 & a_2b_0 \\
 & a_3b_3 & a_3\overline{b}_2 & a_3\overline{b}_1 & a_3\overline{b}_0 \\
 & \overline{a}_3 & & & a_3 \\
+ & 1 & \overline{b}_3 & & & b_3 \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

- – *Less efficient* and *regular* than modified Braun multiplier

6.4 Booth Recoding

- *Speed-up technique* : reduction of partial products

Sequential multiplication

- *Minimal* (or canonical) *signed-digit* (SD) represent. of A
- + *One cycle* per non-zero partial product (i.e. $\forall a_i \mid a_i \neq 0$)
- – *Negative* partial products
- – *Data-dependent* reduction of partial products and latency

Combinational multiplication

- Only *fixed* reduction of partial product possible
- Radix-4 *modified Booth recoding* : 2 bits recoded to one multiplier digit $\Rightarrow n/2$ partial products

$$A = \sum_{i=0}^{n/2} \underbrace{(a_{2i-1} + a_{2i} - 2a_{2i+1})}_{\{-2,-1,0,+1,+2\}} 2^{2i} \; ; \; a_{-1} = 0$$

a_{2i+1}	a_{2i}	a_{2i-1}	P_i
0	0	0	$+\ \ 0$
0	0	1	$+\ \ B$
0	1	0	$+\ \ B$
0	1	1	$+2B$
1	0	0	$-2B$
1	0	1	$-\ \ B$
1	1	0	$-\ \ B$
1	1	1	$-\ \ 0$

- Applicable to *sequential*, *array*, and *parallel* multipliers

- – *additional* recoding logic and *more complex* partial product generation (MUX for shift, XOR for negation)

  $A : +8n$
  $T : +7$

  + adder array/tree *cut in half*

  $\Rightarrow$ considerably *smaller* (array and tree) $A : /2$

  $\Rightarrow$ much *faster* for adder arrays $T : /2$

  $\Rightarrow$ *slightly* or *not* faster for adder trees $T : -0$

- *Negative* partial products (avoid *sign-extension*) :

$$
\underbrace{\mathbf{p}_3\ \mathbf{p}_3\ \mathbf{p}_3}_{\text{ext. sign}}\ \mathbf{p}_3\ p_2\ p_1\ p_0 = \quad 0\ 0\ 0\ -\mathbf{p}_3\ p_2\ p_1\ p_0
$$
$$
= \quad \quad \quad 1
$$
$$
+\ 1\ 1\ 1\ \ \overline{\mathbf{p}}_3\ \ p_2\ p_1\ p_0
$$

$$
\begin{array}{lllllll}
\mathbf{p}_{03} & \mathbf{p}_{03} & \mathbf{p}_{03} & \mathbf{p}_{03} & p_{02} & p_{01} & p_{00} \\
\mathbf{p}_{13} & \mathbf{p}_{13} & \mathbf{p}_{13} & p_{12} & p_{11} & p_{10} \\
\mathbf{p}_{23} & \mathbf{p}_{23} & p_{22} & p_{21} & p_{20} \\
\mathbf{p}_{33} & p_{32} & p_{31} & p_{30} & & & + \\
\hline
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
\quad \rightarrow \quad
\begin{array}{lllllll}
& & & & & 1 \\
& \overline{\mathbf{p}}_{03} & p_{02} & p_{01} & p_{00} \\
& \overline{\mathbf{p}}_{13} & p_{12} & p_{11} & p_{10} \\
& \overline{\mathbf{p}}_{23} & p_{22} & p_{21} & p_{20} \\
\overline{\mathbf{p}}_{33} & p_{32} & p_{31} & p_{30} & & & + \\
\hline
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

- Suited for *signed* multiplication (incl. Booth recod.)

- Extend A for *unsigned* multiplication : $a_n = 0$

- *Radix-8* (3-bit recoding) and *higher radices* : precomputing $3B, \dots \Rightarrow$ larger overhead

6.5 Wallace Tree Addition

- *Speed-up technique* : fast partial product addition

  $$A = O(n^2) \ , \ T = O(\log n)$$

- Applicable to *parallel multipliers* : parallel partial product generation (normal or Booth recoded)

- – *Irregular* adder tree (Wallace tree) due to different number of bits per column

  $\Rightarrow$ irregular *wiring* and/or *layout*
  $\Rightarrow$ non-uniform *bit arrival times* at final adder

6.6 Multiplier Implementations

- *Sequential* multipliers :
  ○ *low* performance, *small* area, *resource sharing* (adder)

- *Braun* or *Baugh-Wooley* multiplier (array multiplier) :
  ○ *medium* performance, *high* area, *high* regularity
  ○ *layout generators* $\Rightarrow$ data paths and macro-cells
  ○ simple *pipelining*, *faster* CPA $\Rightarrow$ higher speed

- *Booth-Wallace* multiplier (parallel multiplier) [9] :
  ○ *high* performance, *high* area, *low* regularity
  ○ *custom* multipliers, *netlist generators*
  ○ often *pipelined* (e.g. register between CSA-tree and CPA)

- *Signed-unsigned* multiplier : signed multiplier with operands extended by 1 bit ($a_n = a_{n-1}/0$, $b_n = b_{n-1}/0$)

6.7 Composition from Smaller Multipliers

- $(2n \times 2n)$-bit multiplier can be *composed* from 4 $(n \times n)$-bit multipliers (can be repeated recursively)

$$A \cdot B = (A_H 2^n + A_L) \cdot (B_H 2^n + B_L)$$
$$= A_H B_H 2^{2n} + (A_H B_L + A_L B_H) 2^n + A_L B_L$$

- 4 $(n \times n)$-bit multipliers + $(2n)$-bit CSA + $(3n)$-bit CPA

$$\boxed{A_H \cdot B_L}$$
$$\boxed{A_H \cdot B_H} \boxed{A_L \cdot B_L}$$
$$\boxed{A_L \cdot B_H}$$

- *less efficient* (area and speed)

6.8 Squaring

- $\boxed{P = A^2 = AA}$: multiplier *optimizations* possible

				$a_0 a_3$	$\mathbf{a_0 a_2}$	$a_0 a_1$	a_0
			$a_1 a_3$	$a_1 a_2$	a_1	$a_1 a_0$	
		$a_2 a_3$	a_2	$a_2 a_1$	$\mathbf{a_2 a_0}$		
+	a_3	$a_3 a_2$	$a_3 a_1$	$a_3 a_0$			
$\rightarrow$	$a_2 a_3$	$a_1 a_3$	$a_0 a_3$	$\mathbf{a_0 a_2}$	$a_0 a_1$		$a_0 a_0$
	$a_3 a_3$		$a_1 a_2$		$a_1 a_1$		
+			$a_2 a_2$				
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

- $+ (\lfloor n/2 \rfloor + 1)$ partial products (if no Booth recoding used) $\Rightarrow$ optimized squarer *more efficient* than multiplier

- *Table look-up* (ROM) less efficient for every n

---

7 Division / Square Root Extraction

7.1 Division Basics

$$\boxed{\frac{A}{B} = Q + \frac{R}{B}} \qquad \boxed{A = Q \cdot B + R \; ; \; R < B}$$
$$\boxed{R = A \text{ rem } B \;\; (\text{remainder})}$$

- $A \in [0, 2^{2n} - 1]$, $B, Q, R \in [0, 2^n - 1]$, $B \neq 0$
- $Q < 2^n \rightarrow A < 2^n B$, otherwise *overflow* $\Rightarrow$ *normalize* B before division ($B \in [2^{n-1}, 2^n - 1]$)

Algorithms (radix-2)

- *Subtract-and-shift* : partial remainders R_i [1, 2]
- *Sequential* algorithm : recursive, f non-associative

$$\boxed{\begin{array}{l} q_i = \left(R_{i+1} \geq 2^i B \right) , \; R_i = R_{i+1} - q_i 2^i B \\ R_n = A , \; R = R_0 \; ; \; i = n - 1, \ldots, 0 \;\; (\text{r.m.n.}) \end{array}}$$

Basic algorithm : *compare* and *conditionally subtract* $\Rightarrow$ expensive comparison and CPA

Restoring division : *subtract* and *conditionally restore* (adder or multiplexer) $\Rightarrow$ expensive CPA and restoring

Non-restoring division : *detect sign*, *subtract/add*, and *correct* by next steps $\Rightarrow$ expensive CPA

SRT division : *estimate range*, *subtract/add* (CSA), and *correct* by next steps $\Rightarrow$ inexpensive CSA

7.2 Restoring Division

$$q_i = \begin{cases} 1 & \text{if } R_{i+1} - B2^i \geq 0 \\ 0 & \text{if } R_{i+1} - B2^i < 0 \end{cases}$$

$$\begin{array}{c|l} i & R_{i+1} - B2^i < 0 : \quad q_i = 0 , \; R_i = \underline{R_{i+1}} \quad (\text{restored}) \\ i-1 & \underline{R_{i+1}} - B2^{i-1} \geq 0 : q_{i-1} = 1 , \; R_{i-1} = \underline{\underline{R_{i+1} - B2^{i-1}}} \end{array}$$

7.3 Non-Restoring Division

$$q'_i = \begin{cases} 1 & \text{if } R_{i+1} \geq 0 \\ -1 = \overline{1} & \text{if } R_{i+1} < 0 \end{cases}$$

$$\begin{array}{c|l} i & R_{i+1} \geq 0 : \quad q'_i = 1 , \; R_i = \underline{R_{i+1} - B2^i} \\ i-1 & \underline{R_{i+1} - B2^i} < 0 : q'_{i-1} = \overline{1} , \; R_{i-1} = \underline{R_{i+1} - B2^i} \\ & \qquad\qquad\qquad +B2^{i-1} = \underline{\underline{R_{i+1} - B2^{i-1}}} \end{array}$$

- *One* subtraction/addition (CPA) per step
- Final *correction step* for R (additional CPA)
- Simple quotient digit *conversion* : (note: q'_i irredundant)

$$\boxed{\begin{array}{c} q'_i \in \{\overline{1}, 1\} \; \rightarrow \; q_i \in \{0, 1\} \; : \; q_i = \frac{1}{2}(q'_i + 1) \\ Q = (\overline{q_{n-1}}, q_{n-2}, q_{n-3}, \ldots, q_0, 1) \end{array}}$$

$$\boxed{\begin{array}{l} A = (n+1) A_{CPA} \\ \quad = O(n^2) \text{ or } O(n^2 \log n) \\ T = (n+1) T_{CPA} \\ \quad = O(n^2) \text{ or } O(n \log n) \end{array}}$$

7.4 Signed Division

$$q'_i = \begin{cases} 1 & \text{if } R_{i+1}, B \text{ same sign} \\ \overline{1} & \text{if } R_{i+1}, B \text{ opposite sign} \end{cases}$$

- Example : signed non-restoring **array divider** (simplifications: $B > 0$, final correction of R omitted)

$$\boxed{A = 9n^2 , \; T = 2n^2 + 4n}$$

7.5 SRT Division (Sweeney, Robertson, Tocher)

$$q_i' = \begin{cases} 1 & \text{if} & B2^i \le R_{i+1} \\ 0 & \text{if} & -B2^i \le R_{i+1} < B2^i \\ \bar{1} & \text{if} & R_{i+1} < -B2^i \end{cases} , \quad q_i' \text{ is } SD \text{ number}$$

- If $2^{n-1} \le B < 2^n$, i.e. B is *normalized* :
  $$\Rightarrow -B2^i \le -2^{n+i-1} \le R_{i+1} < 2^{n+i-1} \le B2^i$$

  $$\Rightarrow q_i' = \begin{cases} 1 & \text{if} & 2^{n+i-1} \le R_{i+1} \\ 0 & \text{if} & -2^{n+i-1} \le R_{i+1} < 2^{n+i-1} \\ \bar{1} & \text{if} & R_{i+1} < -2^{n+i-1} \end{cases}$$

+ Only *3 MSB* are compared $\Rightarrow q_i'$ are *estimated* $\Rightarrow$ *CSA* instead of CPA can be used (precise enough) [19]
- *Correction* in following steps (+ final correction step)
- *Redundant* representation of q_i' (SD representation) $\Rightarrow$ *final conversion* necessary (CPA)
+ *Highly regular* and *fast* ($O(n)$) SRT array dividers $\Rightarrow$ only *slightly slower/larger* than array multipliers

$$\begin{aligned} A &= nA_{CSA} + 2A_{CPA} \\ &= O(n^2) \\ T &= nT_{CSA} + T_{CPA} \\ &= O(n) \end{aligned}$$
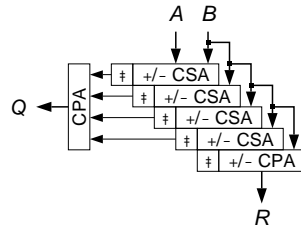
7.6 High-Radix Division

- Radix $\beta = 2^m$, $q_i' \in \{\overline{\beta-1}, \ldots, \bar{1}, 0, 1, \ldots, \beta-1\}$

- m *quotient bits* per step $\Rightarrow$ *fewer*, but *more complex* steps

+ Suitable for *SRT* algorithm $\Rightarrow$ *faster*

- *Complex* comparisons (more bits) and decisions $\Rightarrow$ table look-up ($\Rightarrow$ Pentium bug!)

7.7 Division by Multiplication

Division by convergence

$$Q = \frac{A}{B} = \frac{A \cdot R_0 R_1 \cdots R_{m-1}}{B \cdot R_0 R_1 \cdots R_{m-1}} \to \frac{A \cdot \frac{1}{B}}{B \cdot \frac{1}{B}} = \frac{Q}{1} \text{ resp. } \frac{Q}{2^n}$$

- $B_{i+1} = B_i \cdot R_i = \underbrace{2^n(1-y)}_{B_i} \cdot \underbrace{(1+y)}_{R_i} = \underbrace{2^n(1-y^2)}_{> B_i, \ \to 2^n}$,

  $y = 1 - B_i 2^{-n}$, $R_i = 2 - B_i 2^{-n} = \overline{B}_i + 1$ (signed)

- *Algorithm* :
  $$\begin{aligned} B_{i+1} &= B_i \cdot R_i , \ A_{i+1} = A_i \cdot R_i \\ R_i &= \overline{B}_i + 1 ; \ i = 0, \ldots, m-1 \\ A_0 &= A , \ B_0 = B , \ Q = A_m \quad \text{(r.s.n.)} \end{aligned}$$

- *Quadratic* convergence : $\boxed{L = \lceil \log n \rceil}$

Division by reciprocation

$$Q = \frac{A}{B} = A \cdot \frac{1}{B}$$

- *Newton-Raphson* iteration method :
  $$\text{find } f(X) = 0 \text{ by recursion } X_{i+1} = X_i - \frac{f(X_o)}{f'(X_i)}$$

- $f(X) = \frac{1}{X} - B$, $f'(X) = -\frac{1}{X^2}$, $f\left(\frac{1}{B}\right) = 0$

- *Algorithm* :
  $$\begin{aligned} X_{i+1} &= X_i \cdot (2 - B \cdot X_i) ; \ i = 0, \ldots, m-1 \\ X_0 &= B , \ Q = X_m \quad \text{(r.s.n.)} \end{aligned}$$

- *Quadratic* convergence : $\boxed{L = O(\log n)}$

- *Speed-up* : first approximation X_0 from table

7.8 Remainder / Modulus

Remainder (rem) : *signed* remainder of a division

$$R = A \text{ rem } B = A - \lfloor A/B \rfloor \cdot B , \ \text{sign}(R) = \text{sign}(A)$$

Modulus (mod) : *positive* remainder of a division

$$M = A \text{ mod } B , \ M \ge 0 , \ M = \begin{cases} R & \text{if } A \ge 0 \\ R + B & \text{else} \end{cases}$$

7.9 Divider Implementations

- *Iterative* dividers (through multiplication) :
  - *resource sharing* of existing components (multiplier)
  - *medium* performance, *medium* area
  - *high* efficiency if components are shared

- *Sequential* dividers (restoring, non-restoring, SRT) :
  - *resource sharing* of existing components (e.g. adder)
  - *low* performance, *low* area

- *Array* dividers (restoring, non-restoring, SRT) :
  - *dedicated* hardware component
  - *high* performance, *high* area
  - *high* regularity $\Rightarrow$ layout generators, pipelining
  - *square root extraction* possible by minor changes
  - *combination* with multiplication or/and square root

- No *parallel* dividers exist, as compared to parallel multipliers (sequential nature of division)

7.10 Square Root Extraction

$$\boxed{\sqrt{A-R} = Q} \qquad \boxed{A = Q^2 + R}$$

- $A \in [0, 2^{2n} - 1]$, $Q \in [0, 2^n - 1]$

Algorithm

- *Subtract-and-shift* : partial remainders R_i and quotients
  $Q_i = Q_{i+1} + q_i 2^i = (q_{n-1}, \ldots, q_i, 0, \ldots, 0)$ [1]
- $Q_i^2 = \left(Q_{i+1} + q_i 2^i\right)^2 = Q_{i+1}^2 + q_i \underline{2^i \left(2Q_{i+1} + q_i 2^i\right)}$

$$\boxed{\begin{array}{l} q_i = \left(R_{i+1} \geq 2^i \left(2Q_{i+1} + 2^i\right)\right) , \; Q_i = Q_{i+1} + q_i 2^i \\ R_i = R_{i+1} - q_i 2^i \left(2Q_{i+1} + q_i 2^i\right) \; ; \; i = n-1, \ldots, 0 \\ \quad R_n = A , \; Q_n = 0 , \; R = R_0 , \; Q = Q_0 \quad \text{(r.m.n.)} \end{array}}$$
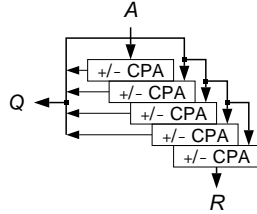
Implementation

+ Similar to *division* $\Rightarrow$ *same algorithms* applicable
  (restoring, non-restoring, SRT, high-radix)

+ *Combination* with division in same component possible

- Only *triangular array* required
  (step i : $q_{k \leq i} = 0$)

$$\boxed{\begin{array}{l} A \approx A_{DIV}/2 \\ T \approx T_{DIV} \end{array}}$$

---

8 Elementary Functions

- *Exponential* function : e^x ($\exp x$)
- *Logarithm* function : $\ln x$, $\log x$
- *Trigonometric* functions : $\sin x$, $\cos x$, $\tan x$
- *Inverse trig.* functions : $\arcsin x$, $\arccos x$, $\arctan x$
- *Hyperbolic* functions : $\sinh x$, $\cosh x$, $\tanh x$

8.1 Algorithms

- *Table look-up* : inefficient for large word lengths [5]
- *Taylor series expansion* : complex implementation
- *Polynomial* and *rational approximations* [1, 5]
- *Shift-and-add* algorithms [5]
- *Convergence algorithms* [1, 2] :
  ○ similar to *division-by-convergence*
  ○ two (or more) *recursive formulas* : one formula
  converges to a constant, the other to the result
- *Coordinate rotation* (CORDIC) [2, 5, 20] :
  ○ *3 equations* for x-, y-coordinate, and angle
  ○ computes *all elementary functions* by proper input
  settings and choice of modes and outputs
  ○ simple, *universal* hardware, small look-up table

8.2 Integer Exponentiation

- *Approximated* exponentiation : $\boxed{x^y = e^{y \ln x} = 2^{y \log x}}$

- *Base-2 integer* exponentiation : $\boxed{2^A = (\ldots, 0, \underbrace{1}_{A}, 0, \ldots)}$

- *Integer* exponentiation (exact) :

$$\boxed{A^B = \underbrace{A \cdot A \cdots A}_{B\times}} \qquad \boxed{L = 0 \cdots 2^n - 1 \; (!)}$$

Applications : modular exponentiation $A^B \pmod{C}$
in *cryptographic* algorithms (e.g. IDEA, RSA)

Algorithms : *square-and-multiply*

a) $E = A^B = A^{b_{n-1}2^{n-1} + \cdots + b_1 2 + b_0}$
$\qquad = A^{2^{n-1}b_{n-1}} \cdot A^{2^{n-2}b_{n-2}} \cdot \cdots A^{4b_2} \cdot \underline{A^{2b_1}} \cdot A^{b_0}$

$$\boxed{\begin{array}{l} E_i = P_i^{b_i} \cdot E_{i-1} , \; P_{i+1} = P_i^2 \; ; \; i = 0, \ldots, n-1 \\ \quad E_{-1} = 1 , \; P_0 = A , \; E = E_{n-1} \quad \text{(r.s.n.)} \end{array}}$$

$$\boxed{\begin{array}{l} A = 2A_{MUL} , \; T = T_{MUL} , \; L = n \quad \text{or} \\ A = A_{MUL} , \; T = T_{MUL} , \; L = 2n \end{array}}$$

---

b) $E = A^B = A^{b_{n-1}2^{n-1} + \cdots + b_1 2 + b_0}$
$\qquad = \left(\cdots \left(\left(A^{b_{n-1}}\right)^2 \cdot A^{b_{n-2}}\right)^2 \cdots A^{b_1}\right)^2 \cdot A^{b_0}$

$$\boxed{\begin{array}{l} E_i = E_{i+1}^2 \cdot A^{b_i} \; ; \; i = n-1, \ldots, 0 \\ \quad E_n = 1 , \; E = E_0 \quad \text{(r.s.n.)} \end{array}}$$

$$\boxed{A = A_{MUL} , \; T = T_{MUL} , \; L = 2(n-1)}$$

8.3 Integer Logarithm

$$\boxed{Z = \lfloor \log_2 A \rfloor}$$

- For detection/comparison of *order of magnitude*

- Corresponds to *leading-zeroes detection* (LZD) with
  encoded output

9 VLSI Design Aspects

9.1 Design Levels

Transistor-level design

- Circuit and layout designed *by hand* (full custom)

- *Low* design efficiency

- *High* circuit performance : high speed, low area

- *High* flexibility : choice of architecture and logic style

- Transistor-level circuit *optimizations* :

  ○ *logic style* : static vs. dynamic logic,
    complementary CMOS vs. pass-transistor logic

  ○ special *arithmetic* circuits : better than with gates

carry chain :

full-adder :
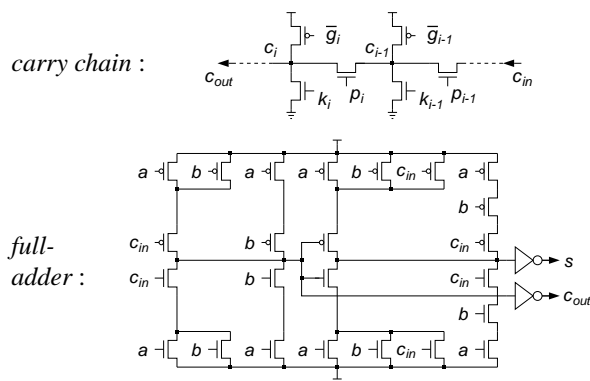
Gate-level design

- *Cell-based* design techniques : standard-cells, gate-array/
  sea-of-gates, field-programmable gate-array (FPGA)

- Circuit implemented by *hand* or by *synthesis* (library)

- Layout implemented by automated *place-and-route*

- *Medium* to *high* design efficiency

- *Medium* to *low* circuit performance

- *Medium* to *low* flexibility : full choice of architecture

Block-level design

- Layout blocks and netlists from parameterized automatic
  generators or *compilers* (library)

- *High* design efficiency

- *Medium* to *high* circuit performance

- *Low* flexibility : limited choice of architectures

- *Implementations* :

  data-path : *bit-sliced*, *bus-oriented* layout (array of
    cells: n bits × m operations), implementation of entire
    data paths, *medium* performance, *medium* diversity

  macro-cells : *tiled* layout, *fixed/single*-operation
    components, *high* performance, *small* diversity

  portable netlists : ⇒ gate-level design

9.2 Synthesis

High-level synthesis

- Synthesis from *abstract*, *behavioral* hardware description
  (e.g. data dependency graphs) using e.g. VHDL

- Involves *architectural synthesis* and *arithmetic
  transformations*

- High-level synthesis is still in the *beginnings*

Low-level synthesis

- *Layout* and *netlist generators*

- Included in *libraries* and *synthesis tools*

- Low-level synthesis is *state-of-the-art*

- *Basis* for efficient ASIC design

- Limited *diversity* and *flexibility* of library components

Circuit optimization

- Efficient optimization of *random logic* is state-of-the-art

- Optimization of entire *arithmetic circuits* is **not** feasible
  ⇒ only local optimizations possible

- Logic optimization *cannot* replace the synthesis of
  efficient arithmetic circuit structures using *generators*

9.3 VHDL

Arithmetic types : `unsigned`, `signed` (2's complement)

Arithmetic packages

- `numeric_bit`, `numeric_std` (IEEE standard 1076.3),
  `std_logic_arith` (Synopsys)

- contain overloaded *arithmetic operators* and *resizing* /
  type conversion routines for `unsigned`, `signed` types

Arithmetic operators (VHDL'87/93) [21]

$$\begin{aligned}
relational &: \ =, /=, <, <=, >, >= \\
shift, rotate \ (\text{'93 only}) &: \ \texttt{rol}, \texttt{ror}, \texttt{sla}, \texttt{sll}, \texttt{sra}, \texttt{srl} \\
adding &: \ +, - \\
sign \ (unary) &: \ +, - \\
multiplying &: \ \texttt{*}, /, \texttt{mod}, \texttt{rem} \\
exponent, absolute &: \ \texttt{**}, \texttt{abs}
\end{aligned}$$

Synthesis

- Typical *limitations* of synthesis tools :

  `/`, `mod`, `rem` : both operands must be constant or divisor
    must be a power of two

  `**` : for power-of-two bases only

- Variety of arithmetic components provided in *separate
  libraries* (e.g. DesignWare by Synopsys)

Resource sharing

- *Sharing* one resource for multiple operations
- Done *automatically* by some synthesis tools
- Otherwise, appropriate *coding* is necessary :

  a) `S <= A + C when SELA = '1' else B + C;`
                  $\Rightarrow$ 2 adders + 1 multiplexer
  b) `T <= A when SELA = '1' else B;`
      `S <= T + C;` $\Rightarrow$ 1 multiplexer + 1 adder

Coding & synthesis hints

- *Addition* : single adder with carry-in/carry-out :

  ```
  Aext <= resize(A, width+1) & Cin;
  Bext <= resize(B, width+1) & '1';
  Sext <= Aext + Bext;
  S    <= Sext(width downto 1);
  Cout <= Sext(width+1);
  ```

- *Synthesis* : check synthesis result for *allocated arithmetic units* $\Rightarrow$ code sanity check, control of circuit size

VHDL library of arithmetic units

- *Structural, synthesizable VHDL code* for most circuits described in this text is found in [22]

9.4 Performance

Pipelining

- *Pipelining* is basically possible with every combinational circuit $\Rightarrow$ *higher throughput*
- Arithmetic circuits are *well suited* for pipelining due to high regularity
- Pipelining of arithmetic circuits can be *very costly* :
  - large amount of *internal* signals in arithmetic circuits
  - *array structures* : many small pipeline registers
  - *tree structures* : few large pipeline registers
  - $\Rightarrow$ *no advantage* of tree structures anymore (except for smaller latency)
- *Fine-grain* pipelining $\Rightarrow$ *systolic arrays* (often applied to arithmetic circuits)

High speed

- Fast circuit *architectures*, *pipelining*, *replication* (parallelization), and combinations of those
- Optimal solution depends on arithmetic *operation*, circuit *architecture*, user *specifications*, and circuit *environment*

Low power

Power-related properties of arithmetic circuits :

- High *glitching activity* due to high bit dependencies and large logic depth

Power reduction in arithmetic circuits [23] :

- Reduce the *switched capacitance* by choosing an *area efficient* circuit architecture
- Allow for *lower supply voltage* by *speeding up* the circuitry
- Reduce the *transition activity* :
  - apply *stable inputs* while circuit is not in use ($\Rightarrow$ *disabling* subcircuits)
  - reduce *glitching transitions* by *balancing* signal paths (partly done by speed-up techniques, otherwise difficult to realize)
  - reduce *glitching transitions* by reducing *logic depth* (pipelining)
  - take advantage of *correlated data* streams
  - choose appropriate number representations (e.g. *Gray codes* for counters)

9.5 Testability

Testability goal : high *fault coverage* with few *test vectors* that are easy to generate/apply

Random test vectors : *easy* to generate and apply/propagate, few vectors give *high* (but not perfect) fault coverage for *most* arithmetic circuits

Special test vectors : sometimes *hard* to generate and apply, required for coverage of *hard-detectable* faults which are inherent in most arithmetic circuits

Hard-detectable faults found in :

- circuits of arithmetic operations with inherent *special cases* (arithmetic exceptions) : detectors, comparators, incrementers and counters (MSBs), adder flags
- circuits using *redundant number representations* ($\neq$ redundant hardware) : dividers (Pentium bug!)

Bibliography

[1] I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

[2] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, 1979.

[3] O. Spaniol, *Computer Arithmetic*, John Wiley & Sons, 1981.

[4] J. J. F. Cavanagh, *Digital Computer Arithmetic: Design and Implementation*, McGraw-Hill, 1984.

[5] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Boston, 1997.

[6] *Proceedings of the Xth Symposium on Computer Arithmetic*.

[7] *IEEE Transactions on Computers*.

[8] D. R. Lutz and D. N. Jayasimha, "Programmable modulo-k counters", *IEEE Trans. Circuits and Syst.*, vol. 43, no. 11, pp. 939–941, Nov. 1996.

[9] H. Makino *et al.*, "An 8.8-ns 54 × 54-bit multiplier with high speed redundant binary architecture", *IEEE J. Solid-State Circuits*, vol. 31, no. 6, pp. 773–783, June 1996.

[10] W. N. Holmes, "Composite arithmetic: Proposal for a new standard", *IEEE Computer*, vol. 30, no. 3, pp. 65–73, Mar. 1997.

[11] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*, PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.

[12] A. Tyagi, "A reduced-area scheme for carry-select adders", *IEEE Trans. Comput.*, vol. 42, no. 10, pp. 1162–1170, Oct. 1993.

[13] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders", in *Proc. 8th Computer Arithmetic Symp.*, Como, May 1987, pp. 49–56.

[14] D. W. Dobberpuhl *et al.*, "A 200-MHz 64-b dual-issue CMOS microprocessor", *IEEE J. Solid-State Circuits*, vol. 27, no. 11, pp. 1555–1564, Nov. 1992.

[15] A. De Gloria and M. Olivieri, "Statistical carry lookahead adders", *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 340–347, Mar. 1996.

[16] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach", *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 294–305, Mar. 1996.

[17] Z. Wang, G. A. Jullien, and W. C. Miller, "A new design technique for column compression multipliers", *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 962–970, Aug. 1995.

[18] J. Cortadella and J. M. Llaberia, "Evaluation of A + B = K conditions without carry propagation", *IEEE Trans. Comput.*, vol. 41, no. 11, pp. 1484–1488, Nov. 1992.

[19] S. E. McQuillan and J. V. McCanny, "Fast VLSI algorithms for division and square root", *J. VLSI Signal Processing*, vol. 8, pp. 151–168, Oct. 1994.

[20] Y. H. Hu, "CORDIC-based VLSI architectures for digital signal processing", *IEEE Signal Processing Magazine*, vol. 9, no. 3, pp. 16–35, July 1992.

[21] K. C. Chang, *Digital Design and Modeling with VHDL and Synthesis*, IEEE Computer Society Press, Los Alamitos, California, 1997.

[22] R. Zimmermann, "*VHDL Library of Arithmetic Units*", http://www.iis.ee.ethz.ch/~zimmi/arith_lib.html.

[23] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*, Kluwer, Norwell, MA, 1995.