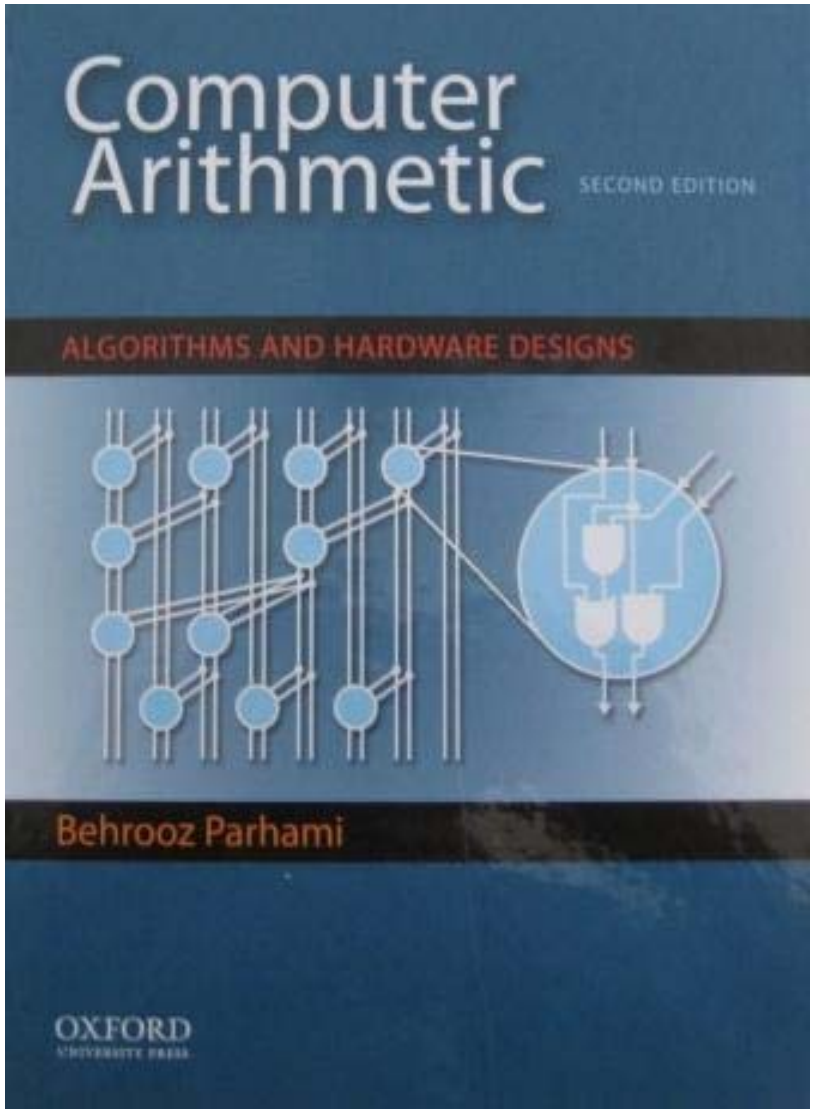


Part VII
Implementation Topics

	Parts	Chapters
Elementary Operations	I. Number Representation	1. Numbers and Arithmetic 2. Representing Signed Numbers 3. Redundant Number Systems 4. Residue Number Systems
	II. Addition / Subtraction	5. Basic Addition and Counting 6. Carry-Lookahead Adders 7. Variations in Fast Adders 8. Multioperand Addition
	III. Multiplication	9. Basic Multiplication Schemes 10. High-Radix Multipliers 11. Tree and Array Multipliers 12. Variations in Multipliers
	IV. Division	13. Basic Division Schemes 14. High-Radix Dividers 15. Variations in Dividers 16. Division by Convergence
	V. Real Arithmetic	17. Floating-Point Reperesentations 18. Floating-Point Operations 19. Errors and Error Control 20. Precise and Certifiable Arithmetic
	VI. Function Evaluation	21. Square-Rooting Methods 22. The CORDIC Algorithms 23. Variations in Function Evaluation 24. Arithmetic by Table Lookup
	VII. Implementation Topics	25. High-Throughput Arithmetic 26. Low-Power Arithmetic 27. Fault-Tolerant Arithmetic 28. Reconfigurable Arithmetic

Appendix: Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Computer Arithmetic: Algorithms and Hardware Designs* (Oxford U. Press, 2nd ed., 2010, ISBN 978-0-19-532848-6). It is updated regularly by the author as part of his teaching of the graduate course ECE 252B, Computer Arithmetic, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Unauthorized uses are strictly prohibited. © Behrooz Parhami

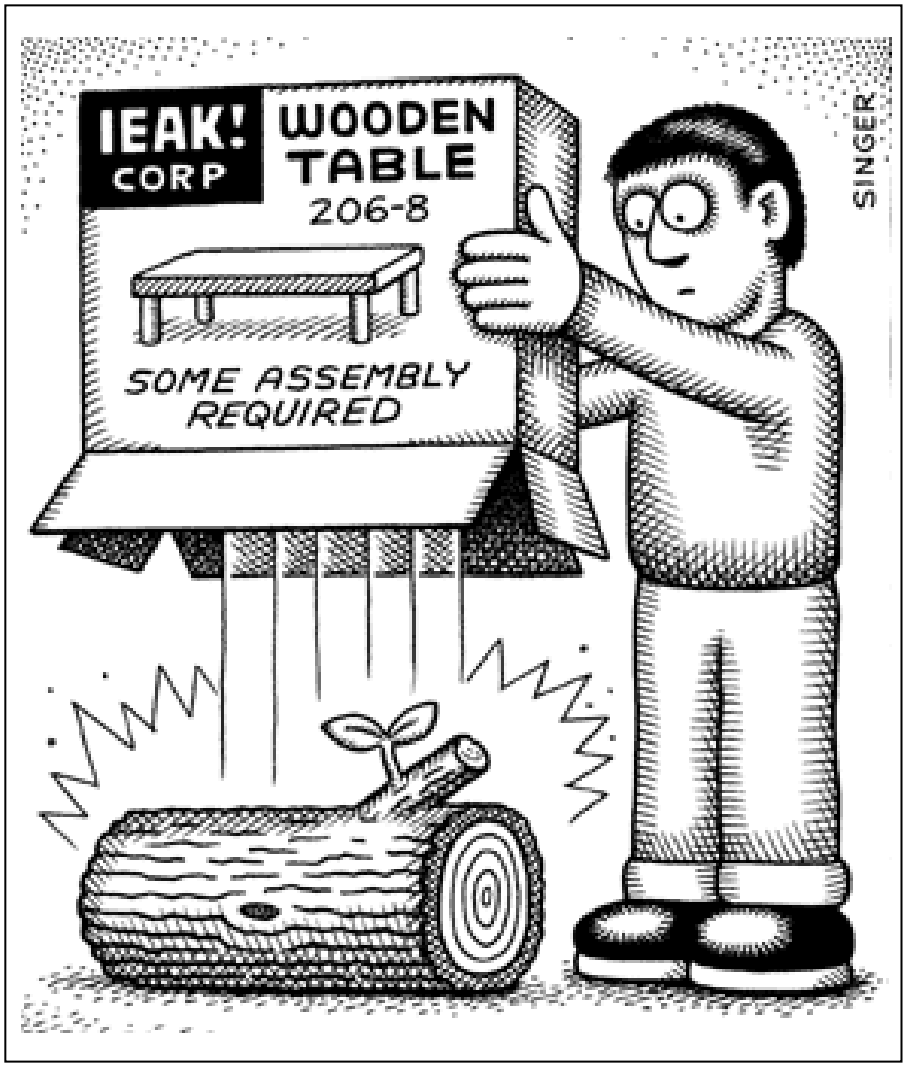
Edition	Released	Revised	Revised	Revised	Revised
First	Jan. 2000	Sep. 2001	Sep. 2003	Oct. 2005	Dec. 2007
Second	May 2010				

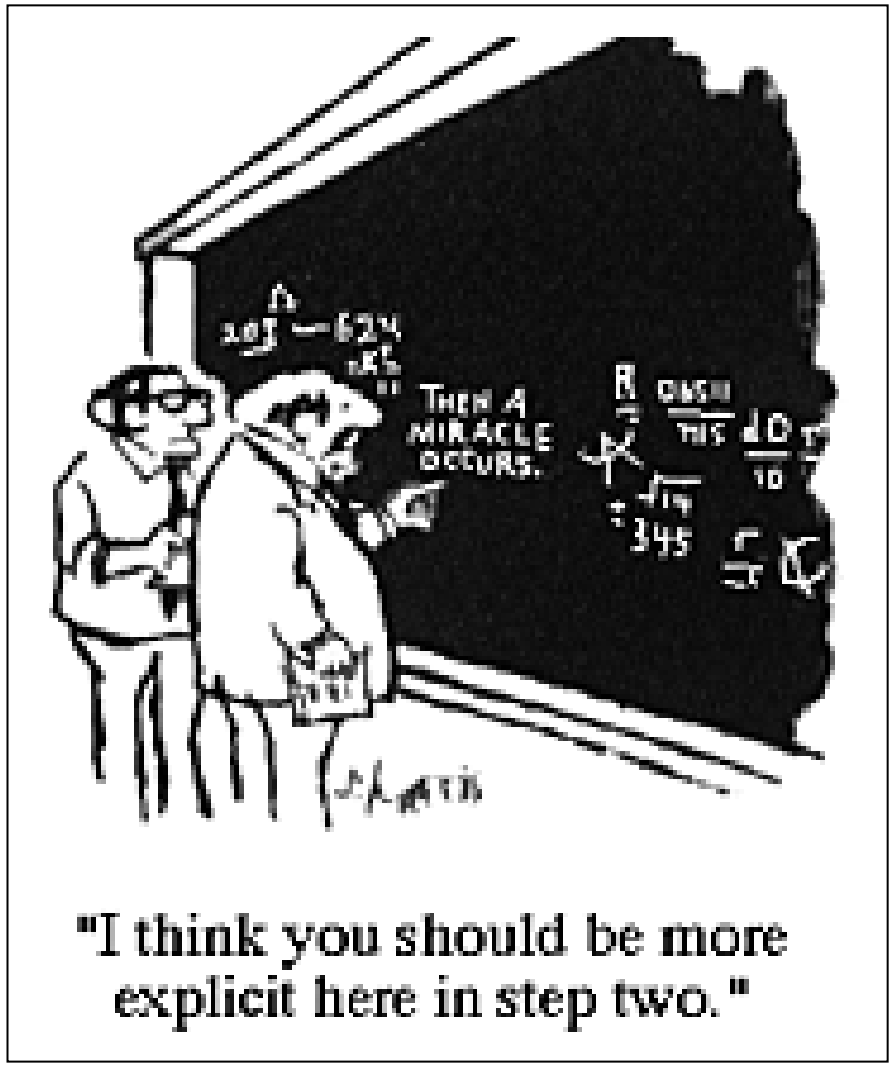
VII Implementation Topics

Sample advanced implementation methods and tradeoffs

- Speed/latency is seldom the only concern
- We also care about throughput, size, power/energy
- Fault-induced errors are different from arithmetic errors
- Implementation on Programmable logic devices

Topics in This Part	
Chapter 25	High-Throughput Arithmetic
Chapter 26	Low-Power Arithmetic
Chapter 27	Fault-Tolerant Arithmetic
Chapter 28	Reconfigurable Arithmetic





"I think you should be more explicit here in step two."

25 High-Throughput Arithmetic

Chapter Goals

Learn how to improve the performance of an arithmetic unit via higher throughput rather than reduced latency

Chapter Highlights

To improve overall performance, one must
- Look beyond individual operations
- Trade off latency for throughput

For example, a multiply may take 20 cycles, but a new one can begin every cycle

Data availability and hazards limit the depth

High-Throughput Arithmetic: Topics

Topics in This Chapter
25.1 Pipelining of Arithmetic Functions
25.2 Clock Rate and Throughput
25.3 The Earle Latch
25.4 Parallel and Digit-Serial Pipelines
25.5 On-Line of Digit-Pipelined Arithmetic
25.6 Systolic Arithmetic Units

25.1 Pipelining of Arithmetic Functions

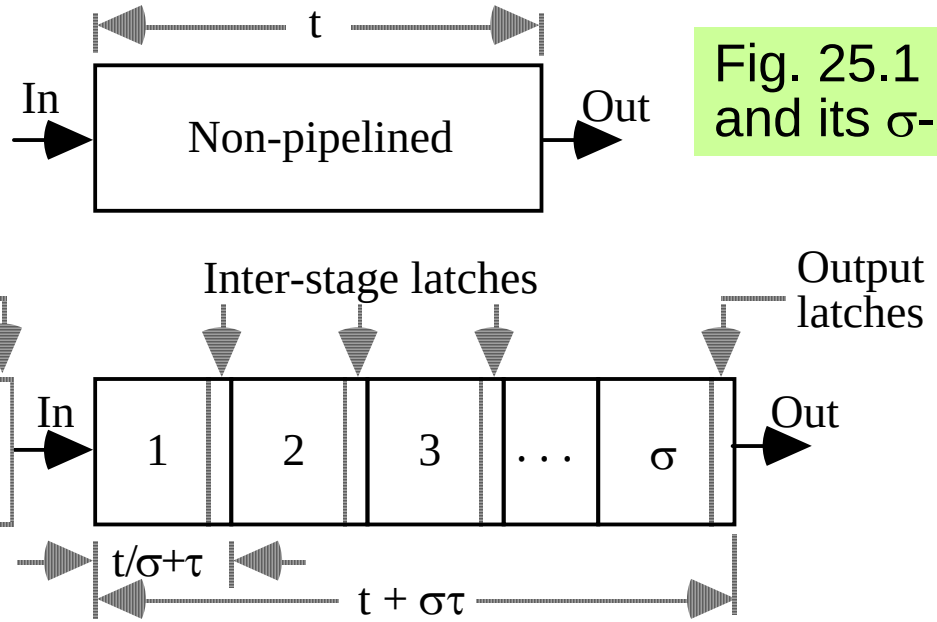


Fig. 25.1 An arithmetic function unit and its σ-stage pipelined version.

Throughput
Operations per unit time

Pipelining period
Interval between applying successive inputs

Latency, though a secondary consideration, is still important because:

a. Occasional need for doing single operations

b. Dependencies may lead to *bubbles* or even *drainage*

At times, pipelined implementation may improve the latency of a multistep computation and also reduce its cost; in this case, advantage is obvious

Analysis of Pipelining Throughput

Consider a circuit with cost (gate count) g and latency t

Simplifying assumptions for our analysis:

1. Time overhead per stage is τ (latching delay)
2. Cost overhead per stage is γ (latching cost)
3. Function is divisible into σ equal stages for any σ

Then, for the pipelined implementation:

Latency $\quad T = t + \sigma\tau$

Throughput $\quad R = \dfrac{1}{T/\sigma} = \dfrac{1}{t/\sigma + \tau}$

Cost $\quad G = g + \sigma\gamma$

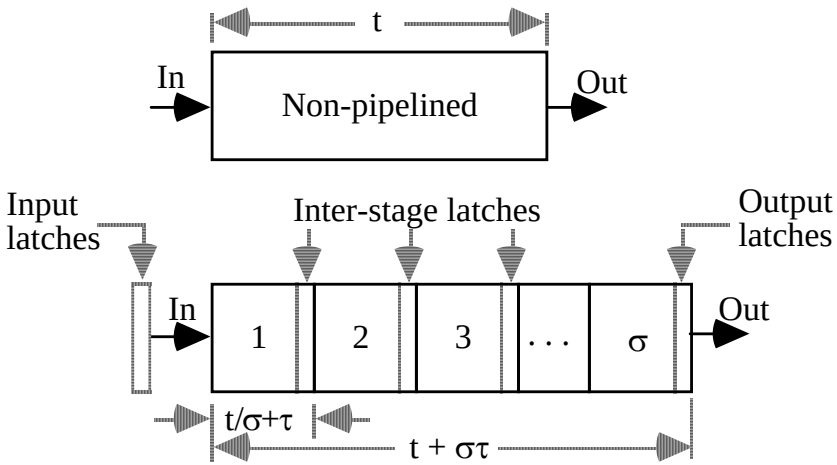


Fig. 25.1

Throughput approaches its maximum of $1/\tau$ for large σ

Analysis of Pipelining Cost-Effectiveness

$T = t + \sigma\tau$ Latency

$R = \frac{1}{T/\sigma} = \frac{1}{t/\sigma + \tau}$ Throughput

$G = g + \sigma\gamma$ Cost

Consider cost-effectiveness to be throughput per unit cost

$E = R/G = \sigma/[(t + \sigma\tau)(g + \sigma\gamma)]$

To maximize E, compute $dE/d\sigma$ and equate the numerator with 0

$tg - \sigma^2\tau\gamma = 0 \quad \Rightarrow \quad \sigma^{opt} = \sqrt{tg/(\tau\gamma)}$

We see that the most cost-effective number of pipeline stages is:

- Directly related to the latency and cost of the function;
  it pays to have many stages if the function is very slow or complex
- Inversely related to pipelining delay and cost overheads;
  few stages are in order if pipelining overheads are fairly high

All in all, not a surprising result!

25.2 Clock Rate and Throughput

Consider a σ-stage pipeline with stage delay t_{stage}

One set of inputs is applied to the pipeline at time t_1

At time $t_1 + t_{stage} + \tau$, partial results are safely stored in latches

Apply the next set of inputs at time t_2 satisfying $t_2 \geq t_1 + t_{stage} + \tau$

Therefore:

Clock period $= \Delta t = t_2 - t_1 \geq t_{stage} + \tau$

Throughput $= 1/$ Clock period $\leq 1/(t_{stage} + \tau)$

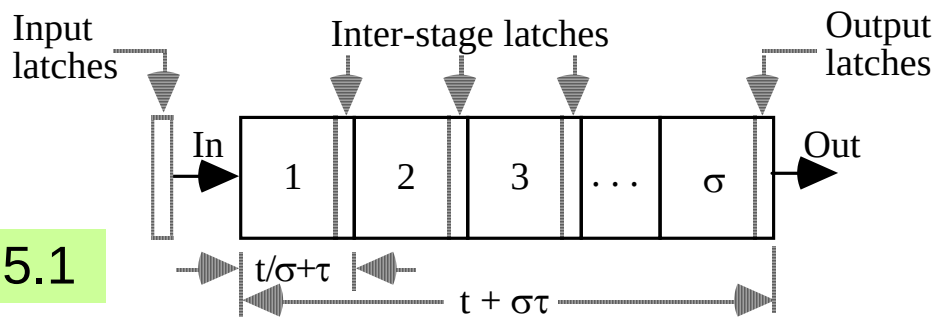


Fig. 25.1

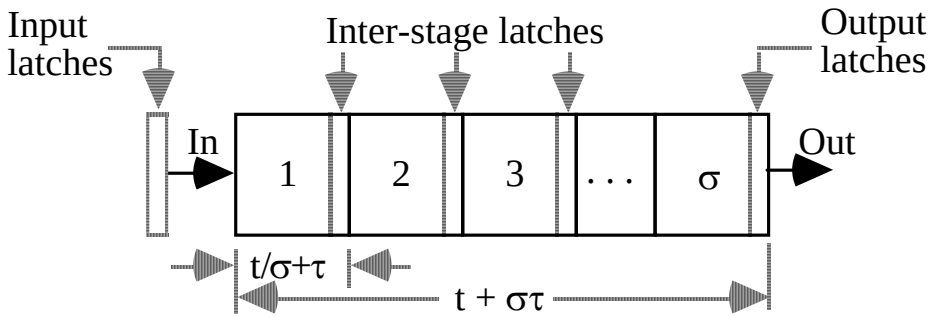
The Effect of Clock Skew on Pipeline Throughput

Two implicit assumptions in deriving the throughput equation below:

One clock signal is distributed to all circuit elements
All latches are clocked at precisely the same time

Throughput = 1/ Clock period
$\leq 1/(t_{stage} + \tau)$

Fig. 25.1



Uncontrolled or random *clock skew* causes the clock signal to arrive at point B before/after its arrival at point A

With proper design, we can place a bound $\pm\varepsilon$ on the uncontrolled clock skew at the input and output latches of a pipeline stage

Then, the clock period is lower bounded as:

Clock period $= \Delta t = t_2 - t_1 \geq t_{stage} + \tau + 2\varepsilon$

Wave Pipelining: The Idea

The stage delay t_{stage} is really not a constant but varies from t_{min} to t_{max}

- t_{min} represents fast paths (with fewer or faster gates)
- t_{max} represents slow paths

Suppose that one set of inputs is applied at time t_1

At time $t_1 + t_{max} + \tau$, the results are safely stored in latches

If that the next inputs are applied at time t_2, we must have:

$$t_2 + t_{min} \geq t_1 + t_{max} + \tau$$

This places a lower bound on the clock period:

$$\text{Clock period} = \Delta t = t_2 - t_1 \geq t_{max} - t_{min} + \tau$$

Two roads to higher pipeline throughput:
Reducing t_{max}
Increasing t_{min}

Thus, we can approach the maximum possible pipeline throughput of $1/\tau$ without necessarily requiring very small stage delay

All we need is a very small delay variance $t_{max} - t_{min}$

Visualizing Wave Pipelining

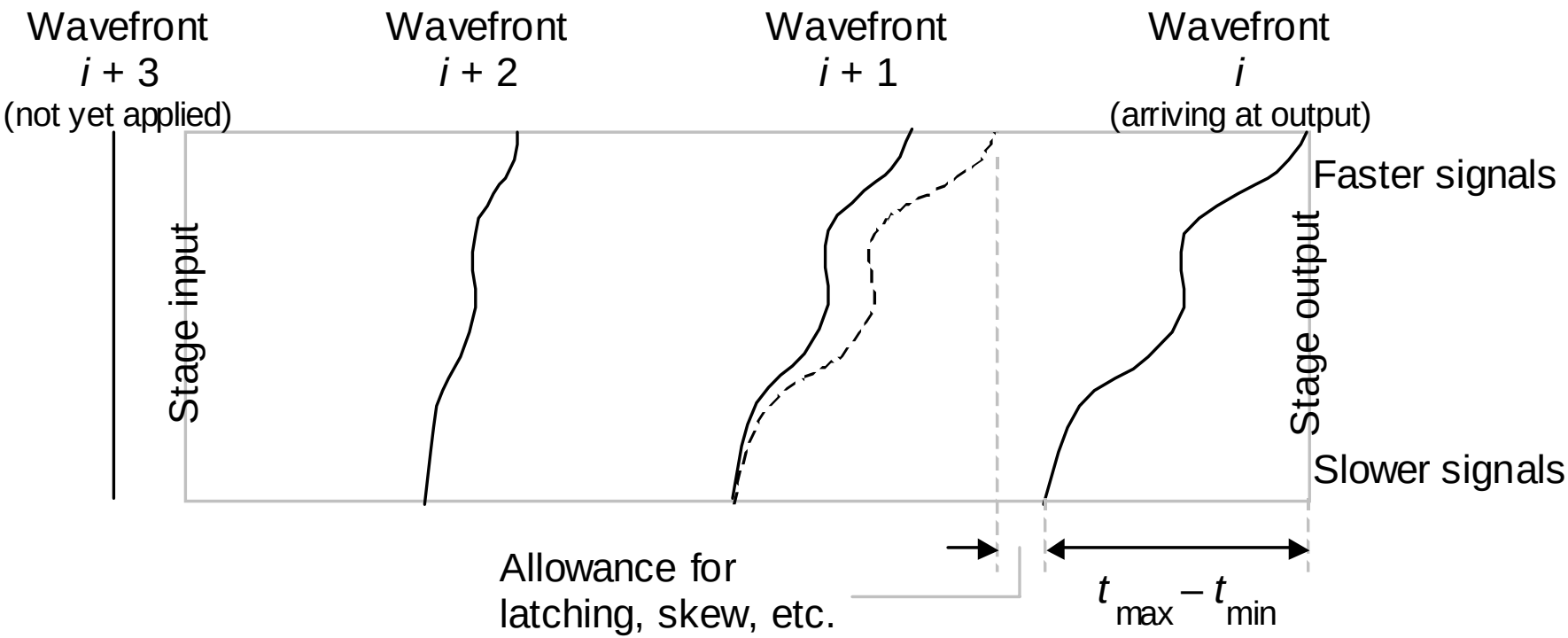
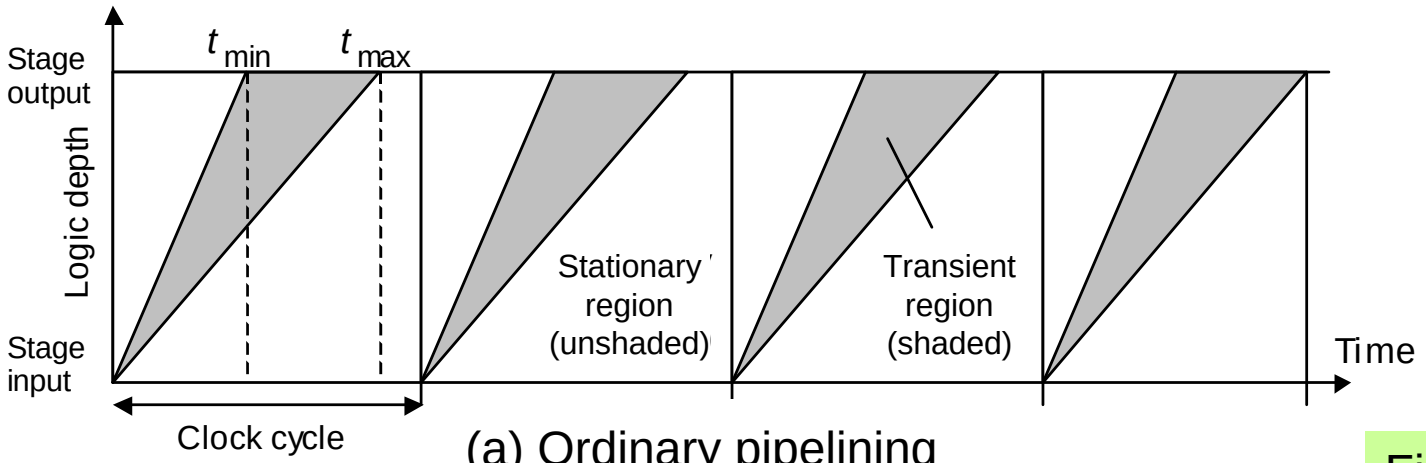
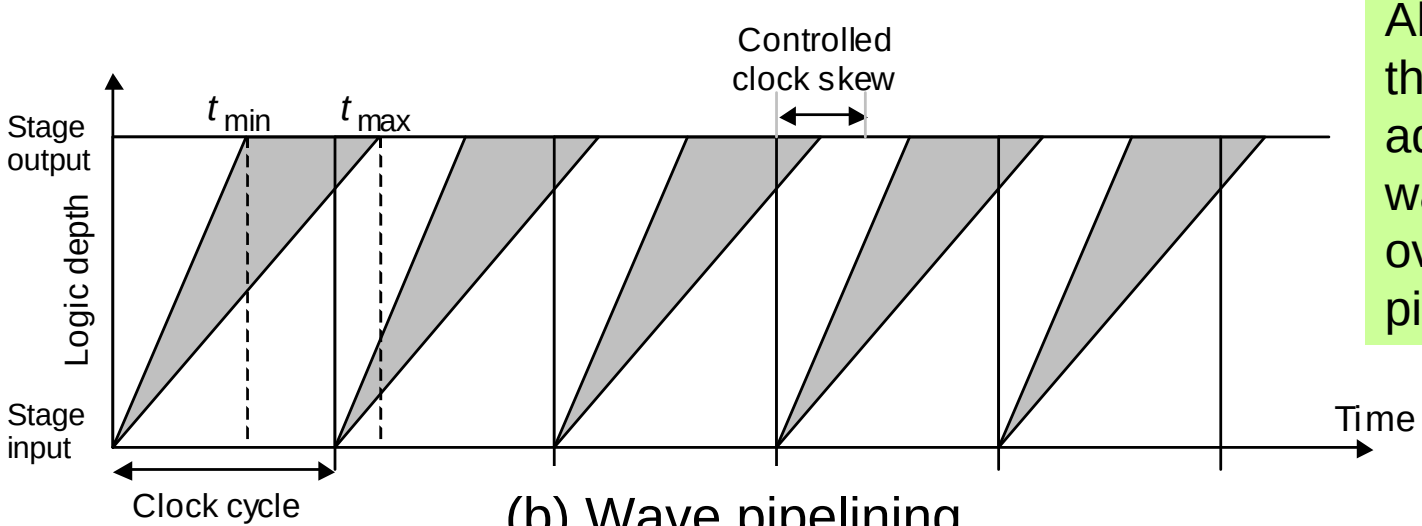


Fig. 25.2 Wave pipelining allows multiple computational wavefronts to coexist in a single pipeline stage .

Another Visualization of Wave Pipelining



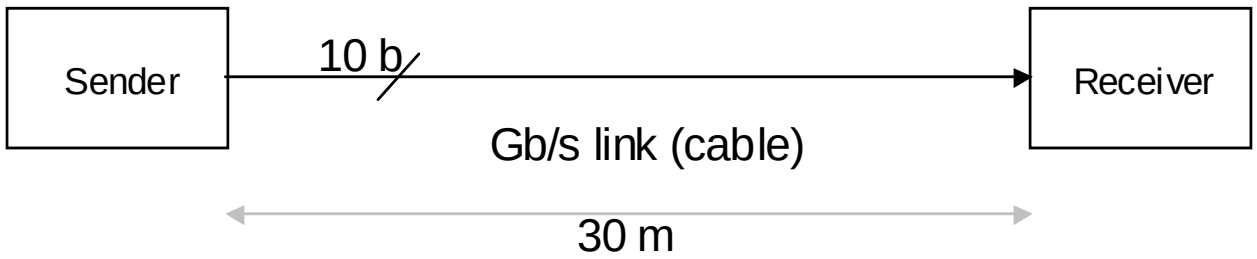
(a) Ordinary pipelining



(b) Wave pipelining

Fig. 25.3 Alternate view of the throughput advantage of wave pipelining over ordinary pipelining.

Difficulties in Applying Wave Pipelining



LAN and other high-speed links (figures rounded from Myrinet data [Bode95])

Gb/s throughput $\rightarrow$ Clock rate = 10^8 $\rightarrow$ Clock cycle = 10 ns
In 10 ns, signals travel 1-1.5 m (speed of light = 0.3 m/ns)
For a 30 m cable, 20-30 characters will be in flight at the same time

At the circuit and logic level (μm-mm distances, not m), there are still problems to be worked out

For example, delay equalization to reduce $t_{max} - t_{min}$ is nearly impossible in CMOS technology:

CMOS 2-input NAND delay varies by factor of 2 based on inputs
Biased CMOS (pseudo-CMOS) fairs better, but has power penalty

Controlled Clock Skew in Wave Pipelining

With wave pipelining, a new input enters the pipeline stage every Δt time units and the stage latency is $t_{max} + \tau$

Thus, for proper sampling of the results, clock application at the output latch must be skewed by $(t_{max} + \tau) \bmod \Delta t$

Example: $t_{max} + \tau = 12$ ns; $\Delta t = 5$ ns

A clock skew of +2 ns is required at the stage output latches relative to the input latches

In general, the value of $t_{max} - t_{min} > 0$ may be different for each stage

$$\Delta t \geq \max_{i=1 \text{ to } \sigma} [t_{max}^{(i)} - t_{min}^{(i)} + \tau]$$

The controlled clock skew at the output of stage i needs to be:

$$S^{(i)} = \sum_{j=1 \text{ to } i} [t_{max}^{(i)} - t_{min}^{(i)} + \tau] \bmod \Delta t$$

Random Clock Skew in Wave Pipelining

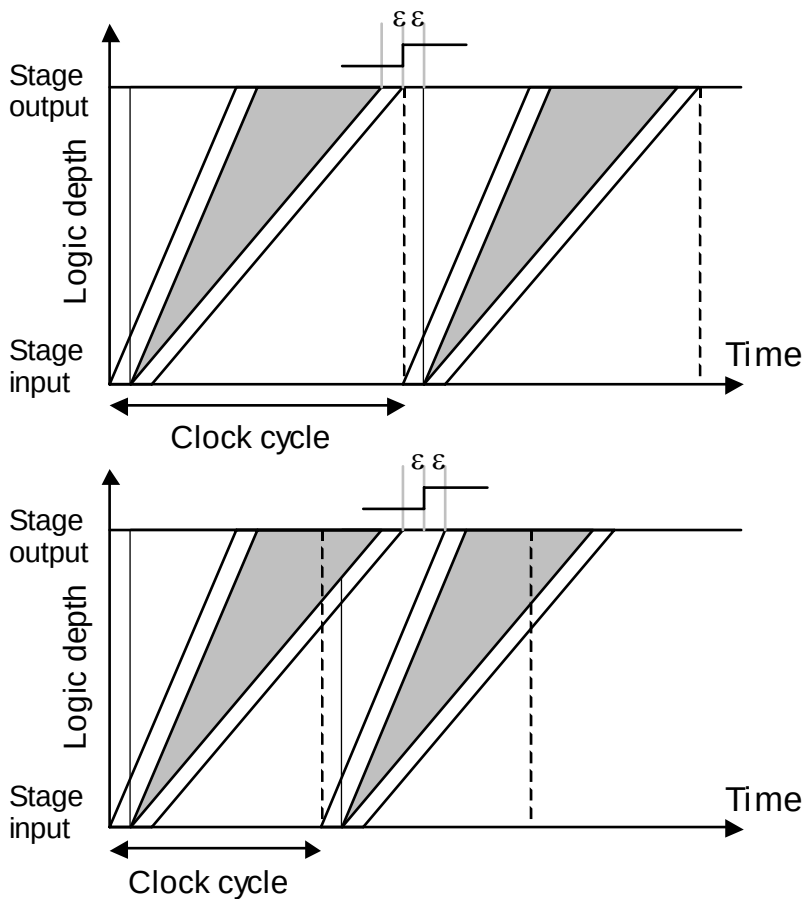
Clock period $= \Delta t = t_2 - t_1$
$\geq t_{max} - t_{min} + \tau + 4\varepsilon$

Reasons for the term 4ε:

Clocking of the first input set may lag by ε, while that of the second set leads by ε (net difference = 2ε)

The reverse condition may exist at the output side

Uncontrolled skew has a larger effect on wave pipelining than on standard pipelining, especially when viewed in relative terms



Graphical justification of the term 4ε

25.3 The Earle Latch

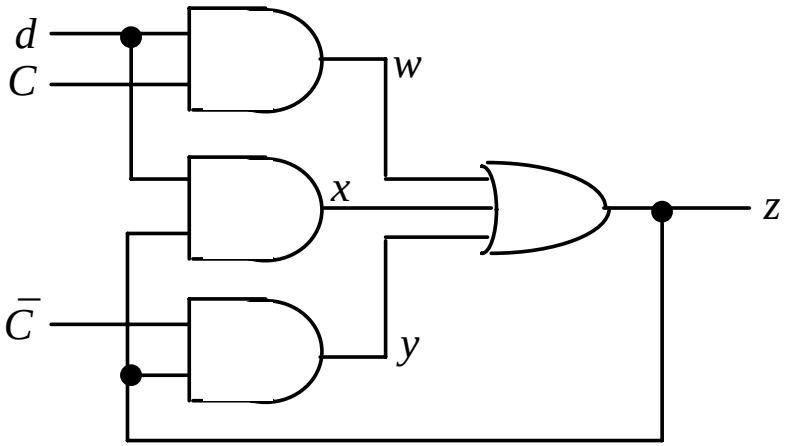


Fig. 25.4 Two-level AND-OR realization of the Earle latch.

Earle latch can be merged with a preceding 2-level AND-OR logic

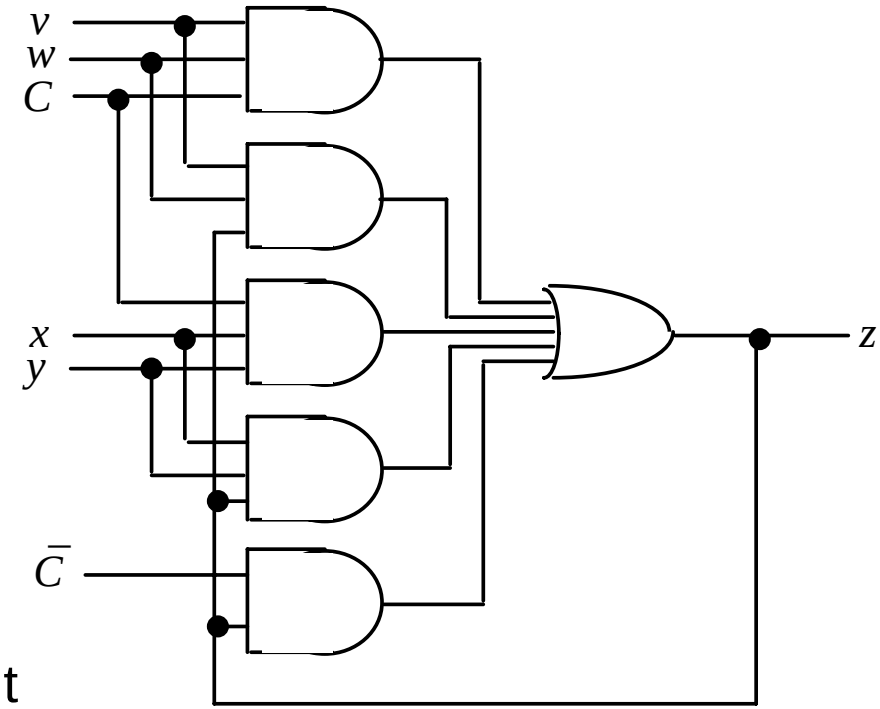


Fig. 25.5 Two-level AND-OR latched realization of the function $z = vw + xy$.

Example: To latch $d = vw + xy$, substitute for d in the latch equation

$$z = dC + dz + \bar{C}z$$

to get a combined "logic + latch" circuit implementing $z = vw + xy$

$$z = (vw + xy)C + (vw + xy)z + \bar{C}z$$
$$= vwC + xyC + vwz + xyz + \bar{C}z$$

Clocking Considerations for Earle Latches

We derived constraints on the maximum clock rate $1/\Delta t$

Clock period Δt has two parts: clock high, and clock low

$$\Delta t = C_{high} + C_{low}$$

Consider a pipeline stage between Earle latches

C_{high} must satisfy the inequalities

$$3\delta_{max} - \delta_{min} + S_{max}(C\uparrow, \overline{C}\downarrow) \leq C_{high} \leq 2\delta_{min} + t_{min}$$

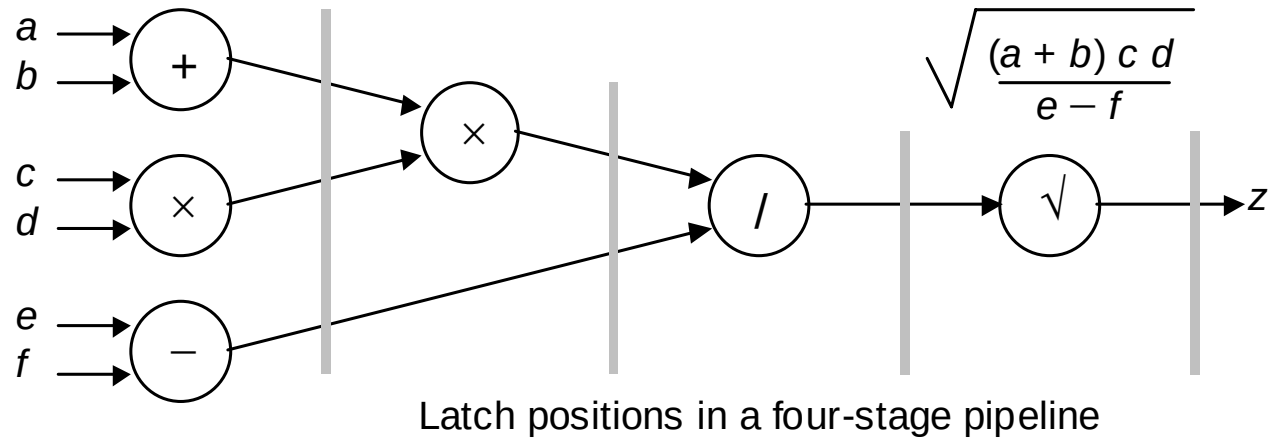
The clock pulse must be wide enough to ensure that valid data is stored in the output latch and to avoid logic hazard should C ⌐¬ slightly lead $\overline{C}$ ¬⌐

Clock must go low before the fastest signals from the next input data set can affect the input z of the latch

δ_{max} and δ_{min} are maximum and minimum gate delays

$S_{max}(C\uparrow, \overline{C}\downarrow) \geq 0$ is the maximum skew between $C\uparrow$ and $\overline{C}\downarrow$

25.4 Parallel and Digit-Serial Pipelines



Latch positions in a four-stage pipeline

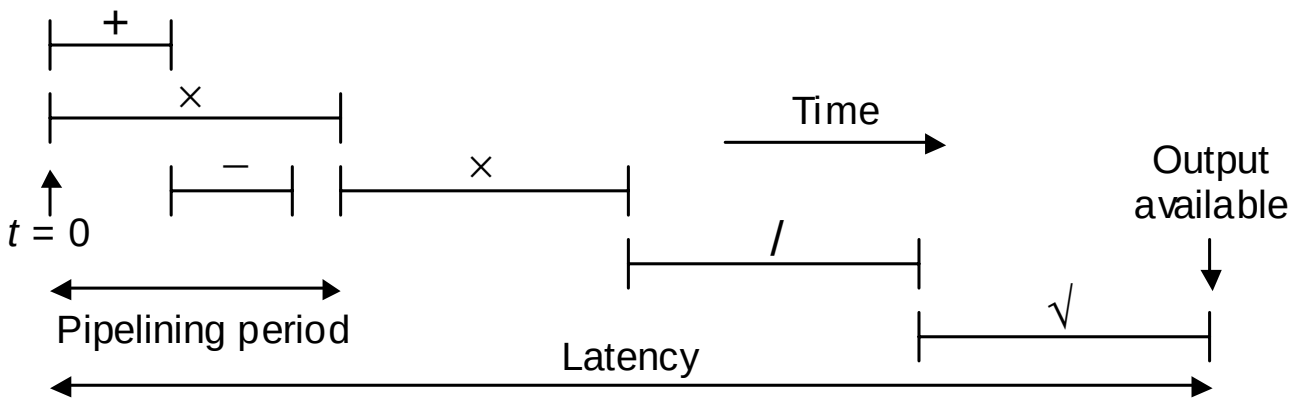


Fig. 25.6 Flow-graph representation of an arithmetic expression and timing diagram for its evaluation with digit-parallel computation.

Feasibility of Bit-Level or Digit-Level Pipelining

Bit-serial addition and multiplication can be done LSB-first, but division and square-rooting are MSB-first operations

Besides, division can't be done in pipelined bit-serial fashion, because the MSB of the quotient *q* in general depends on all the bits of the dividend and divisor

Example: Consider the decimal division .1234/.2469

$$\frac{.1xxx}{.2xxx} = .?xxx \qquad \frac{.12xx}{.24xx} = .?xxx \qquad \frac{.123x}{.246x} = .?xxx$$

Solution: Redundant number representation!

25.5 On-Line or Digit-Pipelined Arithmetic

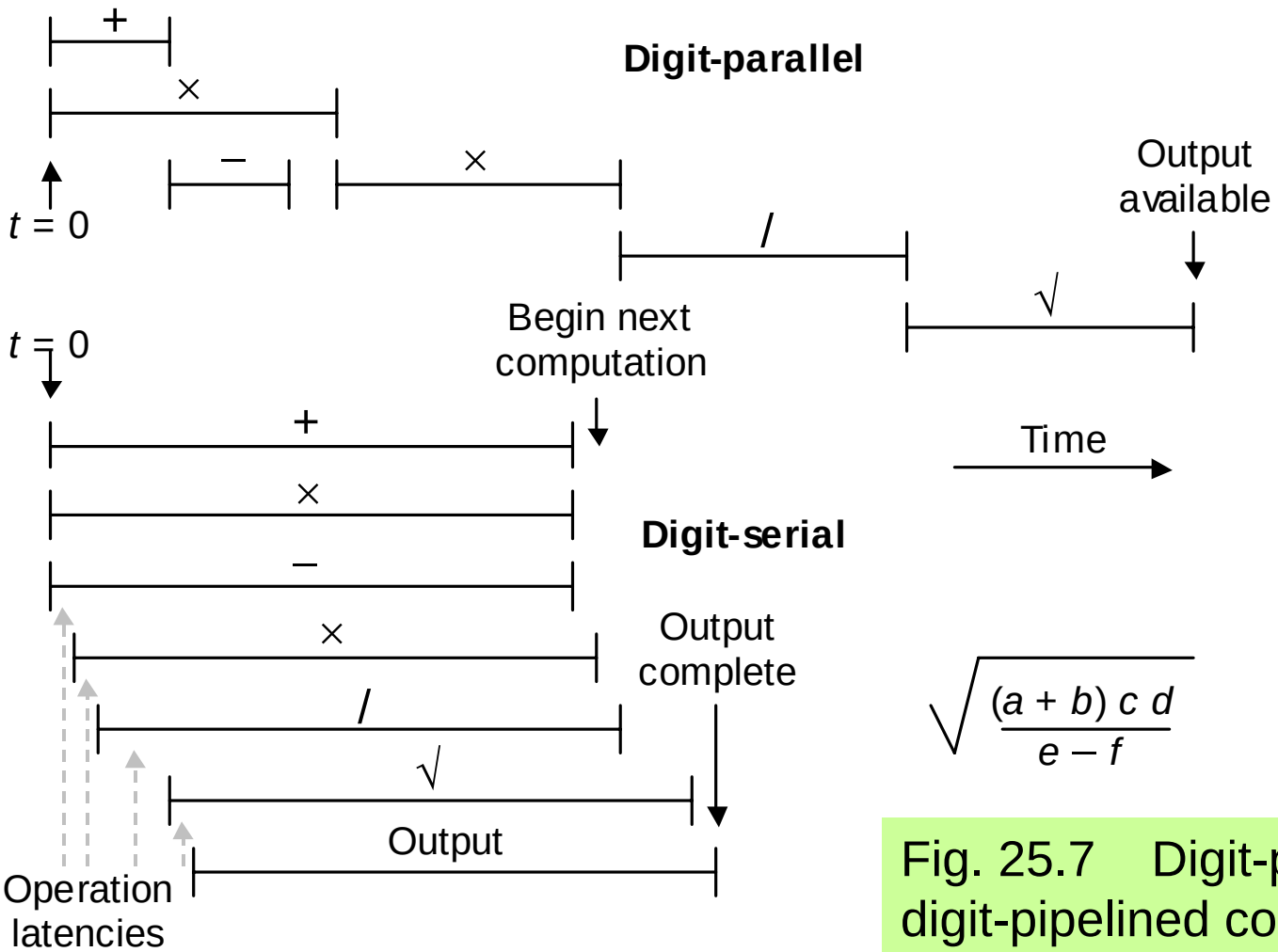
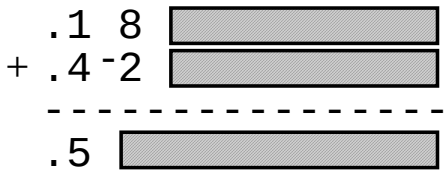


Fig. 25.7 Digit-parallel versus digit-pipelined computation.

Digit-Pipelined Adders

Decimal example:



Shaded boxes show the "unseen" or unprocessed parts of the operands and unknown part of the sum

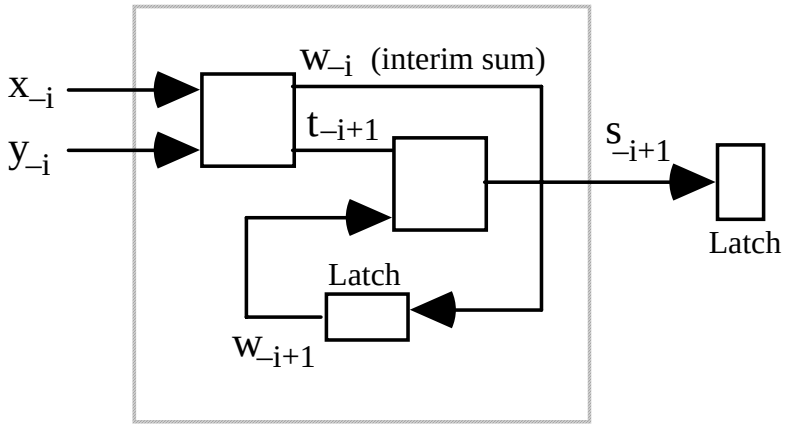
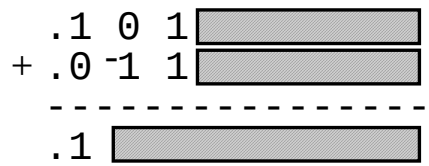


Fig. 25.8 Digit-pipelined MSD-first carry-free addition.

BSD example:



Shaded boxes show the "unseen" or unprocessed parts of the operands and unknown part of the sum

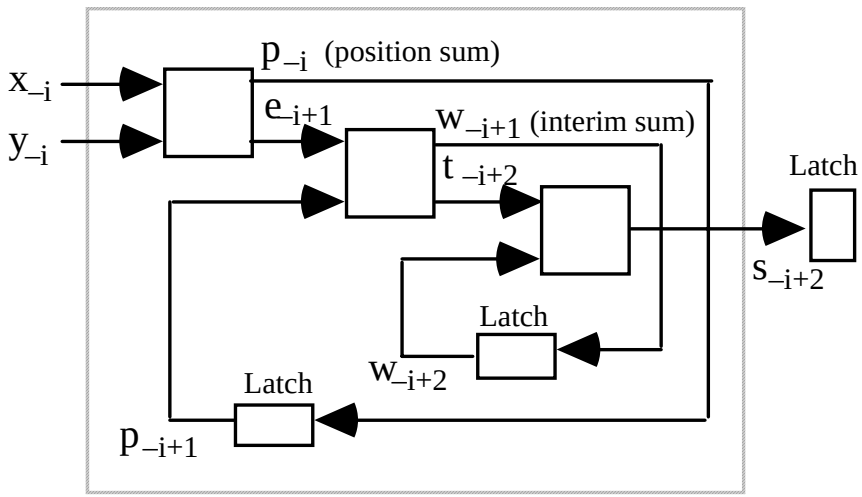


Fig. 25.9 Digit-pipelined MSD-first limited-carry addition.

Digit-Pipelined Multiplier: Algorithm Visualization

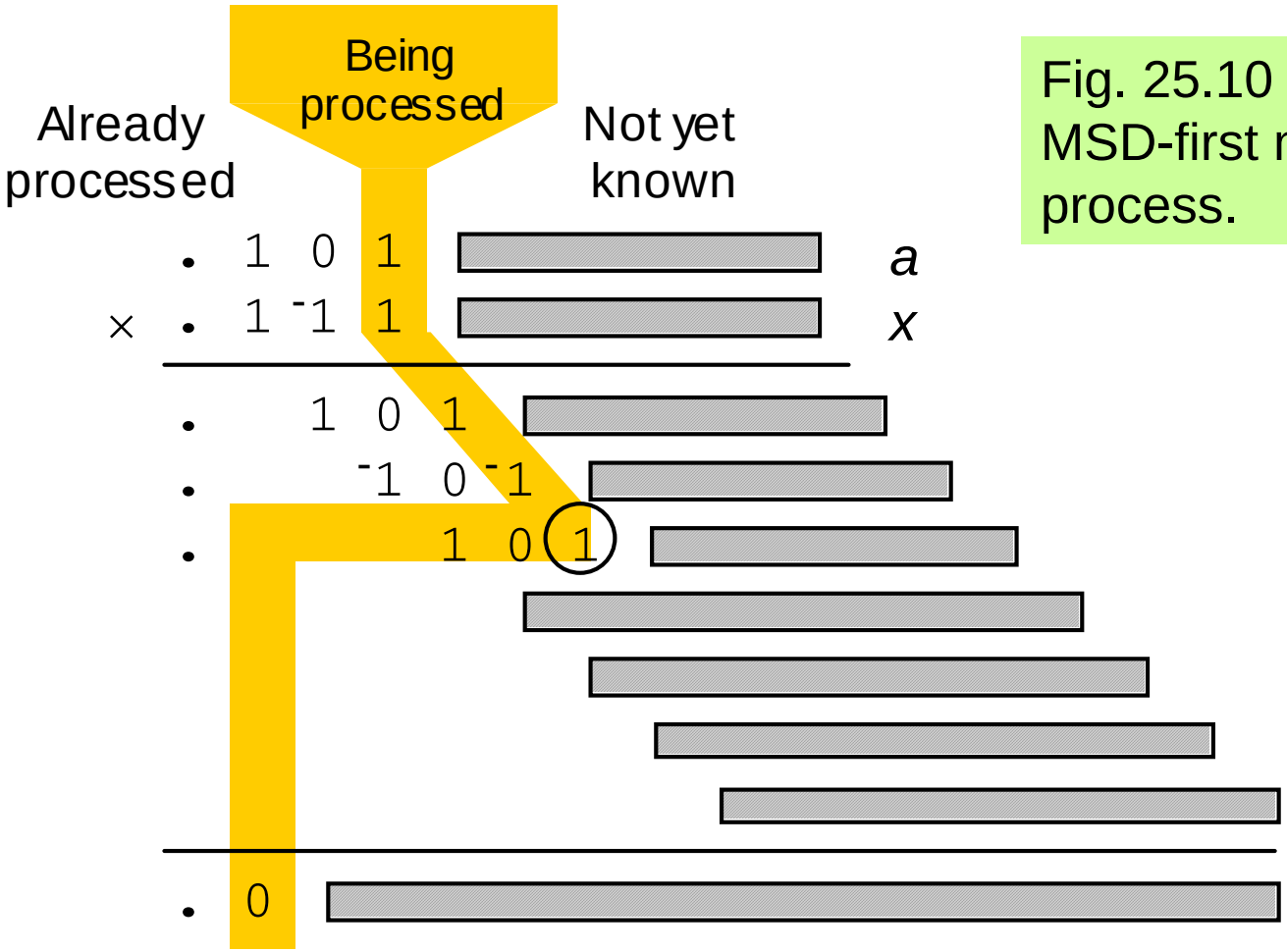


Fig. 25.10 Digit-pipelined MSD-first multiplication process.

Digit-Pipelined Multiplier: BSD Implementation

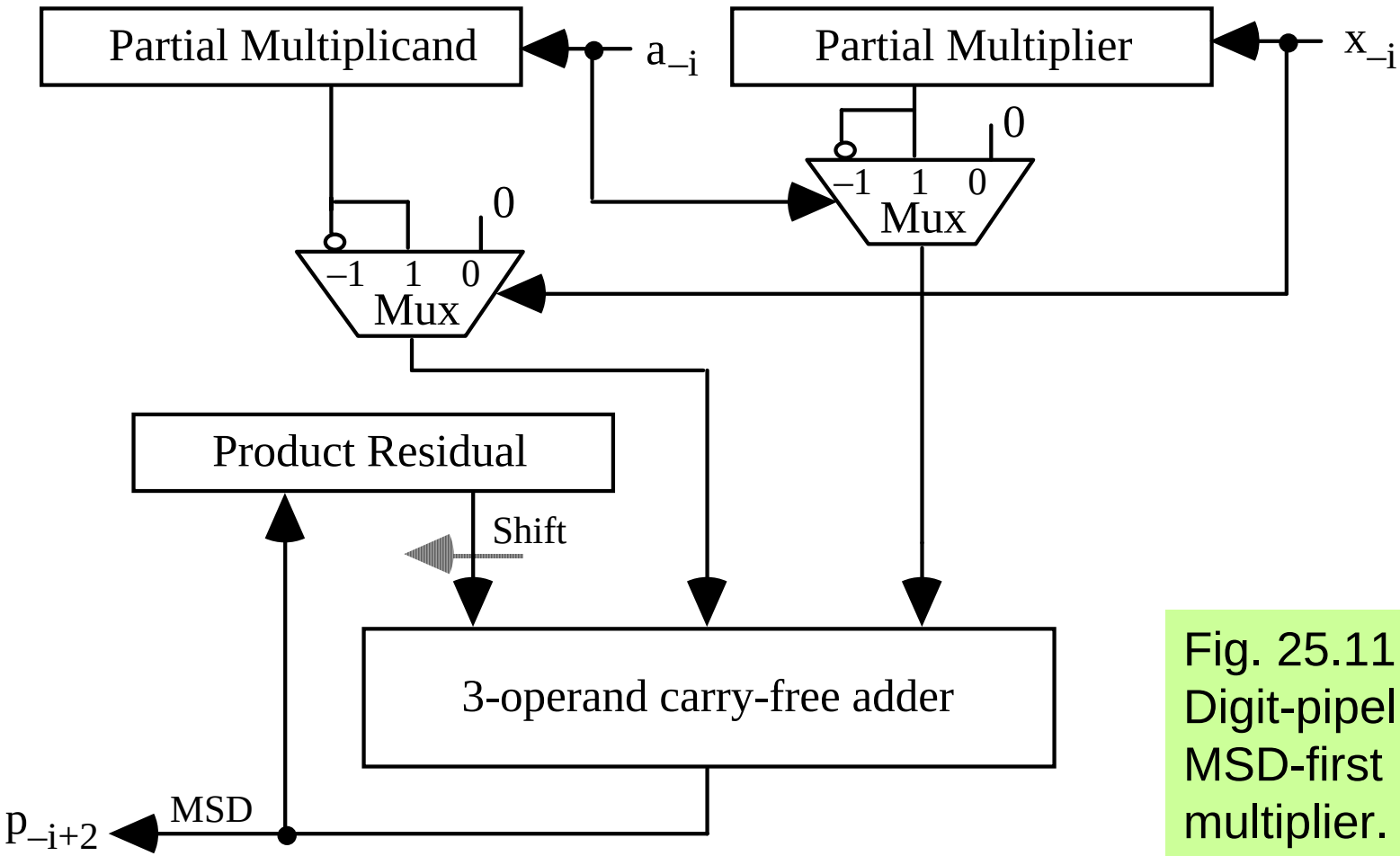


Fig. 25.11
Digit-pipelined
MSD-first BSD
multiplier.

Digit-Pipelined Divider

Table 25.1 Example of digit-pipelined division showing that three cycles of delay are necessary before quotient digits can be output (radix = 4, digit set = [–2, 2])

Cycle	Dividend	Divisor	q Range	q_{-1} Range
1	$(.0\ .\ .\ .)_{four}$	$(.1\ .\ .\ .)_{four}$	(–2/3, 2/3)	[–2, 2]
2	$(.0\ 0\ .\ .\ .)_{four}$	$(.1^{-}2\ .\ .\ .)_{four}$	(–2/4, 2/4)	[–2, 2]
3	$(.0\ 0\ 1\ .\ .\ .)_{four}$	$(.1^{-}2^{-}2\ .\ .\ .)_{four}$	(1/16, 5/16)	[0, 1]
4	$(.0\ 0\ 1\ 0\ .\ .\ .)_{four}$	$(.1^{-}2^{-}2^{-}2\ .\ .\ .)_{four}$	(10/64, 14/64)	1

Digit-Pipelined Square-Rooter

Table 25.2 Examples of digit-pipelined square-root computation showing that 1-2 cycles of delay are necessary before root digits can be output (radix = 10, digit set = [–6, 6], and radix = 2, digit set = [–1, 1])

Cycle	Radicand	q Range	q_{-1} Range
1	$(.3\ .\ .\ .)_{ten}$	$(\sqrt{7/30}\ ,\ \sqrt{11/30})$	[5, 6]
2	$(.3\ 4\ .\ .\ .)_{ten}$	$(\sqrt{1/3}\ ,\ \sqrt{26/75})$	6
1	$(.0\ .\ .\ .)_{two}$	$(0,\ \sqrt{1/2})$	[–2, 2]
2	$(.0\ 1\ .\ .\ .)_{two}$	$(0,\ \sqrt{1/2})$	[0, 1]
3	$(.0\ 1\ 1\ .\ .\ .)_{two}$	$(1/2,\ \sqrt{1/2})$	1

Digit-Pipelined Arithmetic: The Big Picture

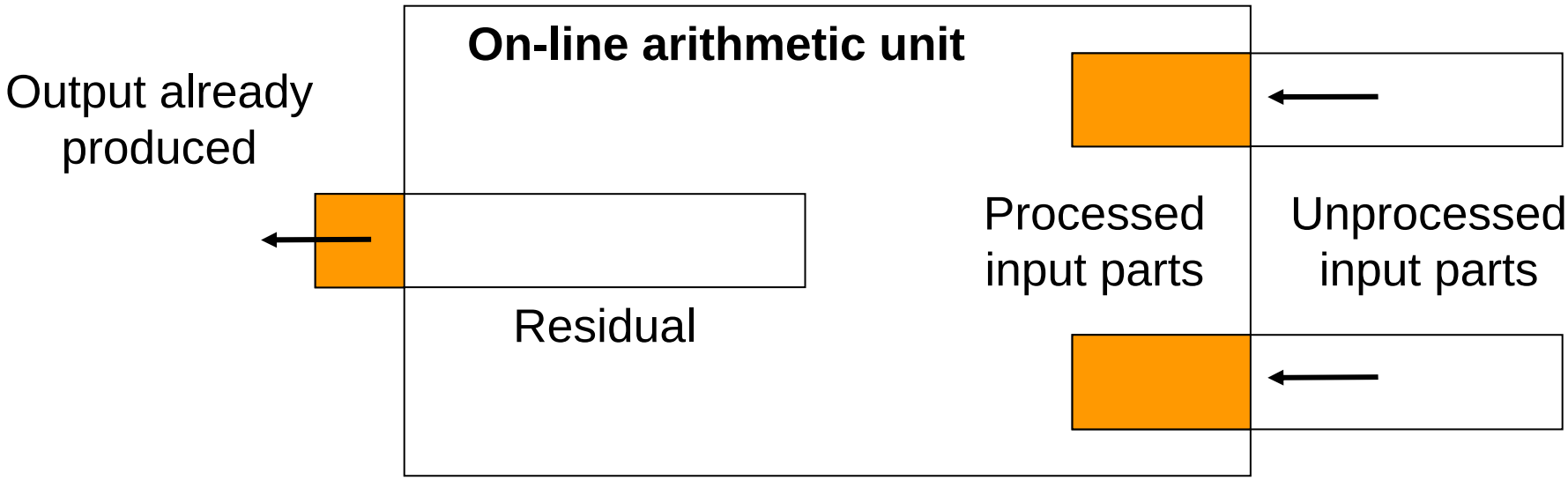


Fig. 25.12 Conceptual view of on-Line or digit-pipelined arithmetic.

25.6 Systolic Arithmetic Units

Systolic arrays: Cellular circuits in which data elements

- Enter at the boundaries
- Advance from cell to cell in lock step
- Are transformed in an incremental fashion
- Leave from the boundaries

Systolic design mitigates the effect of signal propagation delay and allows the use of very clock rates

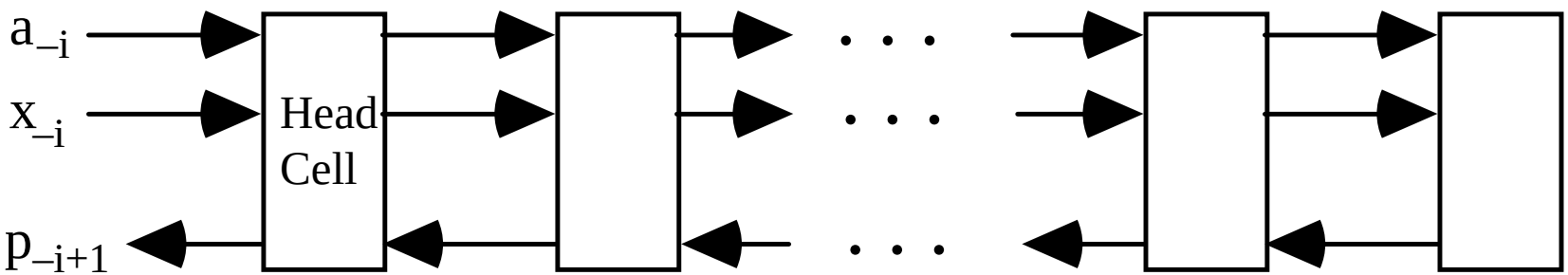
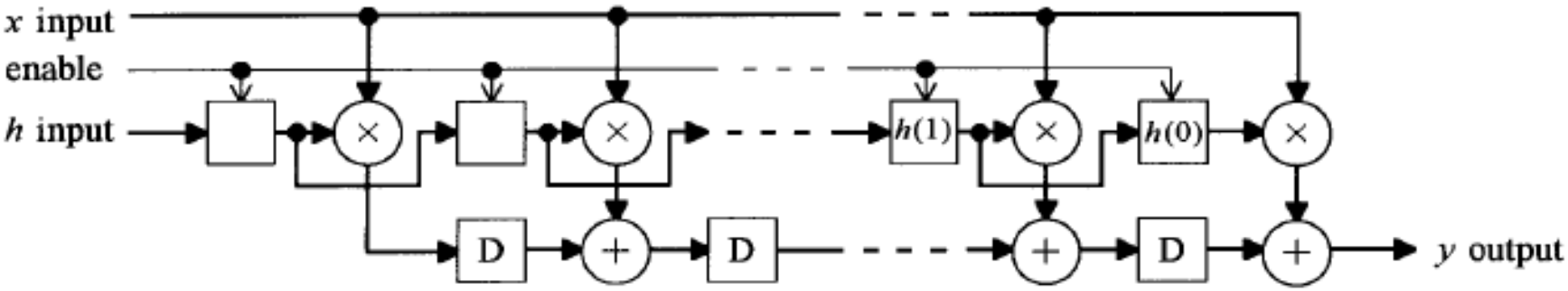
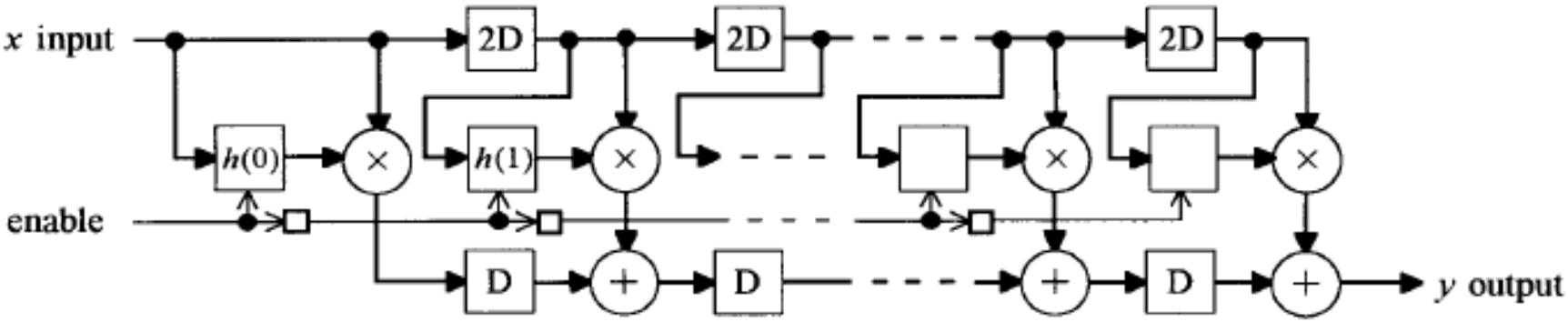


Fig. 25.13 High-level design of a systolic radix-4 digit-pipelined multiplier.

Case Study: Systolic Programmable FIR Filters



(a) Conventional: Broadcast control, broadcast data



(b) Systolic: Pipelined control, pipelined data

Fig. 25.14 Conventional and systolic realizations of a programmable FIR filter.

26 Low-Power Arithmetic

Chapter Goals

Learn how to improve the power efficiency of arithmetic circuits by means of algorithmic and logic design strategies

Chapter Highlights

Reduced power dissipation needed due to
- Limited source (portable, embedded)
- Difficulty of heat disposal

Algorithm and logic-level methods: discussed
Technology and circuit methods: ignored here

Low-Power Arithmetic: Topics

Topics in This Chapter
26.1 The Need for Low-Power Design
26.2 Sources of Power Consumption
26.3 Reduction of Power Waste
26.4 Reduction of Activity
26.5 Transformations and Tradeoffs
26.6 New and Emerging Methods

26.1 The Need for Low-Power Design

Portable and wearable electronic devices

Lithium-ion batteries: 0.2 watt-hr per gram of weight

Practical battery weight < 500 g (< 50 g if wearable device)

Total power ≅ 5-10 watt for a day's work between recharges

Modern high-performance microprocessors use 100s watts

- Power is proportional to die area × clock frequency
- Cooling of micros difficult, but still manageable
- Cooling of MPPs and server farms is a BIG challenge

New battery technologies cannot keep pace with demand

Demand for more speed and functionality (multimedia, etc.)

Processor Power Consumption Trends

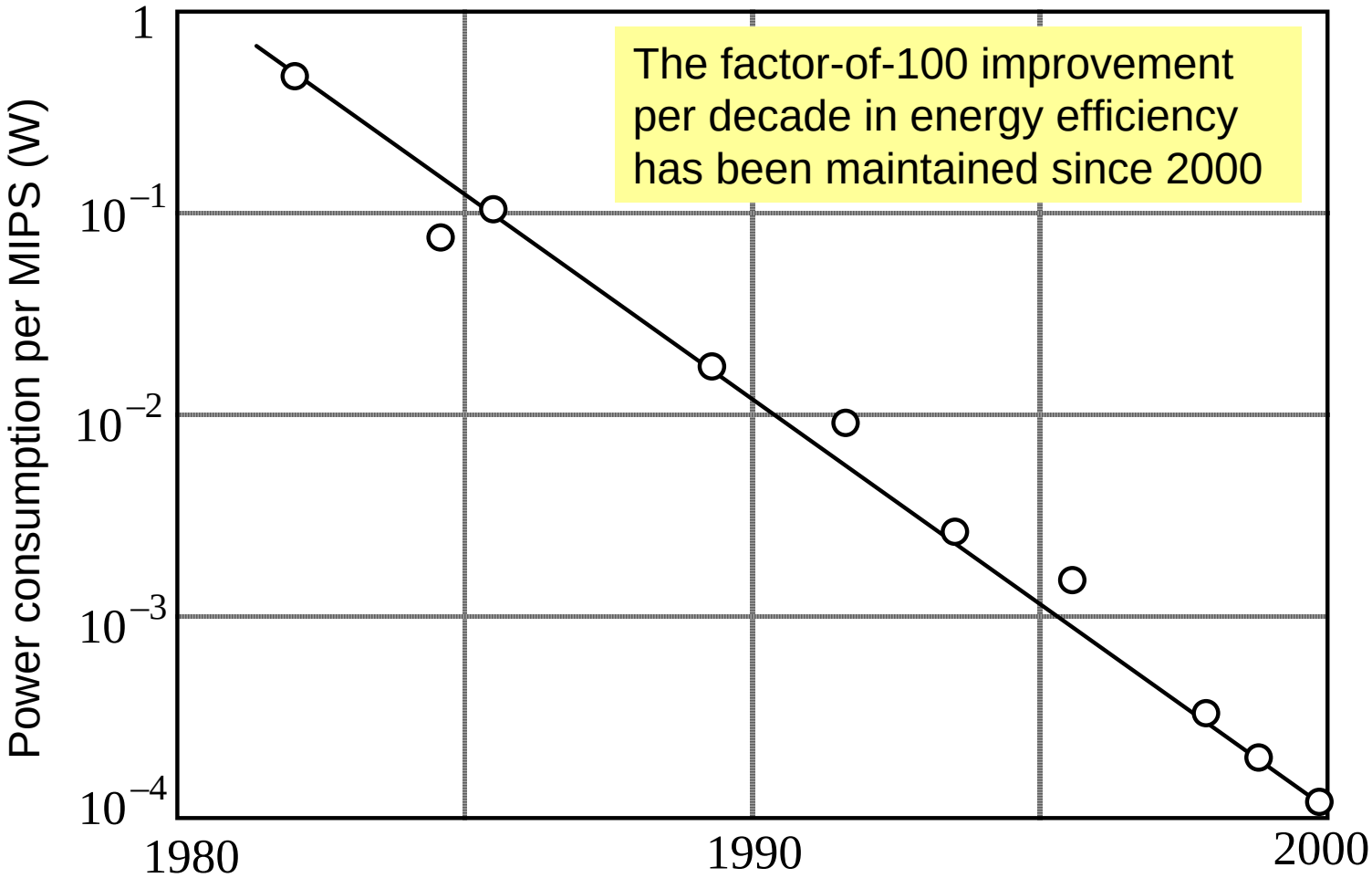


Fig. 26.1 Power consumption trend in DSPs [Raba98].

26.2 Sources of Power Consumption

Both average and peak power are important

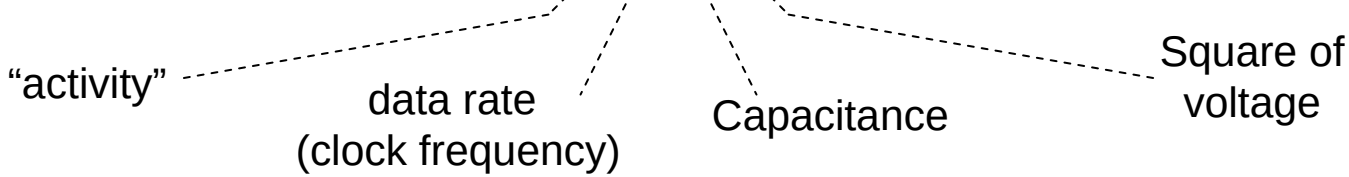
- Average power determines battery life or heat dissipation
- Peak power impacts power distribution and signal integrity

Typically, low-power design aims at reducing both

Power dissipation in CMOS digital circuits

- Static: Leakage current in imperfect switches (< 10%)
- Dynamic: Due to (dis)charging of parasitic capacitance

$$P_{avg} \cong \alpha f C V^2$$



Power Reduction Strategies: The Big Picture

For a given data rate *f*, there are but 3 ways to reduce the power requirements:

1. Using a lower supply voltage *V*
2. Reducing the parasitic capacitance *C*
3. Lowering the switching activity α

$$P_{avg} \cong \alpha f C V^2$$

Example: A 32-bit off-chip bus operates at 5 V and 100 MHz and drives a capacitance of 30 pF per bit. If random values were put on the bus in every cycle, we would have $\alpha = 0.5$. To account for data correlation and idle bus cycles, assume $\alpha = 0.2$. Then:

$$P_{avg} \cong \alpha f C V^2 = 0.2 \times 10^8 \times (32 \times 30 \times 10^{-12}) \times 5^2 = 0.48 \text{ W}$$

26.3 Reduction of Power Waste

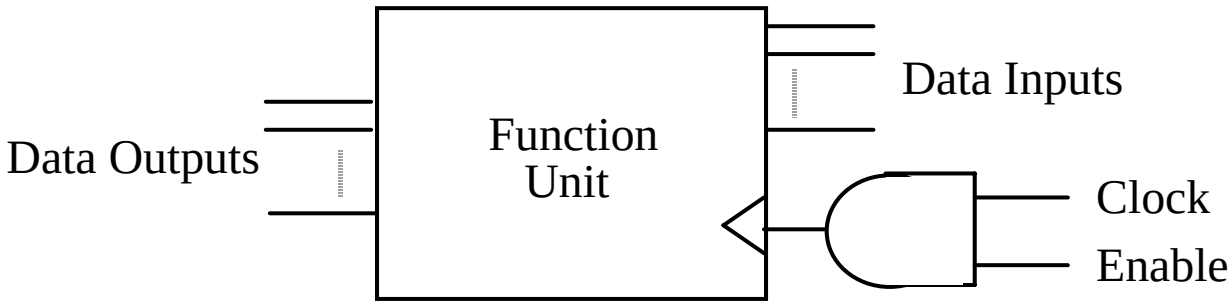


Fig. 26.2 Saving power through clock gating.

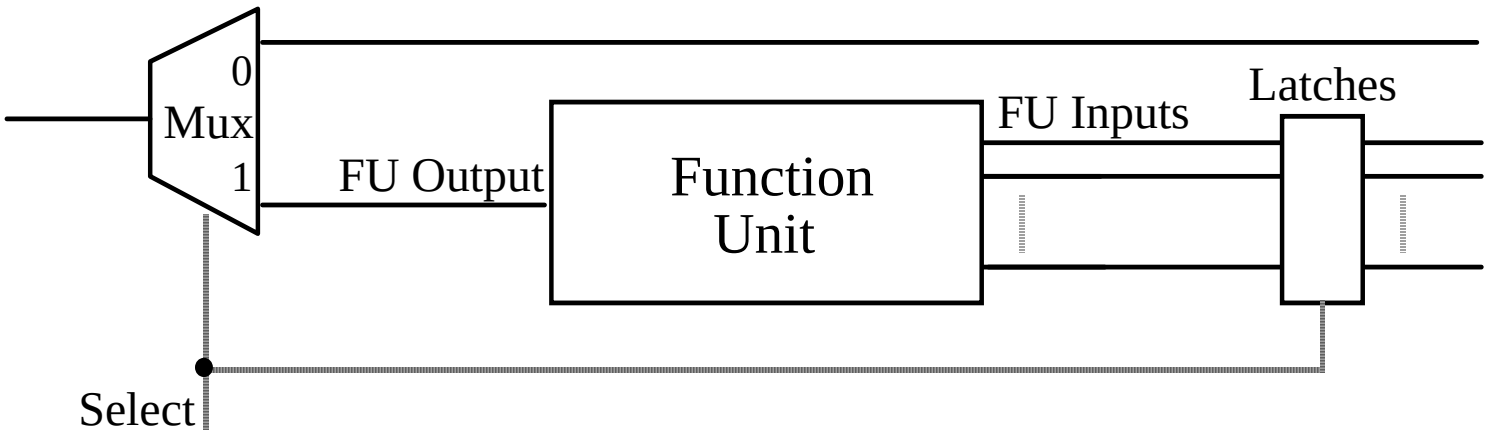


Fig. 26.3 Saving power via guarded evaluation.

Glitching and Its Impact on Power Waste

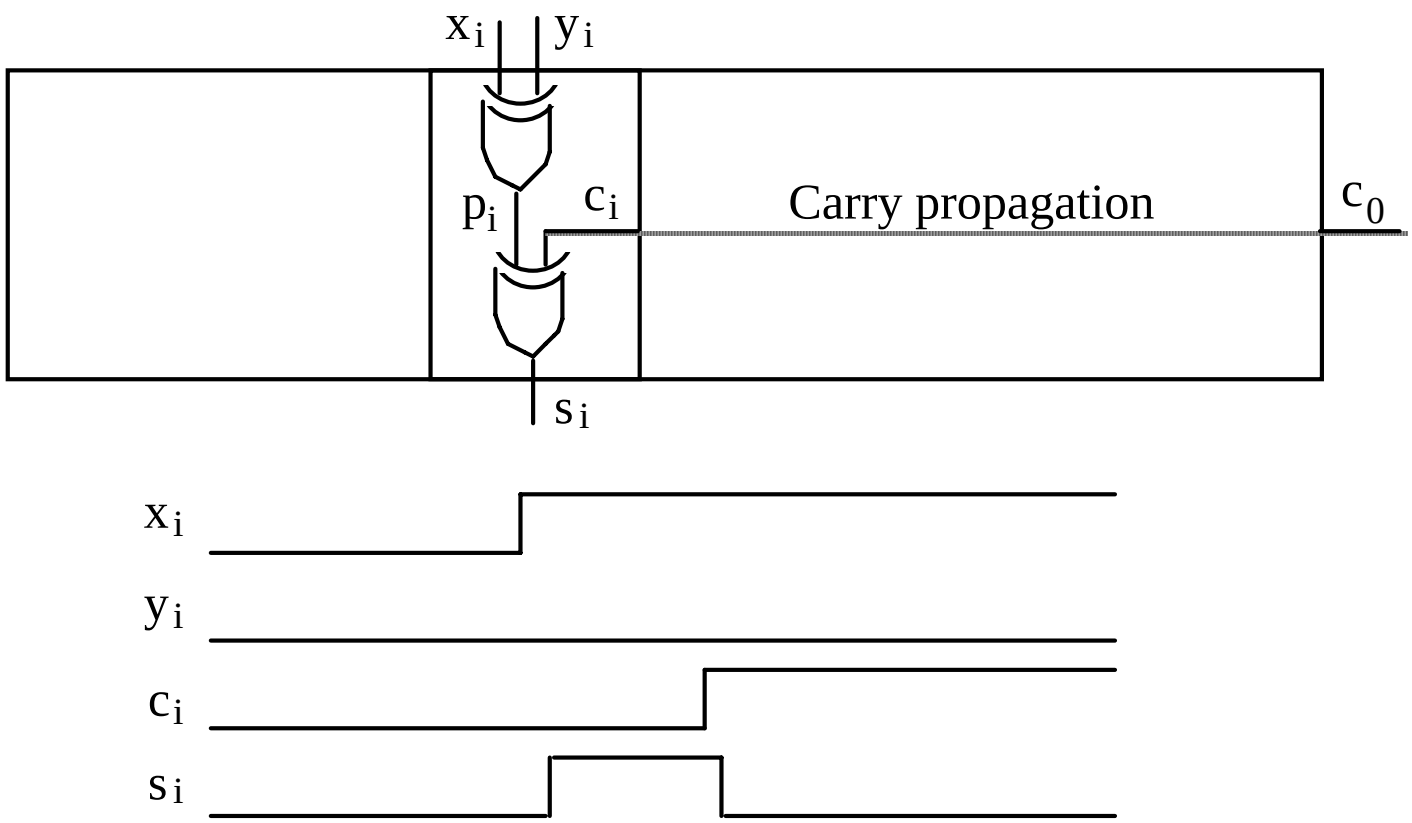


Fig. 26.4 Example of glitching in a ripple-carry adder.

Array Multipliers with Lower Power Consumption

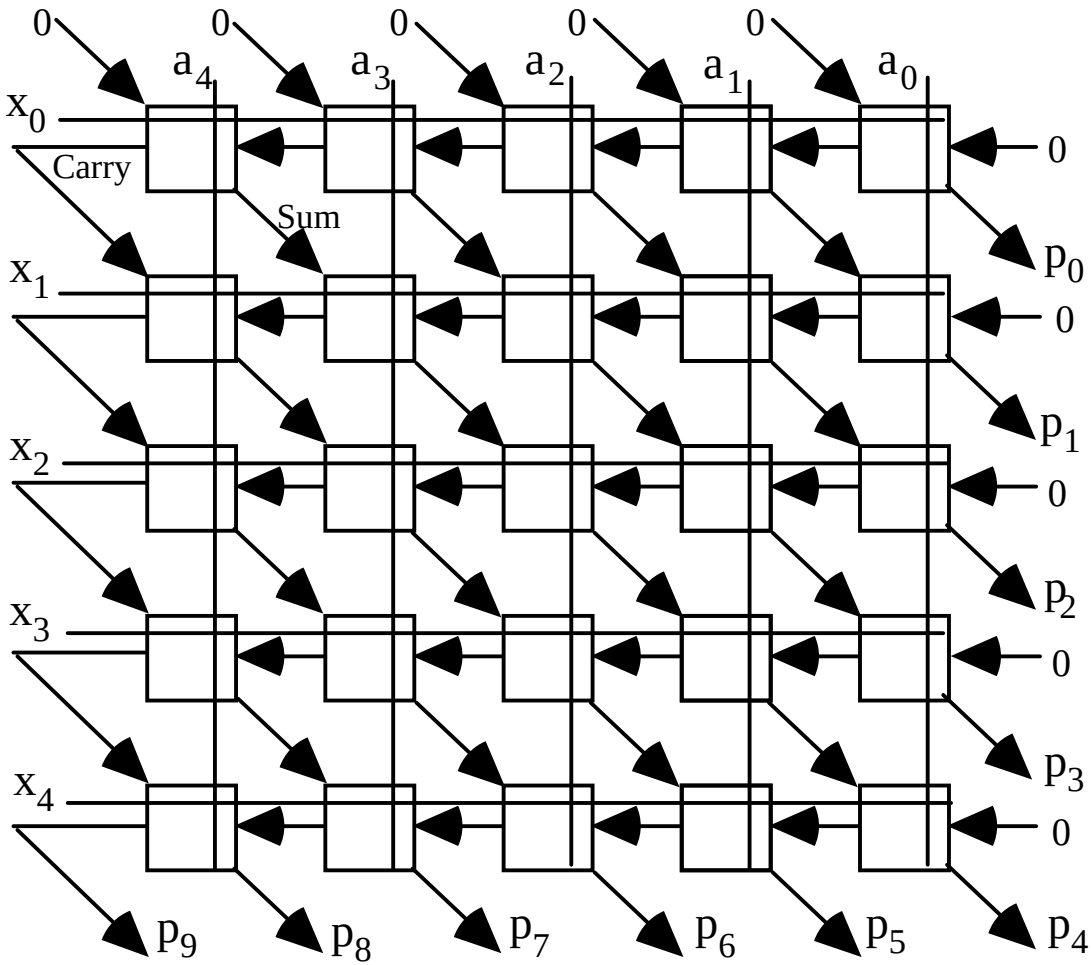


Fig. 26.5 An array multiplier with gated FA cells.

26.4 Reduction of Activity

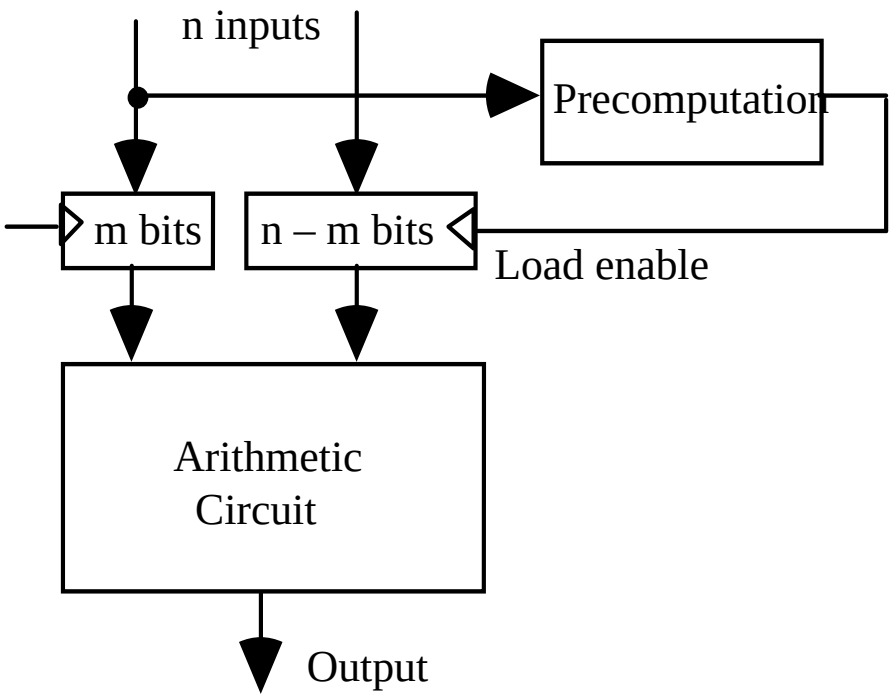


Fig. 26.6 Reduction of activity by precomputation.

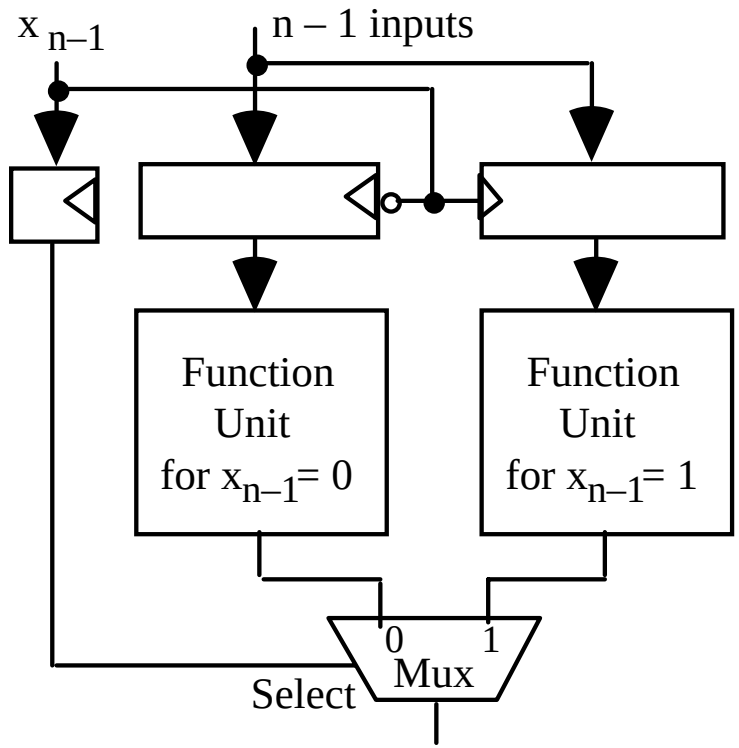
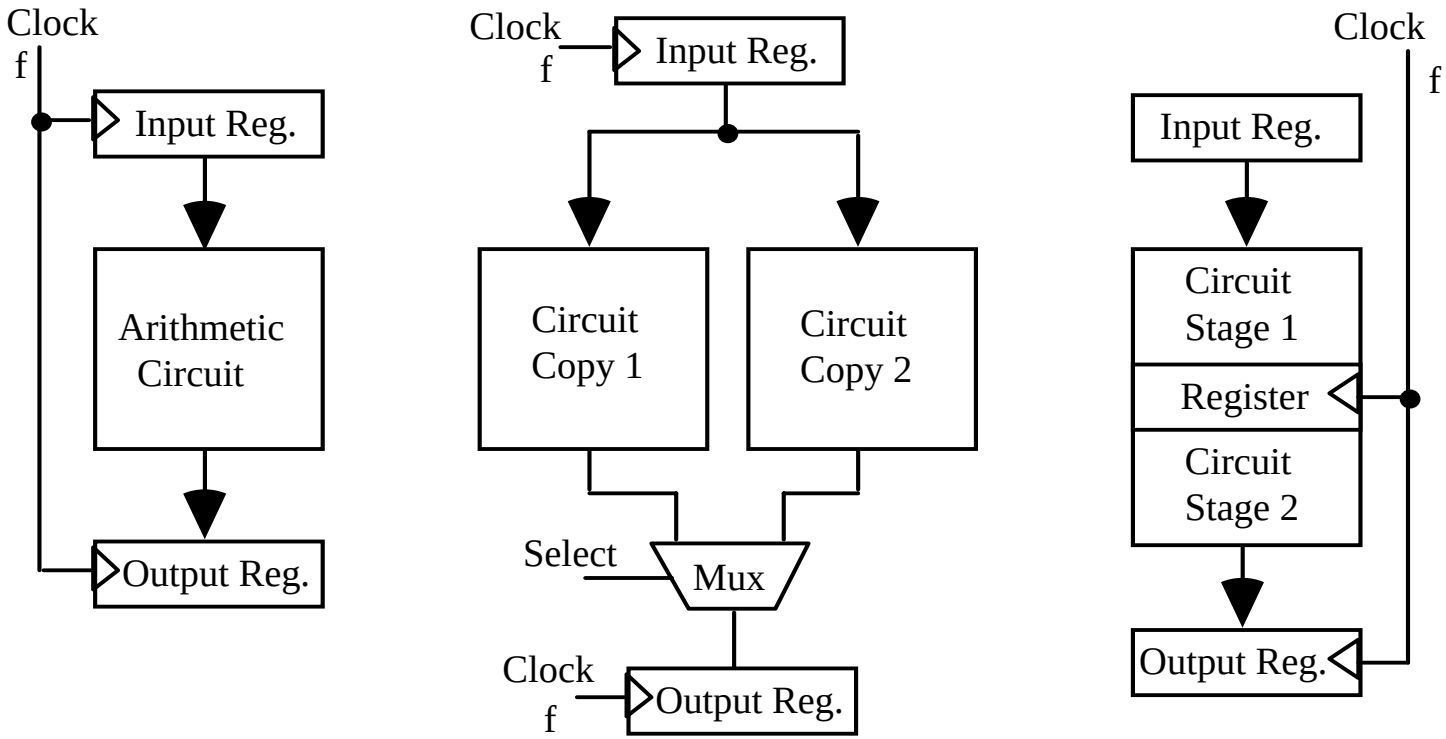


Fig. 26.7 Reduction of activity via Shannon expansion.

26.5 Transformations and Tradeoffs



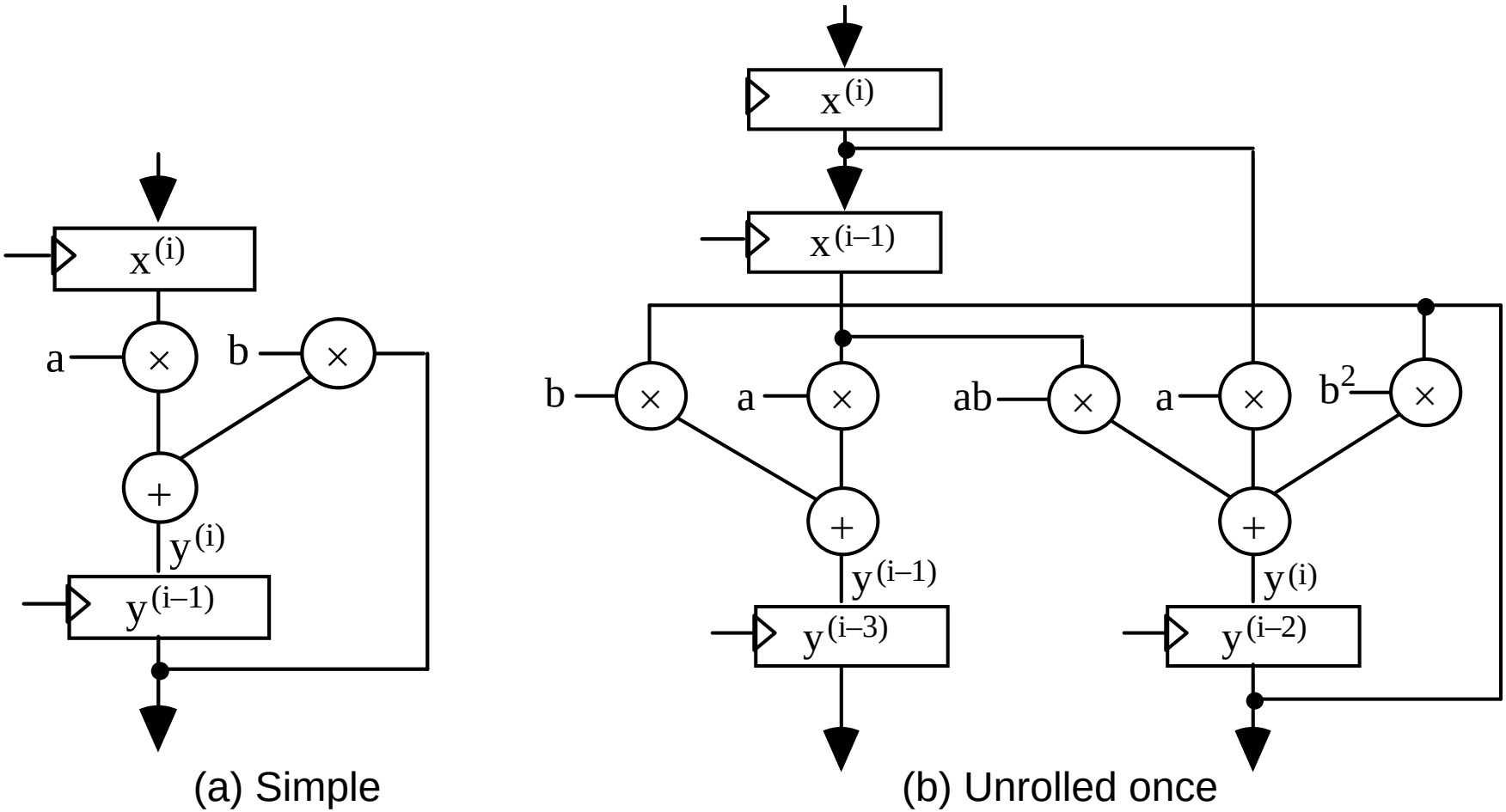
Frequency = f
Capacitance = C
Voltage = V
Power = P

Frequency = 0.5f
Capacitance = 2.2C
Voltage = 0.6V
Power = 0.396P

Frequency = f
Capacitance = 1.2C
Voltage = 0.6V
Power = 0.432P

Fig. 26.8 Reduction of power via parallelism or pipelining.

Unrolling of Iterative Computations



(a) Simple

(b) Unrolled once

Fig. 26.9 Realization of a first-order IIR filter.

Retiming for Power Efficiency

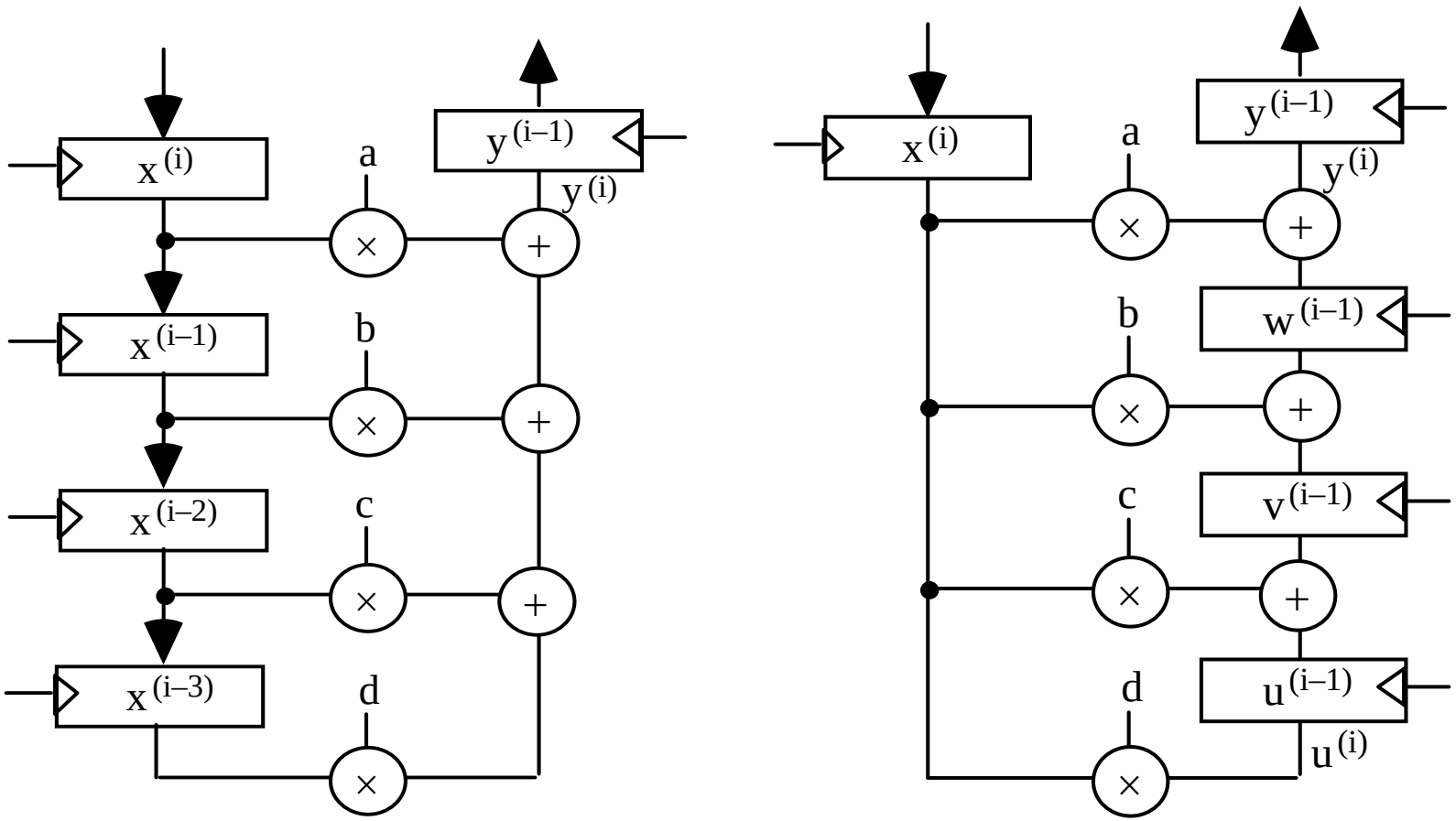
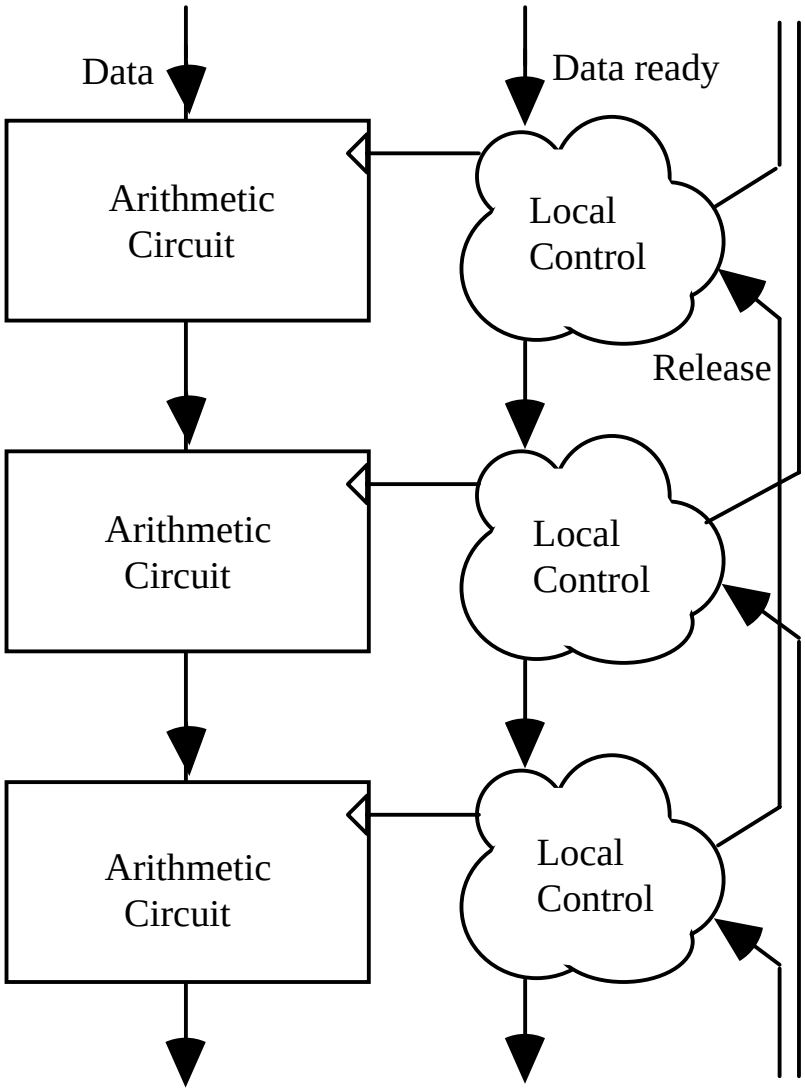


Fig. 26.10 Possible realizations of a fourth-order FIR filter.

26.6 New and Emerging Methods



Dual-rail data encoding with transition signaling:

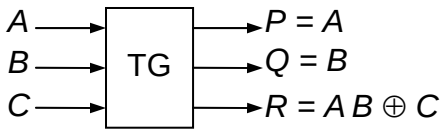
Two wires per signal

Transition on wire 0 (1) indicates the arrival of 0 (1)

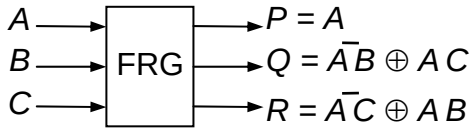
Dual-rail design does increase the wiring density, but it offers the advantage of complete insensitivity to delays

Fig. 26.11 Part of an asynchronous chain of computations.

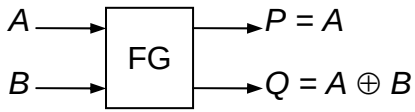
The Ultimate in Low-Power Design



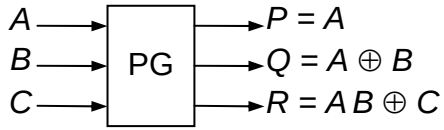
(a) Toffoli gate



(b) Fredkin gate



(c) Feynman gate



(d) Peres gate

Fig. 26.12
Some reversible logic gates.

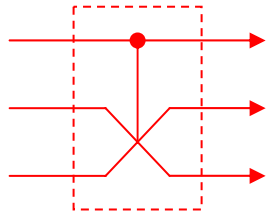
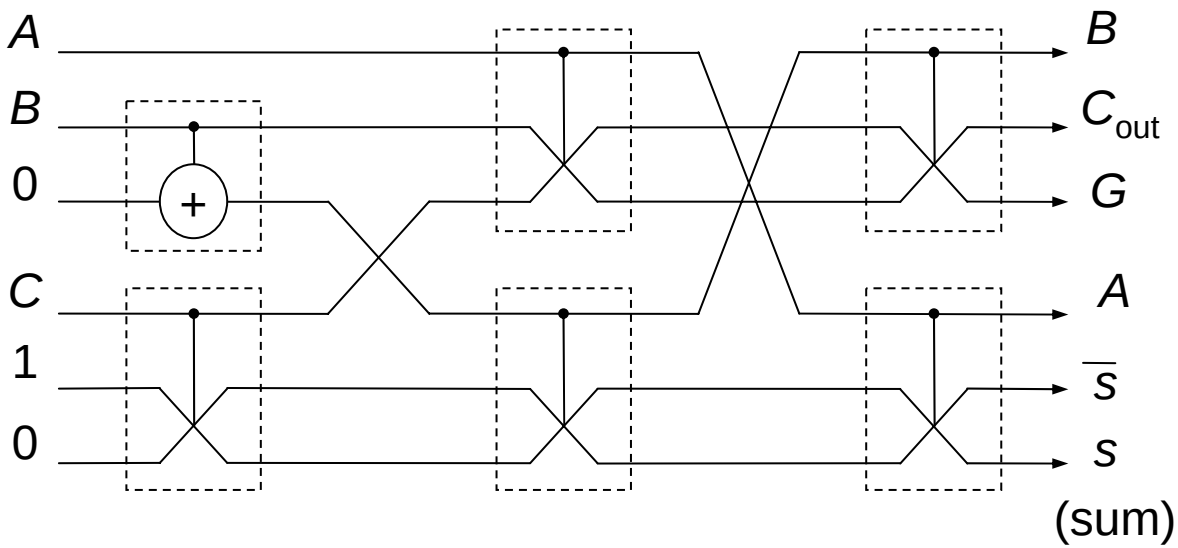


Fig. 26.13 Reversible binary full adder built of 5 Fredkin gates, with a single Feynman gate used to fan out the input B. The label "G" denotes "garbage."



27 Fault-Tolerant Arithmetic

Chapter Goals

Learn about errors due to hardware faults
or hostile environmental conditions,
and how to deal with or circumvent them

Chapter Highlights

Modern components are very robust, but . . .
put millions/billions of them together
and something is bound to go wrong
Can arithmetic be protected via encoding?
Reliable circuits and robust algorithms

Fault-Tolerant Arithmetic: Topics

Topics in This Chapter
27.1 Faults, Errors, and Error Codes
27.2 Arithmetic Error-Detecting Codes
27.3 Arithmetic Error-Correcting Codes
27.4 Self-Checking Function Units
27.5 Algorithm-Based Fault Tolerance
27.6 Fault-Tolerant RNS Arithmetic

27.1 Faults, Errors, and Error Codes

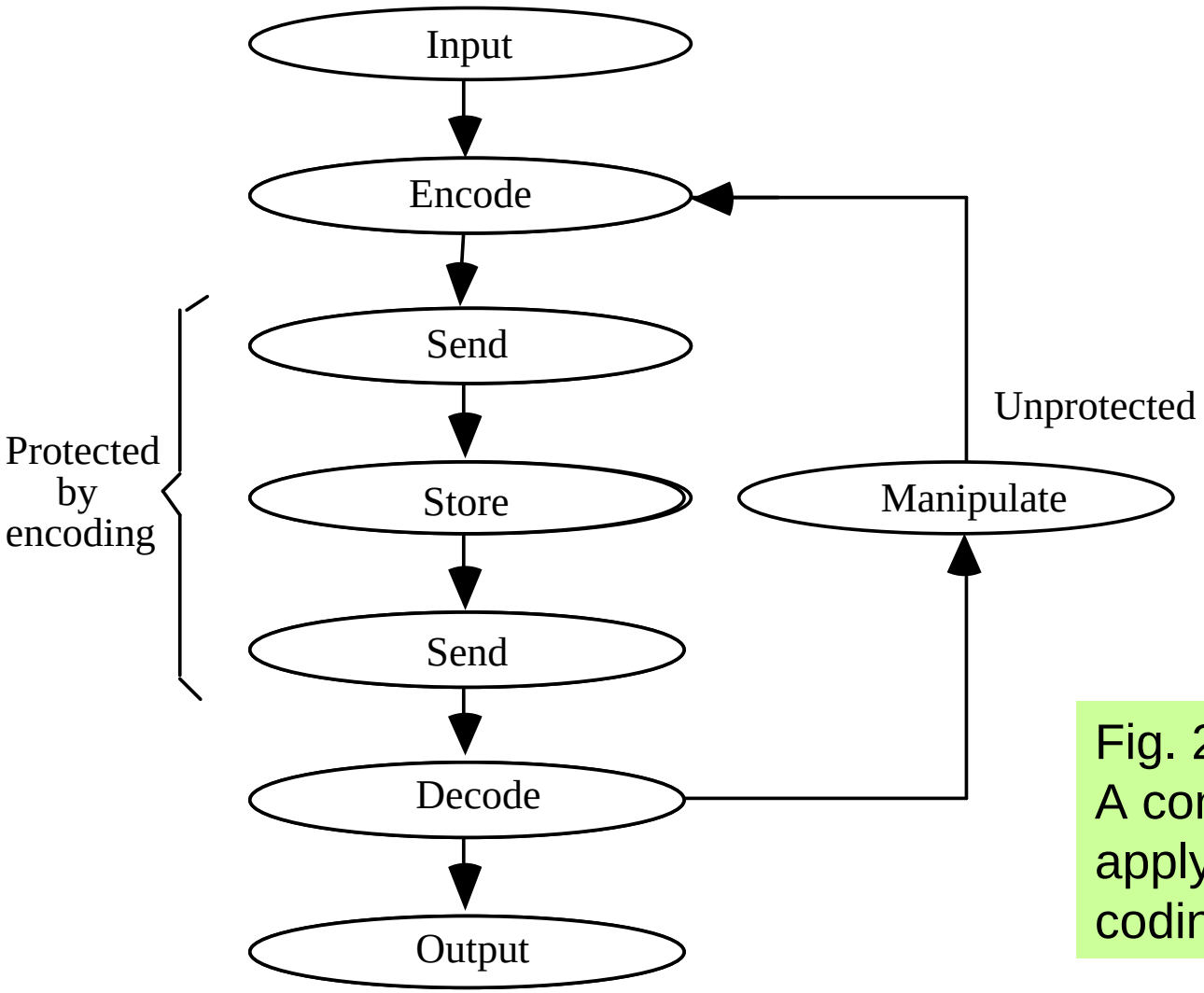


Fig. 27.1
A common way of applying information coding techniques.

Fault Detection and Fault Masking

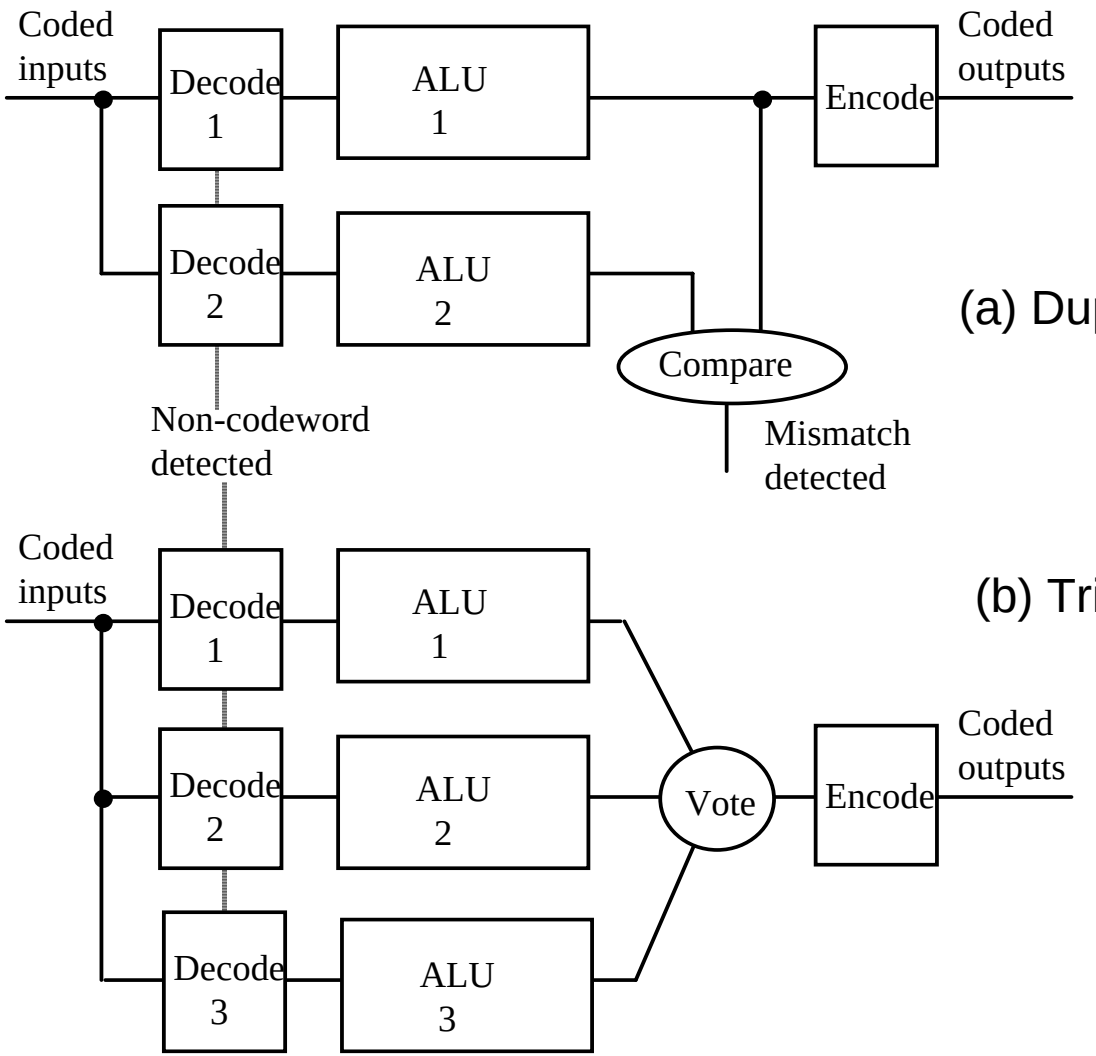


Fig. 27.2 Arithmetic fault detection or fault tolerance (masking) with replicated units.

Inadequacy of Standard Error Coding Methods

Unsigned addition	0010 0111 0010 0001
	+ 0101 1000 1101 0011
Correct sum	0111 1111 1111 0100
Erroneous sum	1000 0000 0000 0100

↑ Stage generating an erroneous carry of 1

Fig. 27.3 How a single carry error can produce an arbitrary number of bit-errors (inversions).

The *arithmetic weight* of an error: Min number of signed powers of 2 that must be added to the correct value to produce the erroneous result

	Example 1	Example 2
Correct value	0111 1111 1111 0100	1101 1111 1111 0100
Erroneous value	1000 0000 0000 0100	0110 0000 0000 0100
Difference (error)	$16 = 2^4$	$-32752 = -2^{15} + 2^4$
Min-weight BSD	0000 0000 0001 0000	$^-$1000 0000 0001 0000
Arithmetic weight	1	2
Error type	Single, positive	Double, negative

27.2 Arithmetic Error-Detecting Codes

Arithmetic error-detecting codes:

- Are characterized by arithmetic weights of detectable errors
- Allow direct arithmetic on coded operands

We will discuss two classes of arithmetic error-detecting codes, both of which are based on a check modulus *A* (usually a small odd number)

- Product or *AN* codes
  - Represent the value *N* by the number *AN*
- Residue (or inverse residue) codes
  - Represent the value *N* by the pair (*N*, *C*), where *C* is *N* mod *A* or (*N* – *N* mod *A*) mod *A*

Product or *AN* Codes

For odd *A*, all weight-1 arithmetic errors are detected

Arithmetic errors of weight ≥ 2 may go undetected

e.g., the error 32 736 = $2^{15} - 2^5$ undetectable with *A* = 3, 11, or 31

Error detection: check divisibility by *A*

Encoding/decoding: multiply/divide by *A*

Arithmetic also requires multiplication and division by *A*

Product codes are *nonseparate* (*nonseparable*) codes
Data and redundant check info are intermixed

Low-Cost Product Codes

Low-cost product codes use low-cost check moduli of the form $A = 2^a - 1$

Multiplication by $A = 2^a - 1$: done by shift-subtract

Division by $A = 2^a - 1$: done a bits at a time as follows

Given $y = (2^a - 1)x$, find x by computing $2^a x - y$

. . . xxxx 0000 – . . . xxxx xxxx = . . . xxxx xxxx

Unknown $2^a x$ Known $(2^a - 1)x$ Unknown x

Theorem 27.1: Any unidirectional error with arithmetic weight of at most $a - 1$ is detectable by a low-cost product code based on $A = 2^a - 1$

Arithmetic on *AN*-Coded Operands

Add/subtract is done directly: $Ax \pm Ay = A(x \pm y)$

Direct multiplication results in: $Aa \times Ax = A^2ax$

The result must be corrected through division by A

For division, if $z = qd + s$, we have: $Az = q(Ad) + As$

Thus, q is unprotected
Possible cure: premultiply the dividend Az by A
The result will need correction

Square rooting leads to a problem similar to division

$\lfloor \sqrt{A^2x} \rfloor = \lfloor A\sqrt{x} \rfloor$ which is not the same as $A \lfloor \sqrt{x} \rfloor$

Residue and Inverse Residue Codes

Represent N by the pair $(N, C(N))$, where $C(N) = N \bmod A$

Residue codes are *separate* (*separable*) codes

Separate data and check parts make decoding trivial

Encoding: given N, compute $C(N) = N \bmod A$

Low-cost residue codes use $A = 2^a - 1$

Arithmetic on residue-coded operands

Add/subtract: data and check parts are handled separately

$$(x, C(x)) \pm (y, C(y)) = (x \pm y, (C(x) \pm C(y)) \bmod A)$$

Multiply

$$(a, C(a)) \times (x, C(x)) = (a \times x, (C(a) \times C(x)) \bmod A)$$

Divide/square-root: difficult

Arithmetic on Residue-Coded Operands

Add/subtract: Data and check parts are handled separately

$$(x, C(x)) \pm (y, C(y)) = (x \pm y, (C(x) \pm C(y)) \bmod A)$$

Multiply

$$(a, C(a)) \times (x, C(x)) = (a \times x, (C(a) \times C(x)) \bmod A)$$

Divide/square-root: difficult

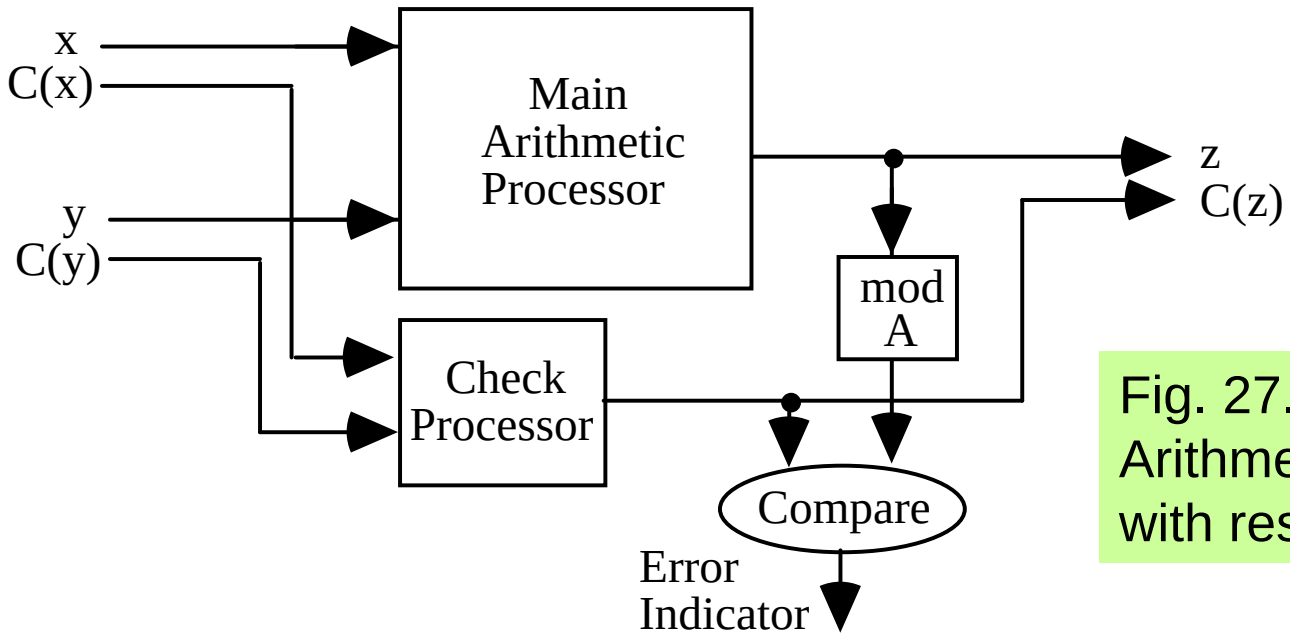
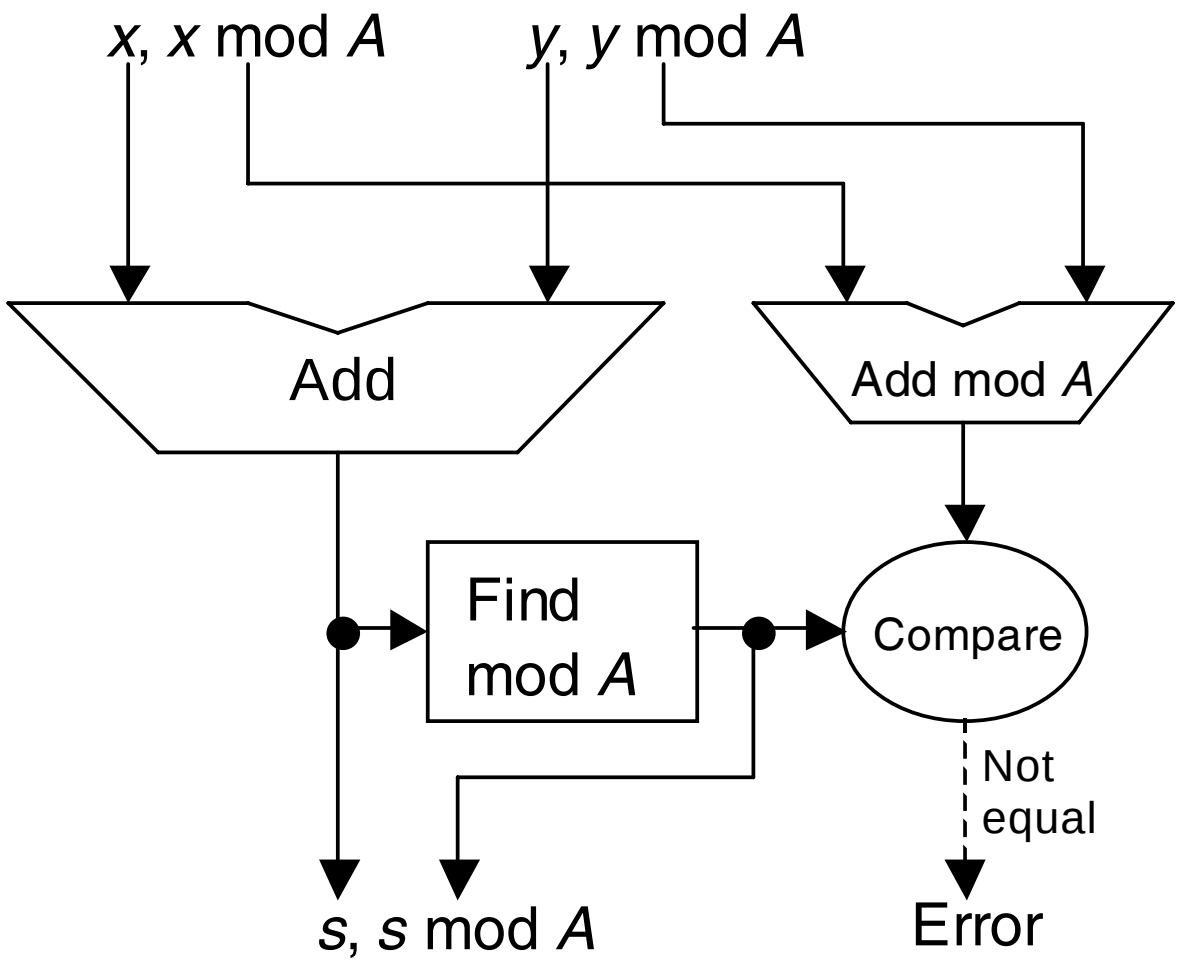


Fig. 27.4
Arithmetic processor with residue checking.

Example: Residue Checked Adder



27.3 Arithmetic Error-Correcting Codes

Positive error	Syndrome mod 7	Syndrome mod 15	Negative error	Syndrome mod 7	Syndrome mod 15
1	1	1	–1	6	14
2	2	2	–2	5	13
4	4	4	–4	3	11
8	1	8	–8	6	7
16	2	1	–16	5	14
32	4	2	–32	3	13
64	1	4	–64	6	11
128	2	8	–128	5	7
256	4	1	–256	3	14
512	1	2	–512	6	13
1024	2	4	–1024	5	11
2048	4	8	–2048	3	7
4096	1	1	–4096	6	14
8192	2	2	–8192	5	13
16,384	4	4	–16,384	3	11
32,768	1	8	–32,768	6	7

Table 27.1
Error syndromes for weight-1 arithmetic errors in the (7, 15) biresidue code

Because all the symptoms in this table are different, any weight-1 arithmetic error is correctable by the (mod 7, mod 15) biresidue code

Properties of Biresidue Codes

Biresidue code with relatively prime low-cost check moduli $A = 2^a - 1$ and $B = 2^b - 1$ supports $a \times b$ bits of data for weight-1 error correction

Representational redundancy = $(a + b)/(ab) = 1/a + 1/b$

27.4 Self-Checking Function Units

Self-checking (SC) unit: any fault from a prescribed set does not affect the correct output (masked) or leads to a noncodeword output (detected)

An invalid result is:

- Detected immediately by a code checker, or
- Propagated downstream by the next self-checking unit

To build SC units, we need SC code checkers that never validate a noncodeword, even when they are faulty

Design of a Self-Checking Code Checker

Example: SC checker for inverse residue code (N, $C'(N)$)
N mod A should be the bitwise complement of $C'(N)$

Verifying that signal pairs (x_i, y_i) are all (1, 0) or (0, 1) is the same as finding the AND of Boolean values encoded as:

1: (1, 0) or (0, 1) 0: (0, 0) or (1, 1)

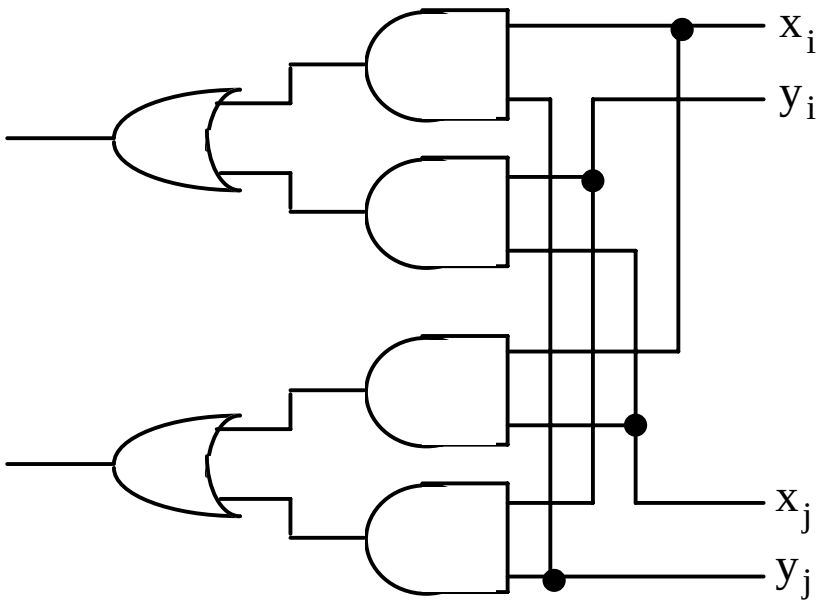
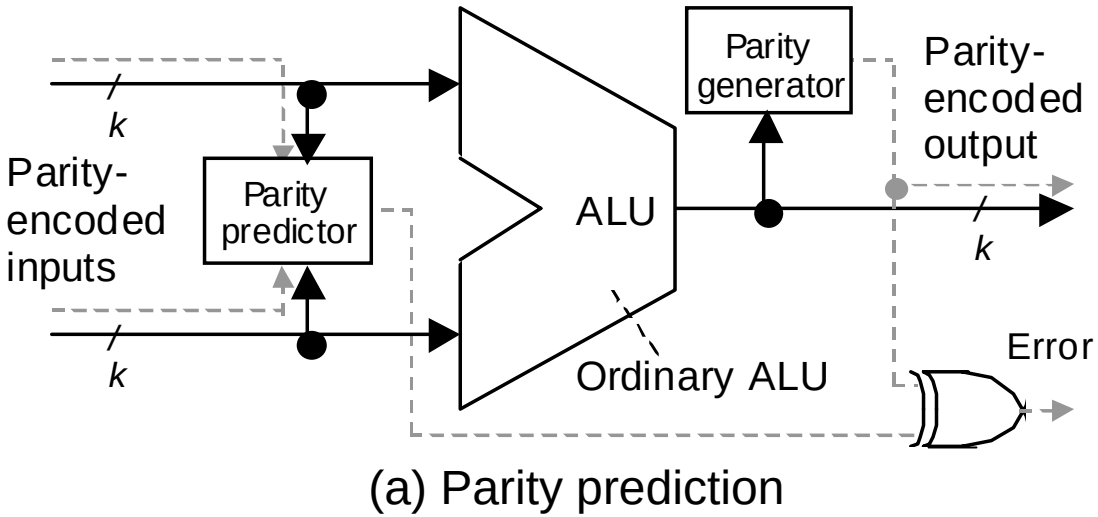
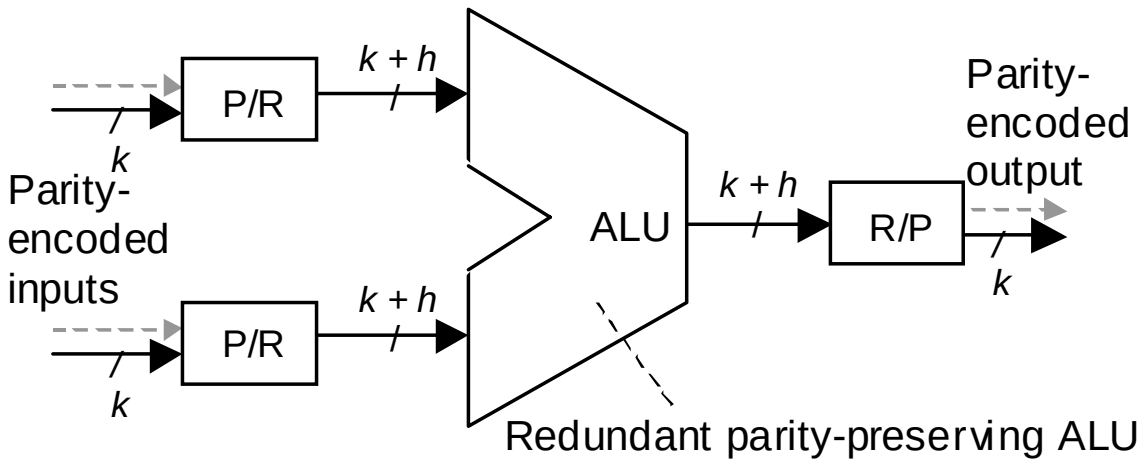


Fig. 27.5 Two-input AND circuit, with 2-bit inputs (x_i, y_i) and (x_i, y_i), for use in a self-checking code checker.

Case Study: Self-Checking Adders



(a) Parity prediction



(b) Parity preservation

Fig. 27.6
Self-checking adders with parity-encoded inputs and output.

P/R = Parity-to-redundant converter

R/P = Redundant-to-parity converter

27.5 Algorithm-Based Fault Tolerance

Alternative strategy to error detection after each basic operation:

Accept that operations may yield incorrect results

Detect/correct errors at data-structure or application level

Example: multiplication of matrices X and Y yielding P
Row, column, and full checksum matrices (mod 8)

$$M = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{pmatrix} \qquad M_r = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{pmatrix}$$

$$M_c = \begin{pmatrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{pmatrix} \qquad M_f = \begin{pmatrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{pmatrix}$$

Fig. 27.7 A 3×3 matrix M with its row, column, and full checksum matrices M_r, M_c, and M_f.

Properties of Checksum Matrices

Theorem 27.3: If $P = X \times Y$, we have $P_f = X_c \times Y_r$
(with floating-point values, the equalities are approximate)

$$M = \begin{matrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \end{matrix} \qquad M_r = \begin{matrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \end{matrix}$$

$$M_c = \begin{matrix} 2 & 1 & 6 \\ 5 & 3 & 4 \\ 3 & 2 & 7 \\ 2 & 6 & 1 \end{matrix} \qquad M_f = \begin{matrix} 2 & 1 & 6 & 1 \\ 5 & 3 & 4 & 4 \\ 3 & 2 & 7 & 4 \\ 2 & 6 & 1 & 1 \end{matrix}$$

Fig. 27.7

Theorem 27.4: In a full-checksum matrix, any single erroneous element can be corrected and any three errors can be detected

27.6 Fault-Tolerant RNS Arithmetic

Residue number systems allow very elegant and effective error detection and correction schemes by means of redundant residues (extra moduli)

Example: RNS(8 | 7 | 5 | 3), Dynamic range $M = 8 \times 7 \times 5 \times 3 = 840$; redundant modulus: 11. Any error confined to a single residue detectable.

Error detection (the redundant modulus must be the largest one, say m):

1. Use other residues to compute the residue of the number mod m (this process is known as *base extension*)
2. Compare the computed and actual mod-m residues

The beauty of this method is that arithmetic algorithms are completely unaffected; error detection is made possible by simply extending the dynamic range of the RNS

Example RNS with two Redundant Residues

RNS(8 | 7 | 5 | 3), with redundant moduli 13 and 11

Representation of 25 = $(12, 3, 1, 4, 0, 1)_{RNS}$

Corrupted version = $(12, 3, 1, 6, 0, 1)_{RNS}$

Transform (–,–,1,6,0,1) to (5,1,1,6,0,1) via base extension

Reconstructed number = $(5, 1, 1, 6, 0, 1)_{RNS}$

The difference between the first two components of the corrupted and reconstructed numbers is (+7, +2)

This constitutes a syndrome, allowing us to correct the error

28 Reconfigurable Arithmetic

Chapter Goals

Examine arithmetic algorithms and designs appropriate for implementation on FPGAs (one-of-a-kind, low-volume, prototype systems)

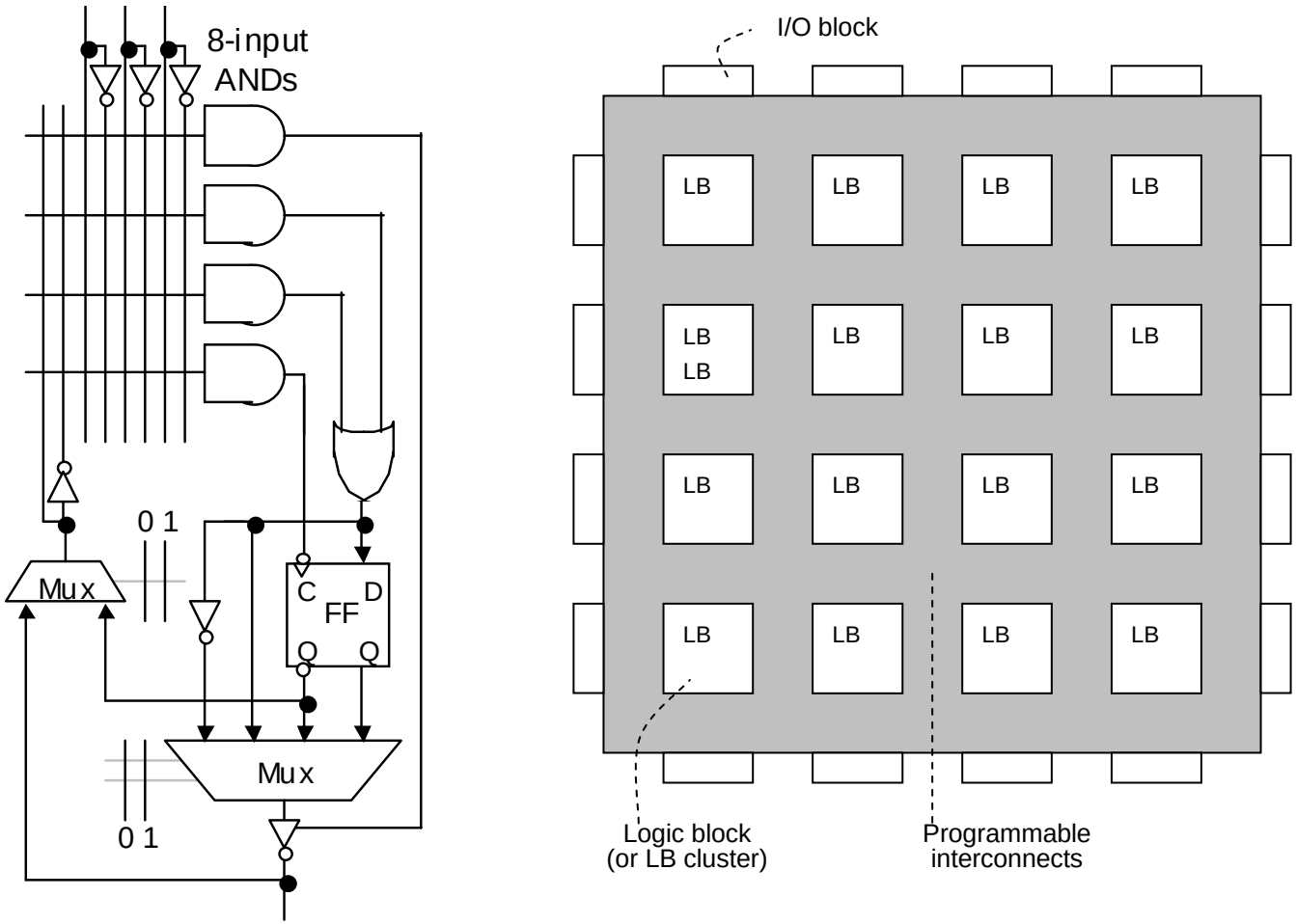
Chapter Highlights

Suitable adder designs beyond ripple-carry
Design choices for multipliers and dividers
Table-based and “distributed” arithmetic
Techniques for function evaluation
Enhanced FPGAs and higher-level alternatives

Reconfigurable Arithmetic: Topics

Topics in This Chapter
28.1 Programmable Logic Devices
28.2 Adder Designs for FPGAs
28.3 Multiplier and Divider Designs
28.4 Tabular and Distributed Arithmetic
28.5 Function Evaluation on FPGAs
28.6 Beyond Fine-Grained Devices

28.1 Programmable Logic Devices



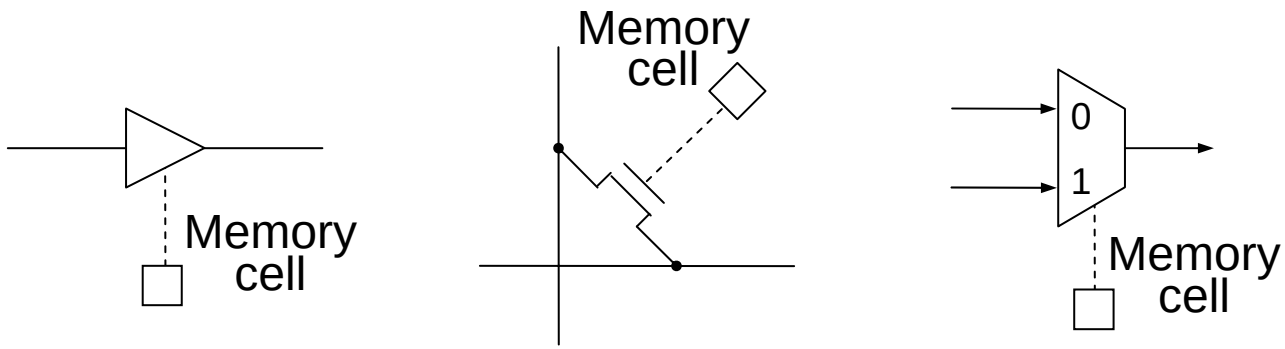
(a) Portion of PAL with storable output

(b) Generic structure of an FPGA

Fig. 28.1 Examples of programmable sequential logic.

Programmability Mechanisms

Slide to be completed



(a) Tristate buffer (b) Pass transistor (c) Multiplexer

Fig. 28.2 Some memory-controlled switches and interconnections in programmable logic devices.

Configurable Logic Blocks

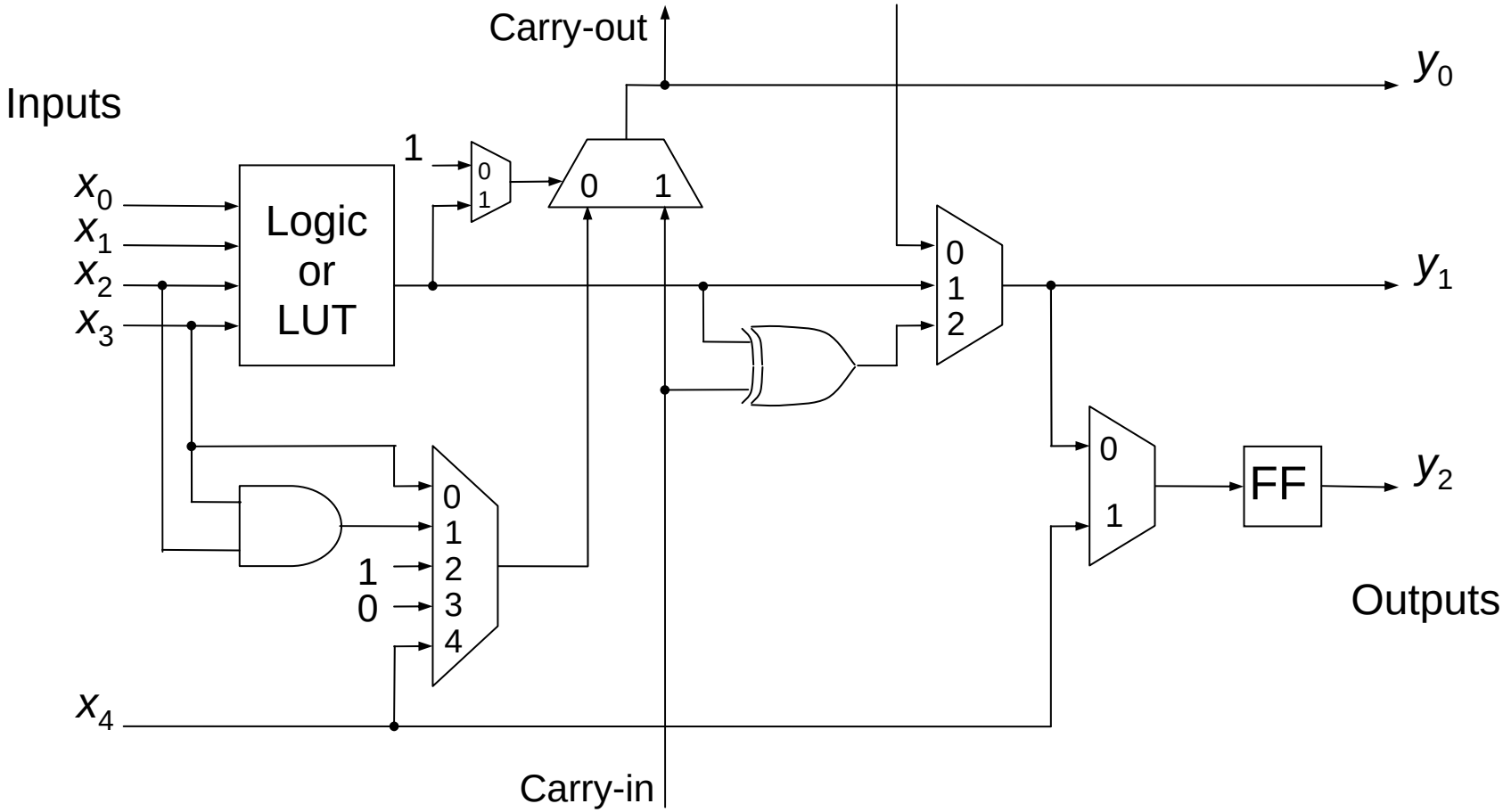


Fig. 28.3 Structure of a simple logic block.

The Interconnect Fabric

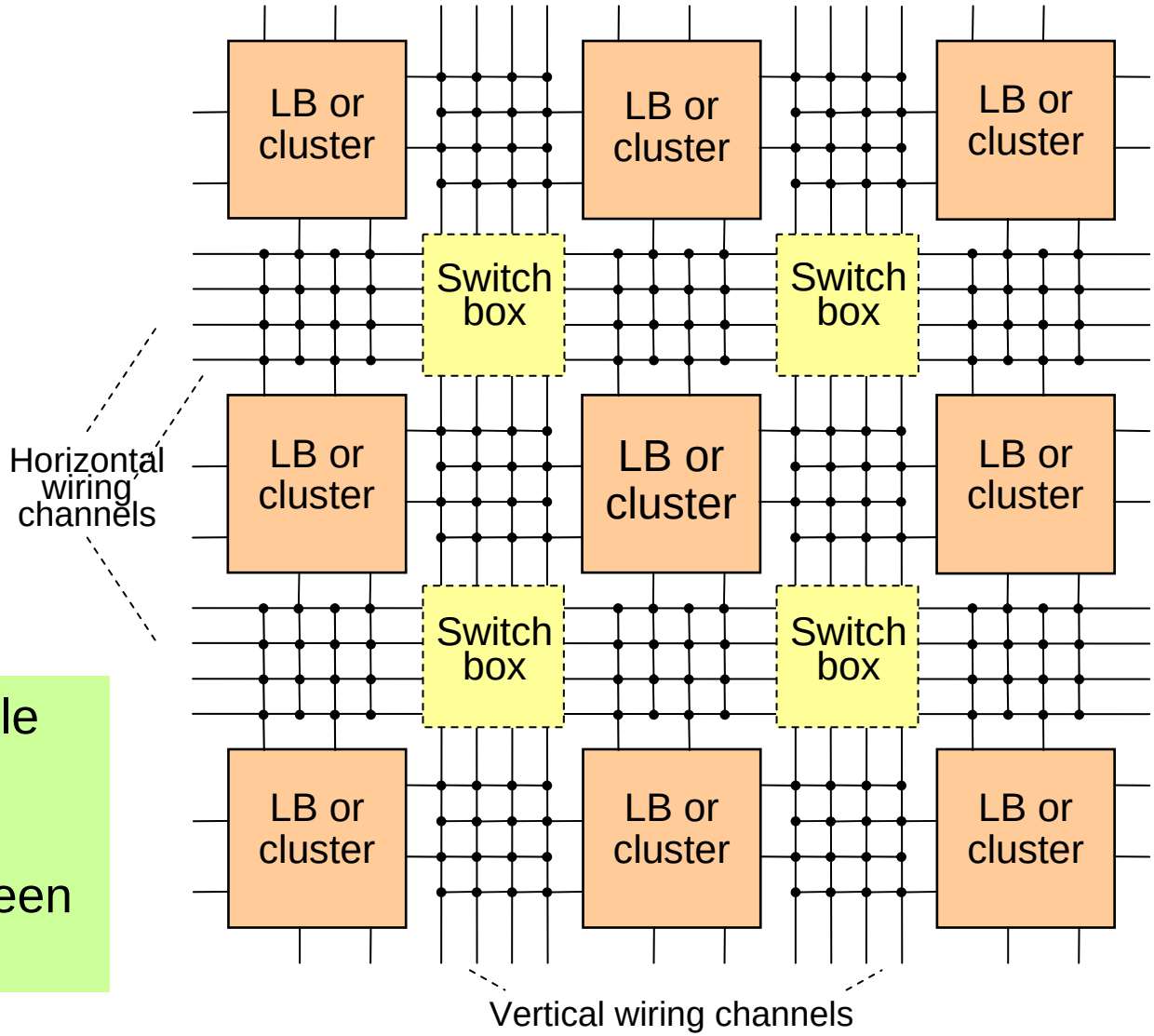


Fig. 28.4 A possible arrangement of programmable interconnects between LBs or LB clusters.

Standard FPGA Design Flow

1. ***Specification:*** Creating the design files, typically via a hardware description language such as Verilog, VHDL, or Abel
2. ***Synthesis:*** Converting the design files into interconnected networks of gates and other standard logic circuit elements
3. ***Partitioning:*** Assigning the logic elements of stage 2 to specific physical circuit elements that are capable of realizing them
4. ***Placement:*** Mapping of the physical circuit elements of stage 3 to specific physical locations of the target FPGA device
5. ***Routing:*** Mapping of the interconnections prescribed in stage 2 to specific physical wires on the target FPGA device
6. ***Configuration:*** Generation of the requisite bit-stream file that holds configuration bits for the target FPGA device
7. ***Programming:*** Uploading the bit-stream file of stage 6 to memory elements within the FPGA device
8. ***Verification:*** Ensuring the correctness of the final design, in terms of both function and timing, via simulation and testing

28.2 Adder Designs for FPGAs

This slide to include a discussion of ripple-carry adders and built-in carry chains in FPGAs

Carry-Skip Addition

Slide to be completed

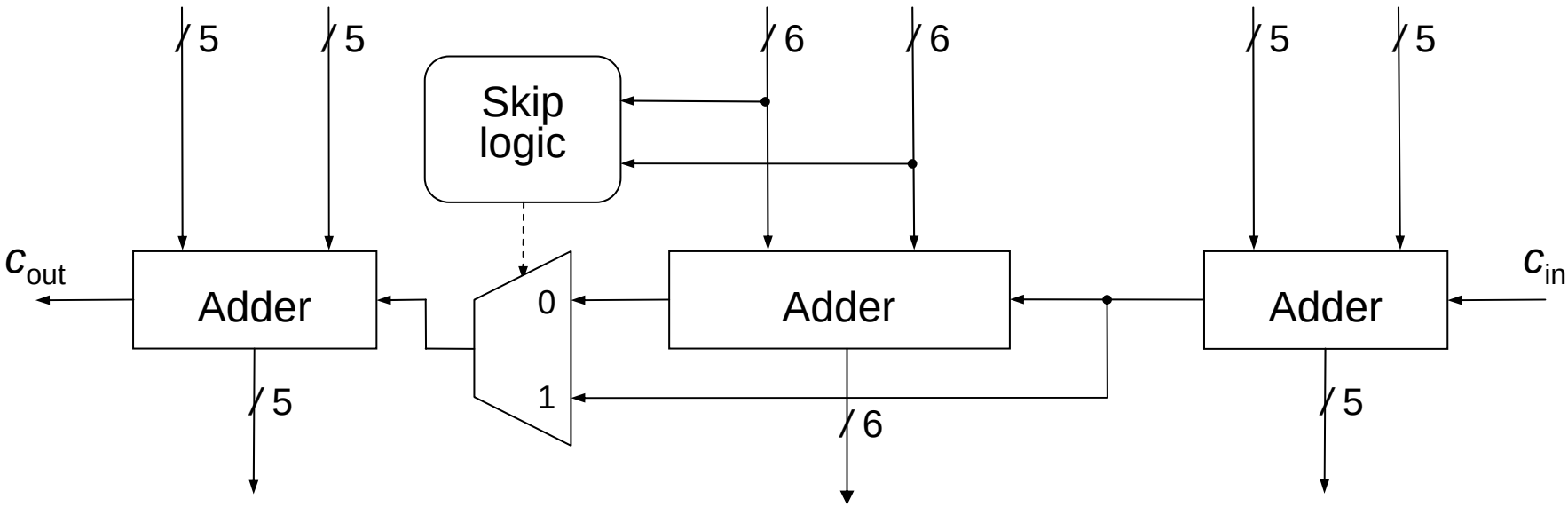


Fig. 28.5 Possible design of a 16-bit carry-skip adder on an FPGA.

Carry-Select Addition

Slide to be completed

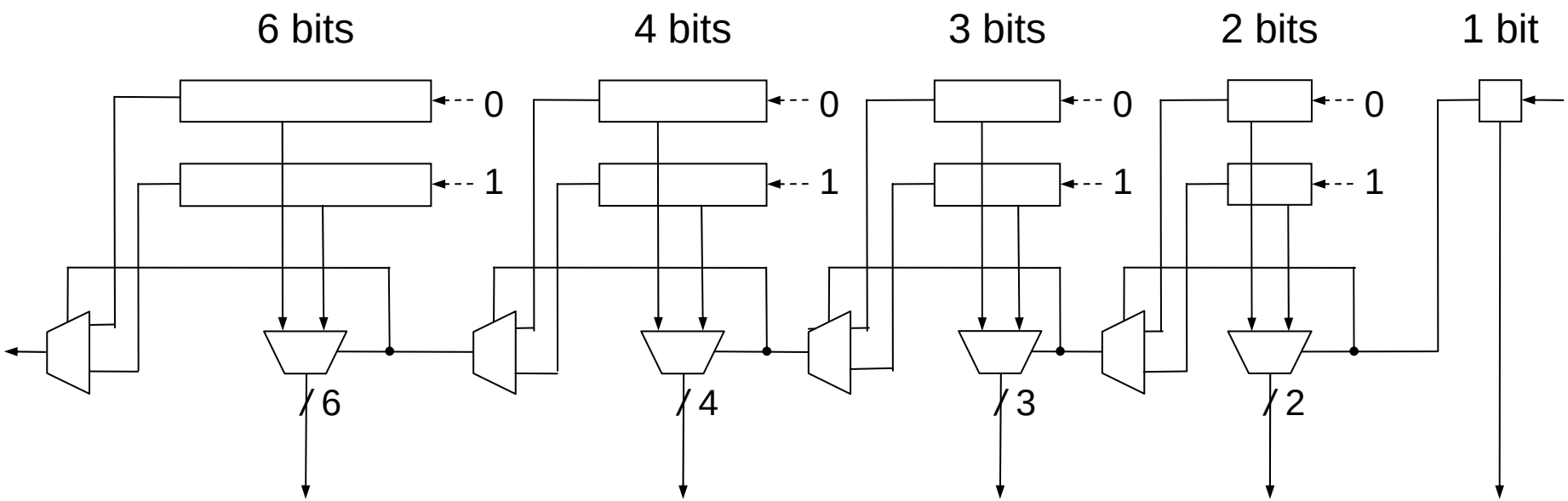


Fig. 28.6 Possible design of a carry-select adder on an FPGA.

28.3 Multiplier and Divider Designs

Slide to be completed

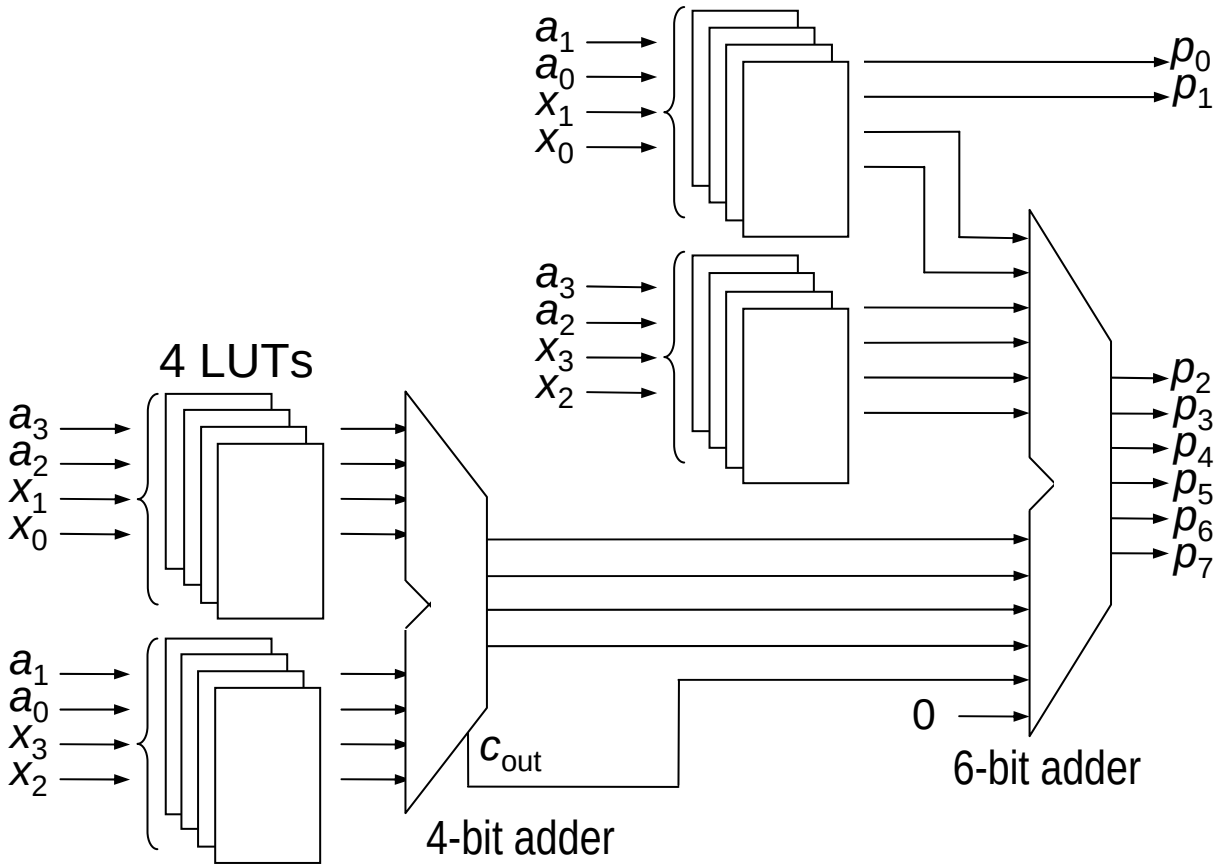


Fig. 28.7 Divide-and-conquer 4 × 4 multiplier design using 4-input lookup tables and ripple-carry adders.

Multiplication by Constants

Slide to be completed

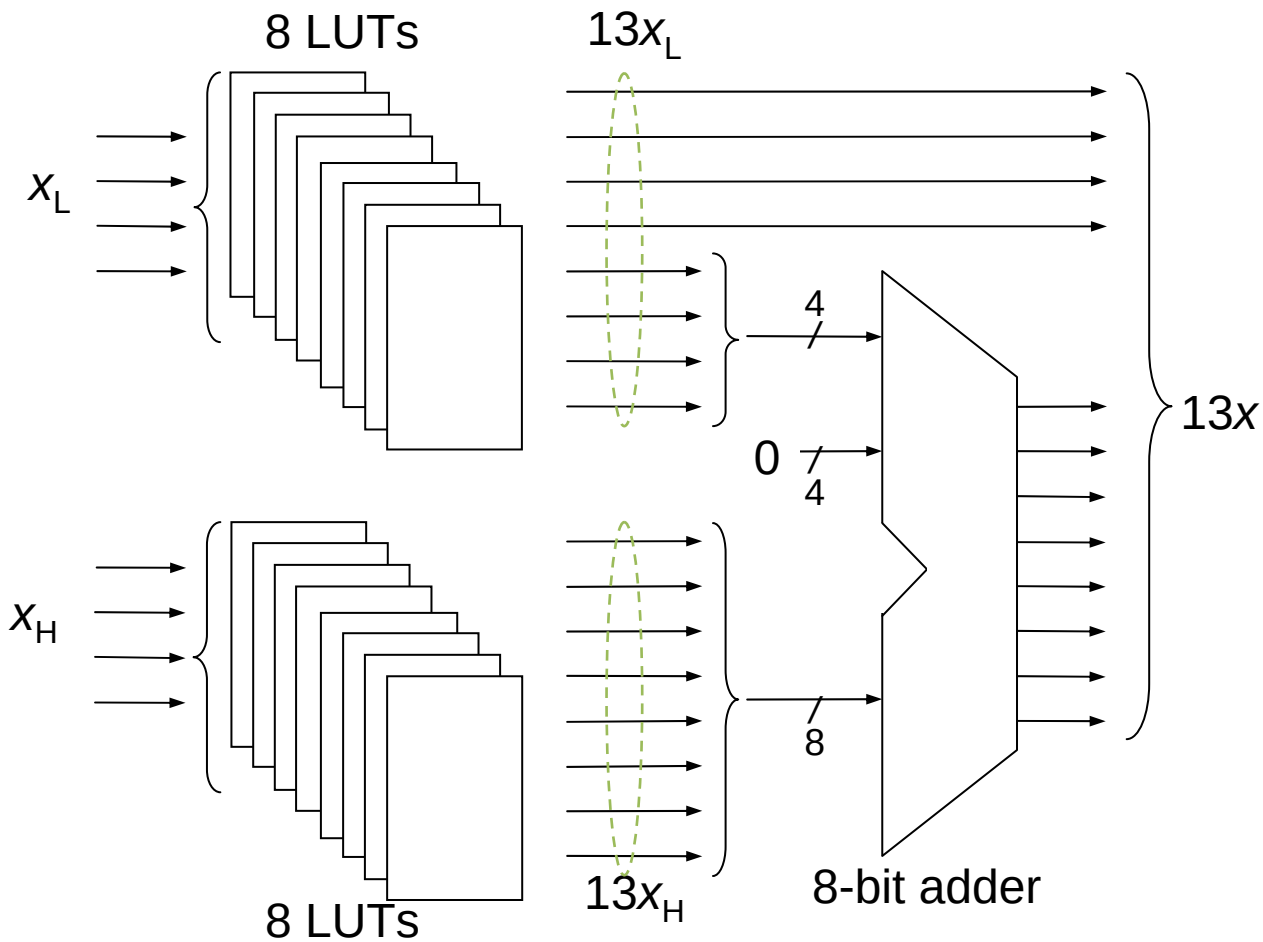


Fig. 28.8 Multiplication of an 8-bit input by 13, using LUTs.

Division on FPGAs

Slide to be completed

28.4 Tabular and Distributed Arithmetic

Slide to be completed

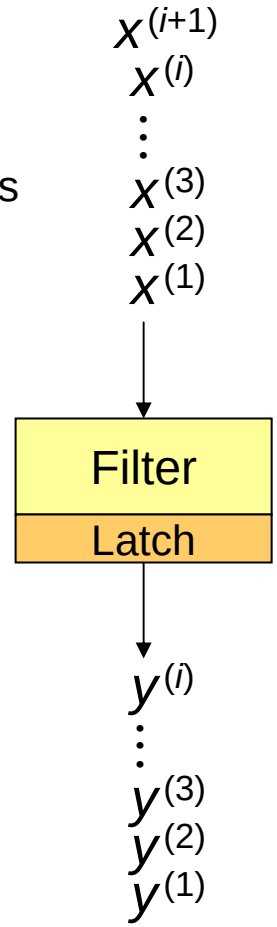
Second-Order Digital Filter: Definition

$$y^{(i)} = a^{(0)}x^{(i)} + a^{(1)}x^{(i-1)} + a^{(2)}x^{(i-2)} - b^{(1)}y^{(i-1)} - b^{(2)}y^{(i-2)}$$

Current and two previous inputs

Two previous outputs

$a^{(j)}$s and $b^{(j)}$s are constants



Expand the equation for $y^{(i)}$ in terms of the bits in operands $x = (x_0.x_{-1}x_{-2} \ldots x_{-l})_{2\text{'s-compl}}$ and $y = (y_0.y_{-1}y_{-2} \ldots y_{-l})_{2\text{'s-compl}}$, where the summations range from $j = -l$ to $j = -1$

$$y^{(i)} = a^{(0)}(-x_0^{(i)} + \sum 2^j x_j^{(i)})$$
$$+ a^{(1)}(-x_0^{(i-1)} + \sum 2^j x_j^{(i-1)}) + a^{(2)}(-x_0^{(i-2)} + \sum 2^j x_j^{(i-2)})$$
$$- b^{(1)}(-y_0^{(i-1)} + \sum 2^j y_j^{(i-1)}) - b^{(2)}(-y_0^{(i-2)} + \sum 2^j y_j^{(i-2)})$$

Define $f(s, t, u, v, w) = a^{(0)}s + a^{(1)}t + a^{(2)}u - b^{(1)}v - b^{(2)}w$

$$y^{(i)} = \sum 2^j f(x_j^{(i)}, x_j^{(i-1)}, x_j^{(i-2)}, y_j^{(i-1)}, y_j^{(i-2)}) - f(x_0^{(i)}, x_0^{(i-1)}, x_0^{(i-2)}, y_0^{(i-1)}, y_0^{(i-2)})$$

Second-Order Digital Filter: Bit-Serial Implementation

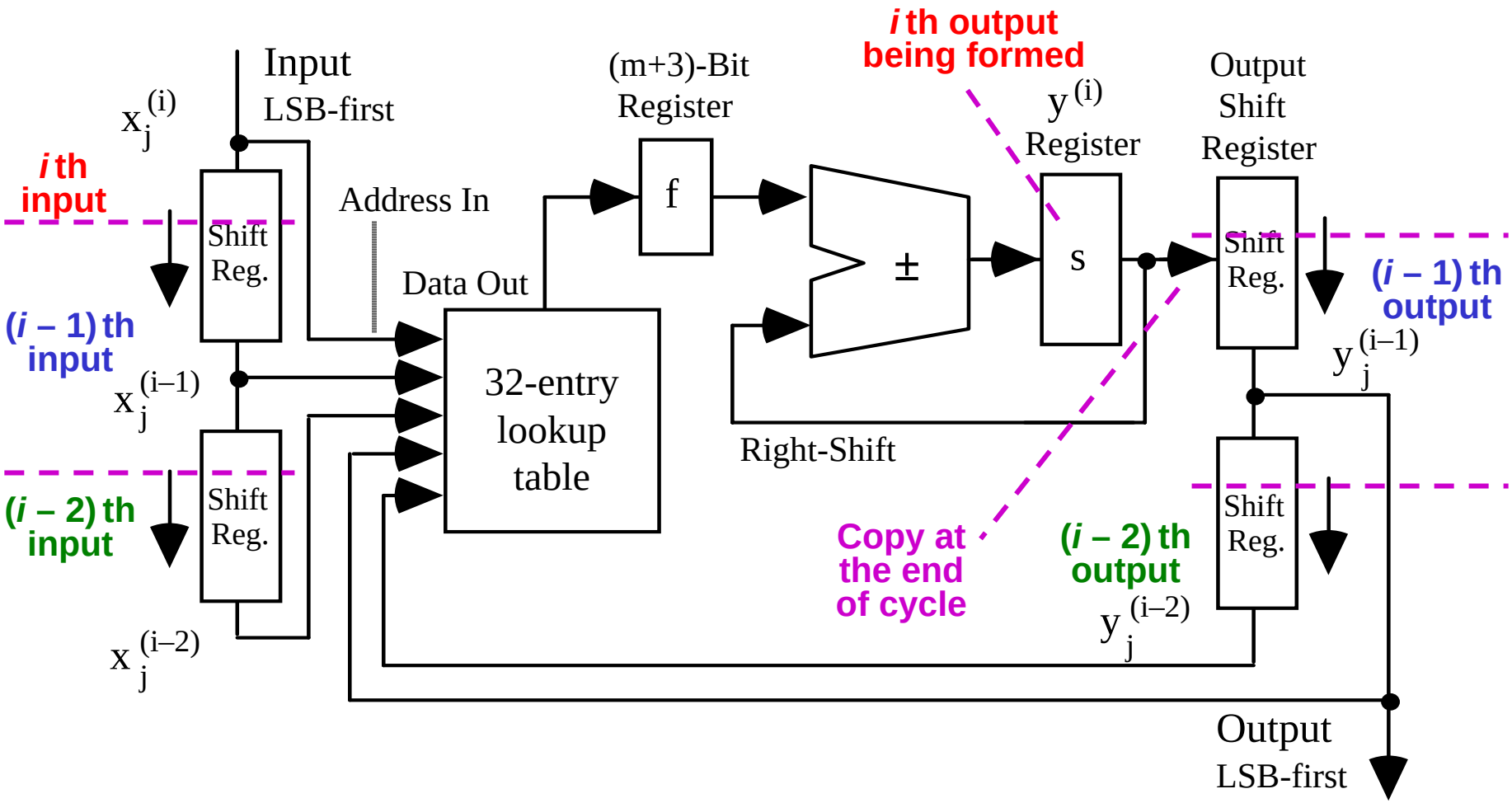


Fig. 28.9 Bit-serial tabular realization of a second-order filter.

28.5 Function Evaluation on FPGAs

Slide to be completed

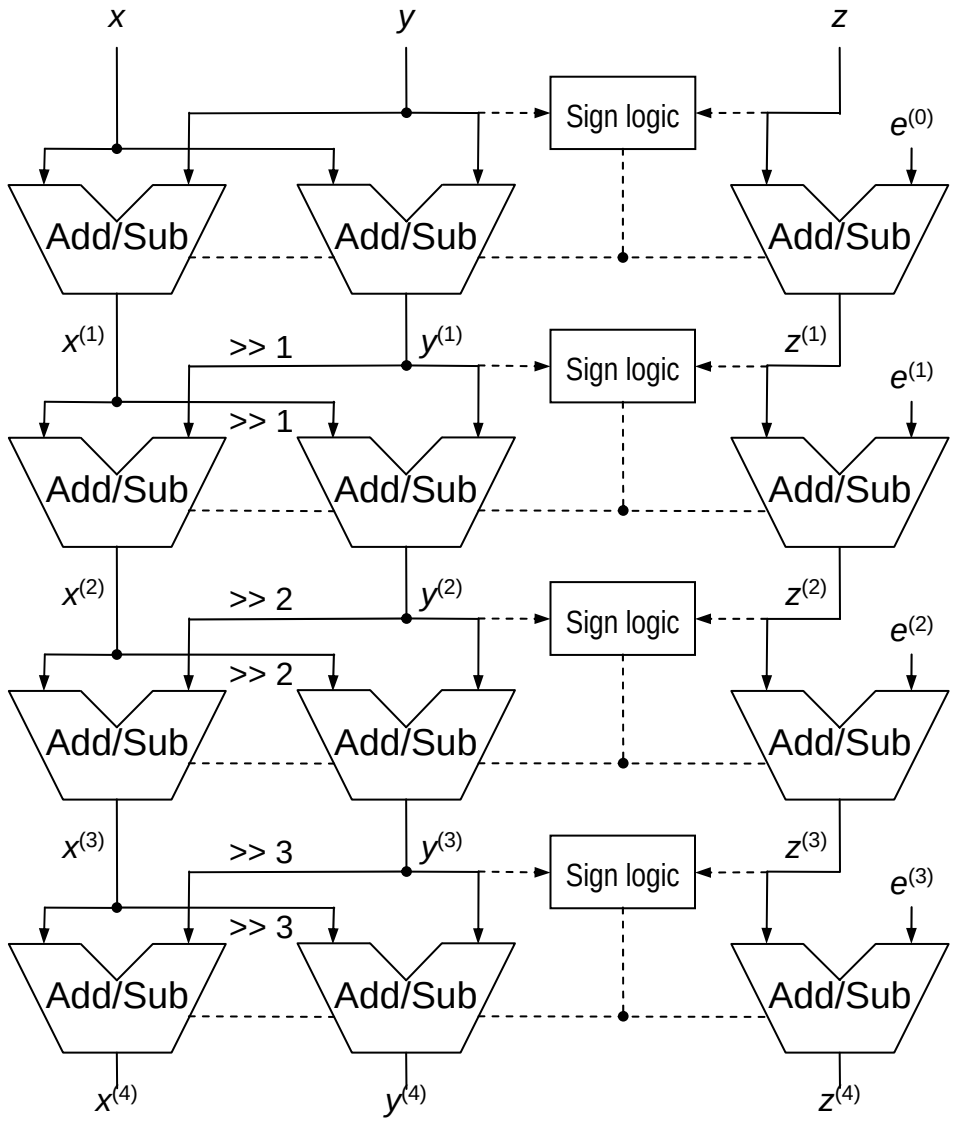


Fig. 28.10 The first four stages of an unrolled CORDIC processor.

Implementing Convergence Schemes

Slide to be completed

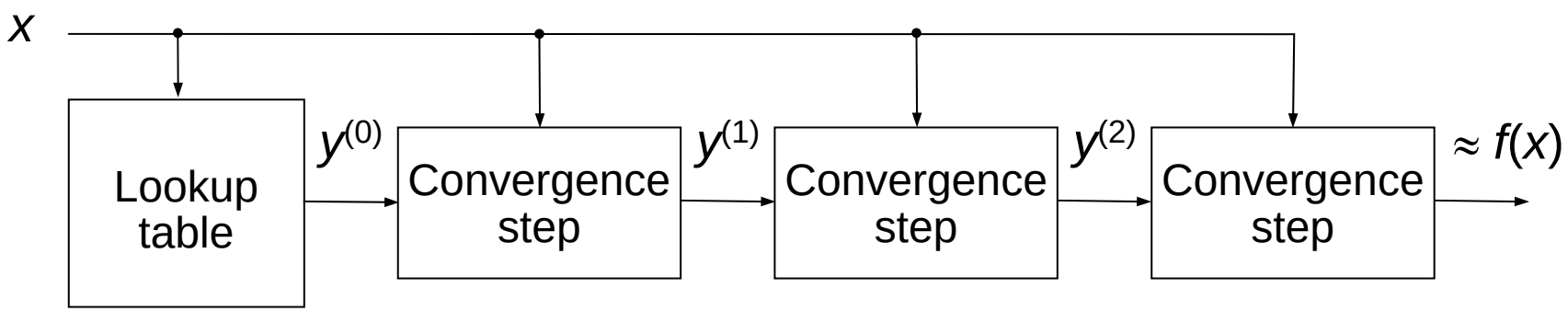


Fig. 28.11 Generic convergence structure for function evaluation.

28.6 Beyond Fine-Grained Devices

Slide to be completed

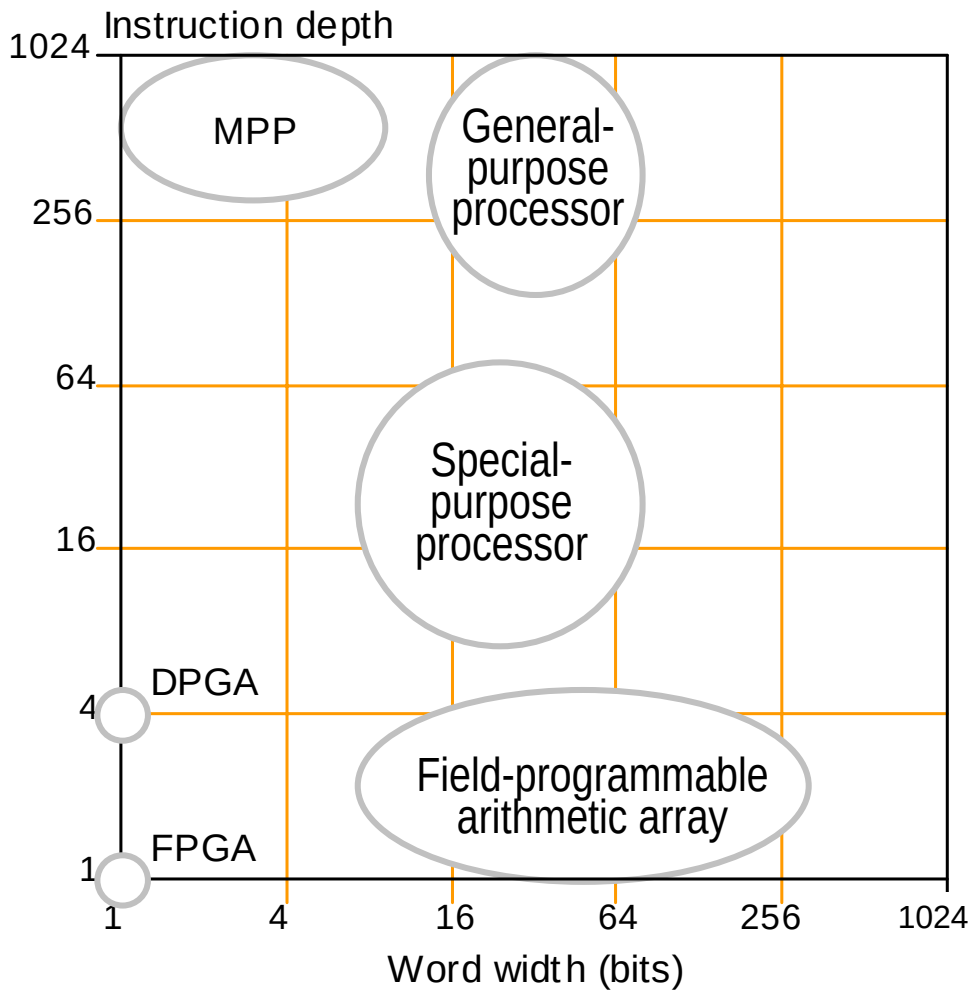


Fig. 28.12 The design space for arithmetic-intensive applications.

A Past, Present, and Future

Appendix Goals

Wrap things up, provide perspective, and examine arithmetic in a few key systems

Appendix Highlights

One must look at arithmetic in context of

- Computational requirements
- Technological constraints
- Overall system design goals
- Past and future developments

Current trends and research directions?

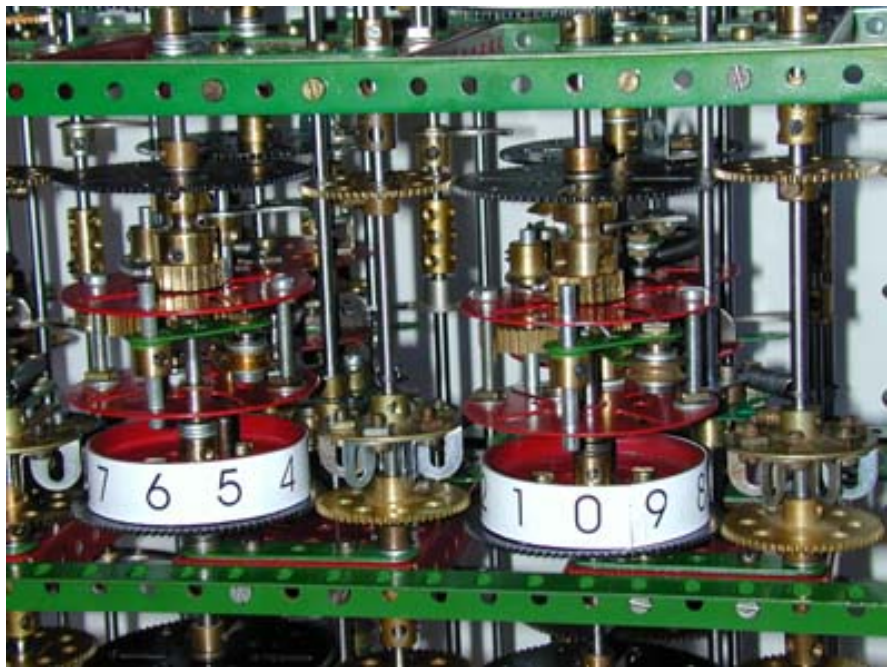
Past, Present, and Future: Topics

Topics in This Chapter
A.1 Historical Perspective
A.2 Early High-Performance Computers
A.3 Deeply Pipelined Vector Machines
A.4 The DSP Revolution
A.5 Supercomputers on Our Laps
A.6 Trends, Outlook, and Resources

A.1 Historical Perspective

Babbage was aware of ideas such as carry-skip addition, carry-save addition, and restoring division

Modern reconstruction from Meccano parts;
http://www.meccano.us/difference_engines/



Computer Arithmetic in the 1940s

Machine arithmetic was crucial in proving the feasibility of computing with stored-program electronic devices

Hardware for addition/subtraction, use of complement representation, and shift-add multiplication and division algorithms were developed and fine-tuned

A seminal report by A.W. Burkes, H.H. Goldstein, and J. von Neumann contained ideas on choice of number radix, carry propagation chains, fast multiplication via carry-save addition, and restoring division

State of computer arithmetic circa 1950:

Overview paper by R.F. Shaw [Shaw50]

Computer Arithmetic in the 1950s

The focus shifted from feasibility to algorithmic speedup methods and cost-effective hardware realizations

By the end of the decade, virtually all important fast-adder designs had already been published or were in the final phases of development

Residue arithmetic, SRT division, CORDIC algorithms were proposed and implemented

Snapshot of the field circa 1960:

Overview paper by O.L. MacSorley [MacS61]

Computer Arithmetic in the 1960s

Tree multipliers, array multipliers, high-radix dividers, convergence division, redundant signed-digit arithmetic were introduced

Implementation of floating-point arithmetic operations in hardware or firmware (in microprogram) became prevalent

Many innovative ideas originated from the design of early supercomputers, when the demand for high performance, along with the still high cost of hardware, led designers to novel and cost-effective solutions

Examples reflecting the sate of the art near the end of this decade:

- IBM's System/360 Model 91 [Ande67]
- Control Data Corporation's CDC 6600 [Thor70]

Computer Arithmetic in the 1970s

Advent of microprocessors and vector supercomputers

Early LSI chips were quite limited in the number of transistors or logic gates that they could accommodate

Microprogrammed control (with just a hardware adder) was a natural choice for single-chip processors which were not yet expected to offer high performance

For high end machines, pipelining methods were perfected to allow the throughput of arithmetic units to keep up with computational demand in vector supercomputers

Examples reflecting the state of the art near the end of this decade:

Cray 1 supercomputer and its successors

Computer Arithmetic in the 1980s

Spread of VLSI triggered a reconsideration of all arithmetic designs in light of interconnection cost and pin limitations

For example, carry-lookahead adders, thought to be ill-suited to VLSI, were shown to be efficiently realizable after suitable modifications. Similar ideas were applied to more efficient VLSI tree and array multipliers

Bit-serial and on-line arithmetic were advanced to deal with severe pin limitations in VLSI packages

Arithmetic-intensive signal processing functions became driving forces for low-cost and/or high-performance embedded hardware: DSP chips

Computer Arithmetic in the 1990s

No breakthrough design concept

Demand for performance led to fine-tuning of arithmetic algorithms and implementations (many hybrid designs)

Increasing use of table lookup and tight integration of arithmetic unit and other parts of the processor for maximum performance

Clock speeds reached and surpassed 100, 200, 300, 400, and 500 MHz in rapid succession; pipelining used to ensure smooth flow of data through the system

Examples reflecting the state of the art near the end of this decade:

- Intel's Pentium Pro (P6) $\rightarrow$ Pentium II
- Several high-end DSP chips

Computer Arithmetic in the 2000s

Three parallel and interacting trends:

Availability of many millions of transistors on a single microchip

Energy requirements and heat dissipation of the said transistors

Shift of focus from scientific computations to media processing

Continued refinement of many existing methods, particularly those based on table lookup

New challenges posed by multi-GHz clock rates

Increased emphasis on low-power design

Work on, and approval of, the IEEE 754-2008 floating-point standard

A.2 Early High-Performance Computers

IBM System 360 Model 91 (360/91, for short; mid 1960s)

Part of a family of machines with the same instruction-set architecture

Had multiple function units and an elaborate scheduling and interlocking hardware algorithm to take advantage of them for high performance

Clock cycle = 20 ns (quite aggressive for its day)

Used 2 concurrently operating floating-point execution units performing:

- Two-stage pipelined addition
- 12×56 pipelined partial-tree multiplication
- Division by repeated multiplications (initial versions of the machine sometimes yielded an incorrect LSB for the quotient)

The IBM System 360 Model 91

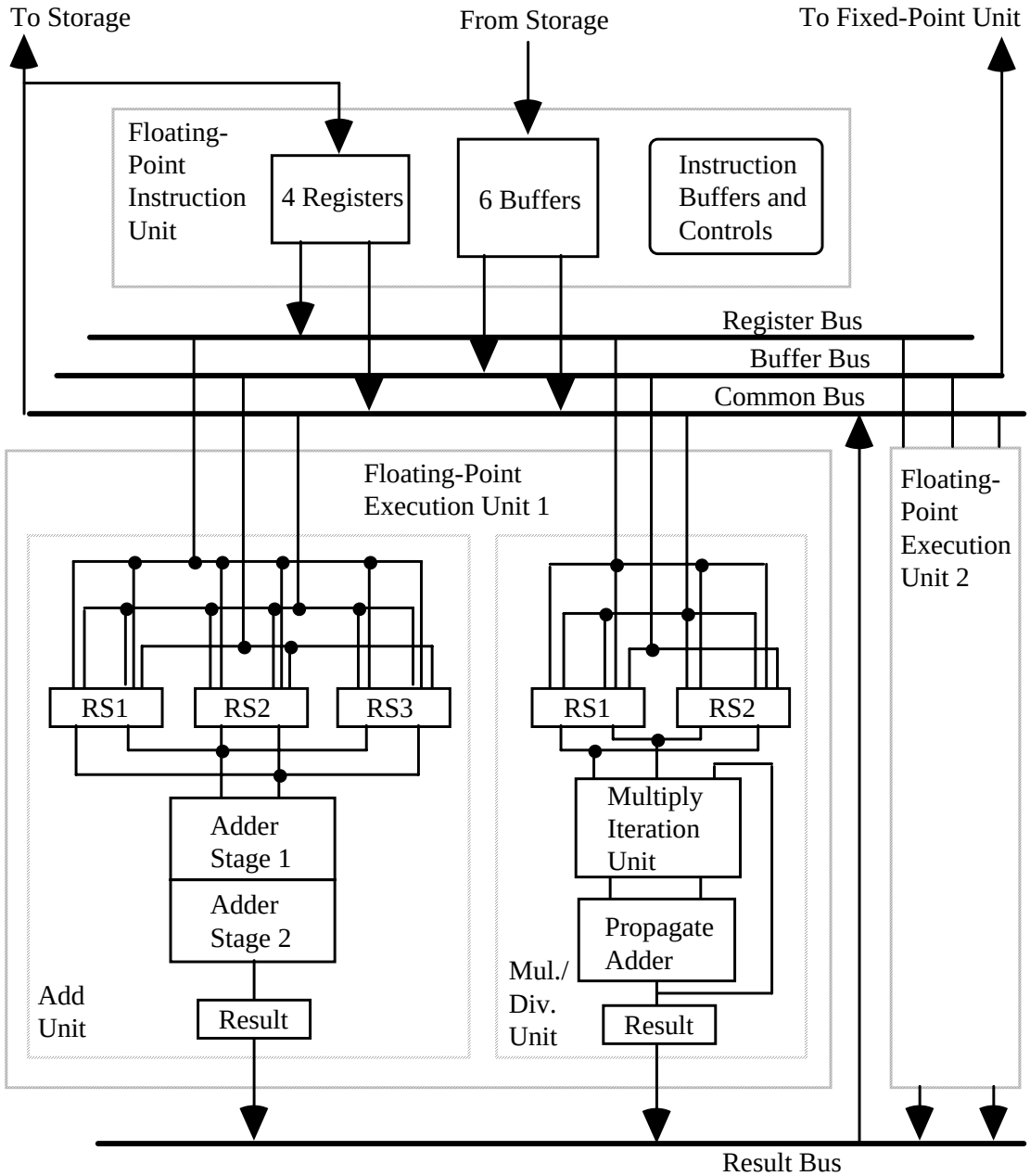


Fig. A.1 Overall structure of the IBM System/360 Model 91 floating-point execution unit.

A.3 Deeply Pipelined Vector Machines

Cray X-MP/Model 24 (multiple-processor vector machine)

Had multiple function units, each of which could produce a new result on every clock tick, given suitably long vectors to process

Clock cycle = 9.5 ns

Used 5 integer/logic function units and 3 floating-point function units

- Integer/Logic units: add, shift, logical 1, logical 2, weight/parity
- Floating-point units: add (6 stages), multiply (7 stages), reciprocal approximation (14 stages)

Pipeline setup and shutdown overheads

- Vector unit not efficient for short vectors (break-even point)
- Pipeline chaining

Cray X-MP Vector Computer

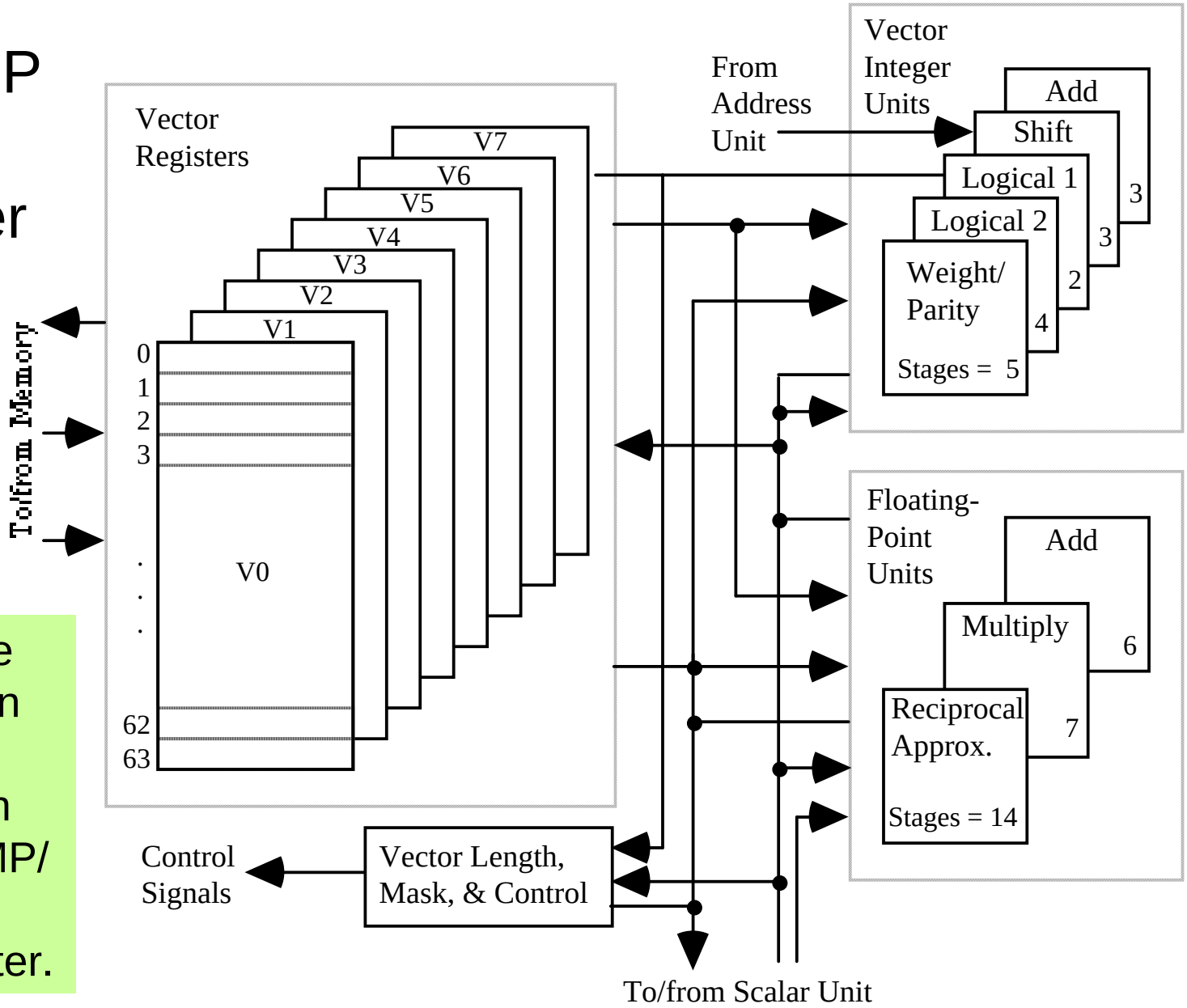


Fig. A.2 The vector section of one of the processors in the Cray X-MP/ Model 24 supercomputer.

A.4 The DSP Revolution

Special-purpose DSPs have used a wide variety of unconventional arithmetic methods; e.g., RNS or logarithmic number representation

General-purpose DSPs provide an instruction set that is tuned to the needs of arithmetic-intensive signal processing applications

Example DSP instructions

ADD	A, B	$\{ A + B \rightarrow B \}$
SUB	X, A	$\{ A - X \rightarrow A \}$
MPY	$\pm$X1, X0, B	$\{ \pm X1 \times X0 \rightarrow B \}$
MAC	$\pm$Y1, X1, A	$\{ A \pm Y1 \times X1 \rightarrow A \}$
AND	X1, A	$\{ A \text{ AND } X1 \rightarrow A \}$

General-purpose DSPs come in integer and floating-point varieties

Fixed-Point DSP Example

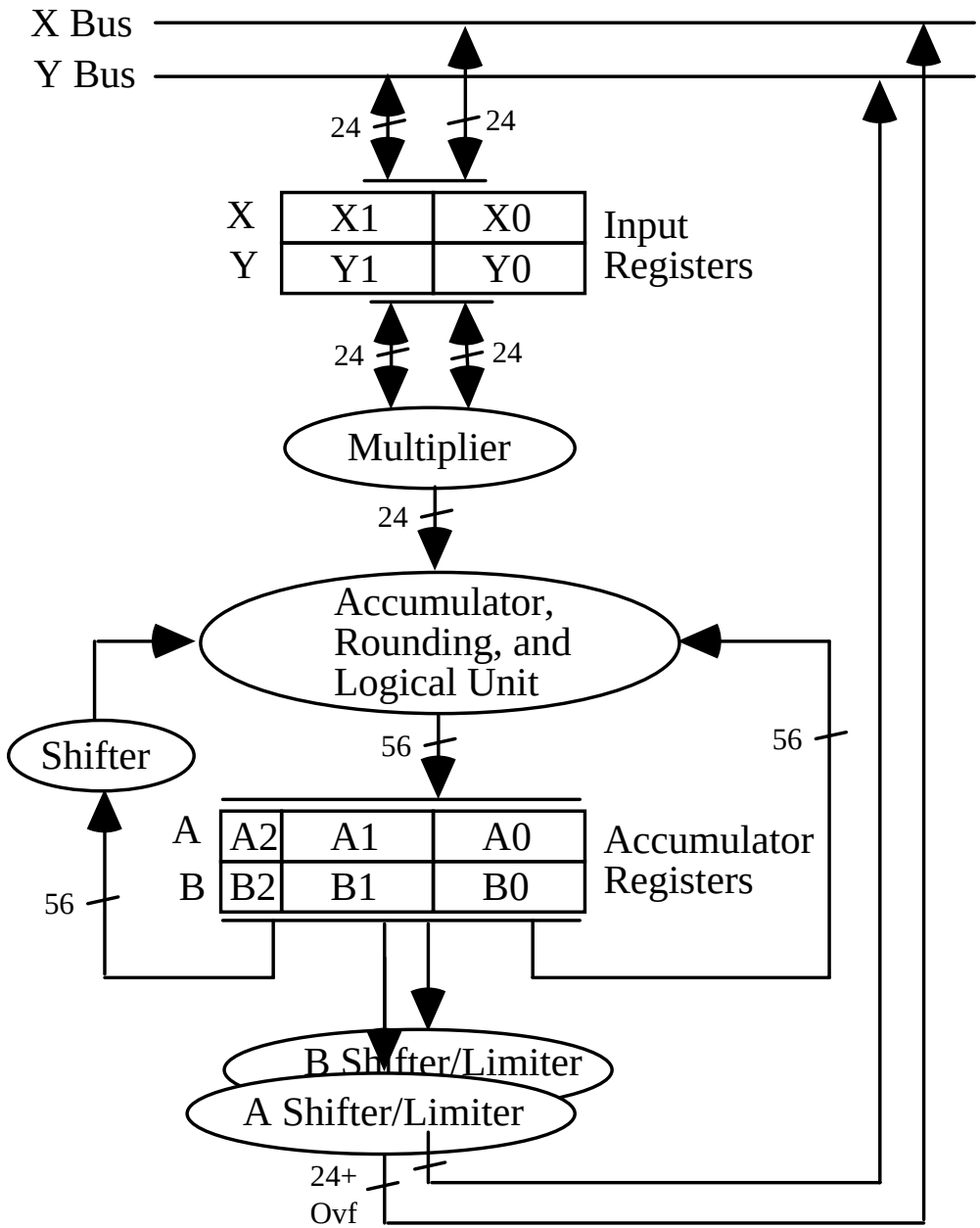


Fig. A.3 Block diagram of the data ALU in Motorola's DSP56002 (fixed-point) processor.

Floating-Point DSP Example

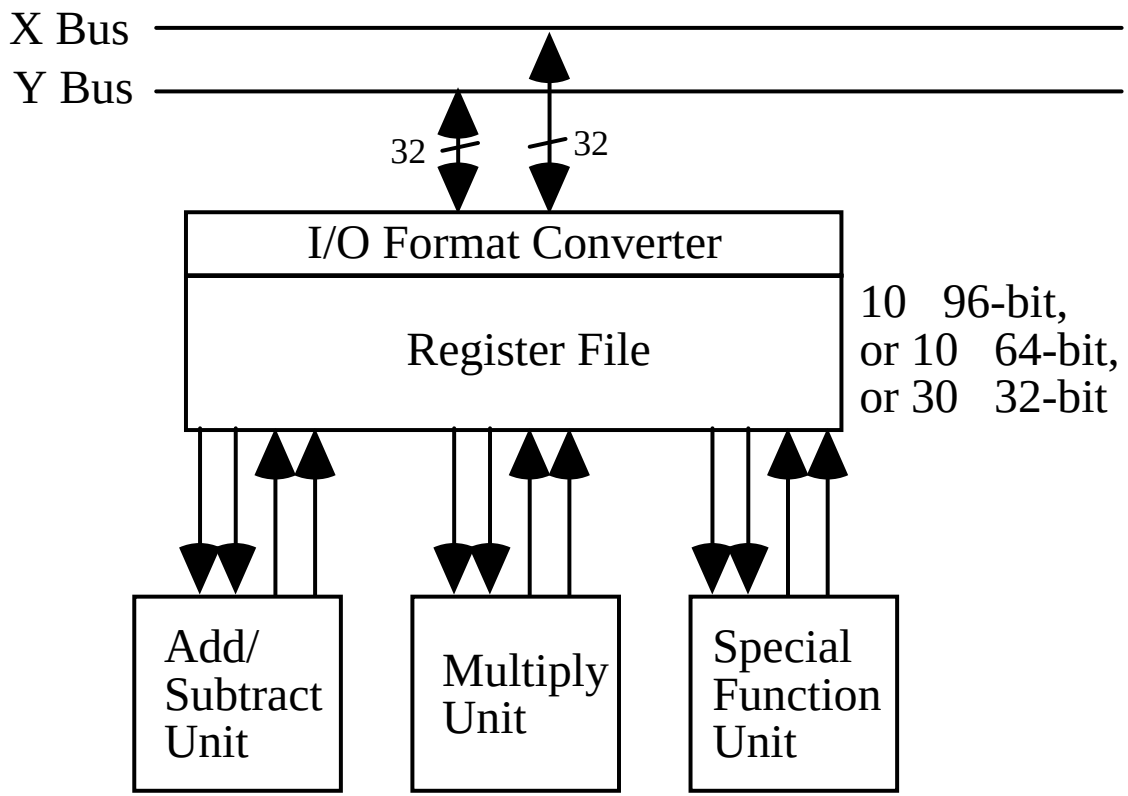


Fig. A.4 Block diagram of the data ALU in Motorola's DSP96002 (floating-point) processor.

A.5 Supercomputers on Our Laps

In the beginning, there was the 8080; led to the 80x86 = IA32 ISA

Half a dozen or so pipeline stages

80286
80386
80486
Pentium (80586)

More advanced technology

A dozen or so pipeline stages, with out-of-order instruction execution

Pentium Pro
Pentium II
Pentium III
Celeron

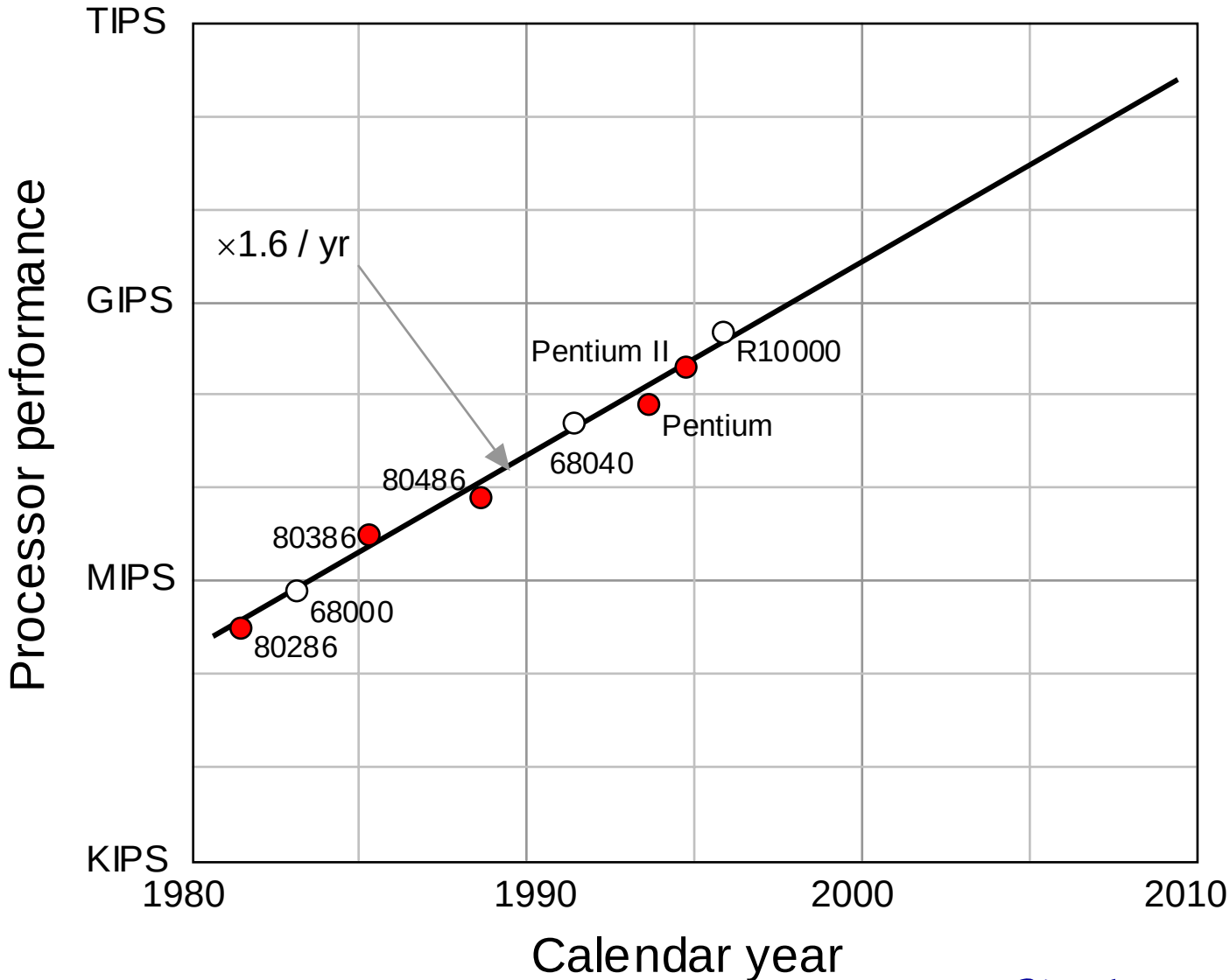
More advanced technology

Instructions are broken into micro-ops which are executed out-of-order but retired in-order

Two dozens or so pipeline stages

Pentium 4

Performance Trends in Intel Microprocessors



Arithmetic in the Intel Pentium Pro Microprocessor

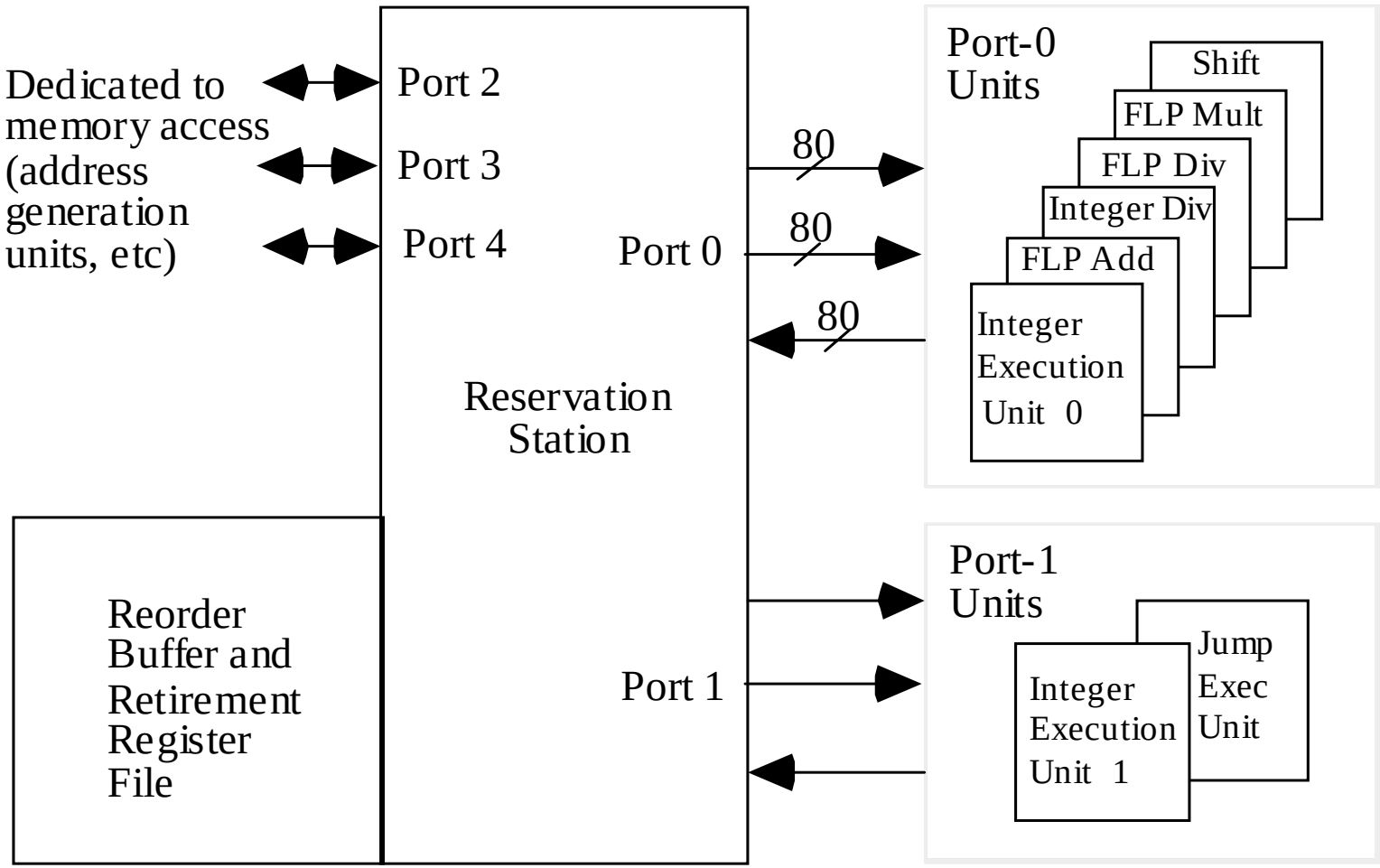


Fig. 28.5 Key parts of the CPU in the Intel Pentium Pro (P6) microprocessor.

A.6 Trends, Outlook, and Resources

Current focus areas in computer arithmetic

Design: Shift of attention from algorithms to optimizations at the level of transistors and wires

This explains the proliferation of hybrid designs

Technology: Predominantly CMOS, with a phenomenal rate of improvement in size/speed

New technologies cannot compete

Applications: Shift from high-speed or high-throughput designs in mainframes to embedded systems requiring

Low cost
Low power

Ongoing Debates and New Paradigms

Renewed interest in bit- and digit-serial arithmetic as mechanisms to reduce the VLSI area and to improve packageability and testability

Synchronous vs asynchronous design (asynchrony has some overhead, but an equivalent overhead is being paid for clock distribution and/or systolization)

New design paradigms may alter the way in which we view or design arithmetic circuits

- Neuronlike computational elements
- Optical computing (redundant representations)
- Multivalued logic (match to high-radix arithmetic)
- Configurable logic

Arithmetic complexity theory

Computer Arithmetic Timeline

Snapshot	Decade	Key ideas, innovations, advancements, technology traits, and milestones
	1940	
[Burk46] [Shaw50]	40s	Binary format, carry chains, stored carry, carry-save multiplier, restoring divider
	50s	Carry-lookahead adder, high-radix multiplier, SRT divider, CORDIC algorithms
	1960	
[MacS61]	60s	Tree/array multiplier, high-radix & convergence dividers, signed-digit, floating point
[Ande67] [Thor70]	70s	Pipelined arithmetic, vector supercomputer, microprocessor, ARITH-2/3/4 symposia
[Garn76]]	1980	
	80s	VLSI, embedded system, digital signal processor, on-line arithmetic, IEEE 754-1985
[Swar90]	90s	CMOS dominance, circuit-level optimization, hybrid design, deep pipeline, table lookup
	2000	
	00s	Power/energy/heat reduction, media processing, FPGA-based arith., IEEE 754-2008
[Swar09]	10s	Teraflops on laptop (or pocket device?), asynchronous design, nanodevice arithmetic
	2020	

Fig. A.6 Computer arithmetic through the decades.

The End!

You're up to date. Take my advice and try to keep it that way. It'll be tough to do; make no mistake about it. The phone will ring and it'll be the administrator — talking about budgets. The doctors will come in, and they'll want this bit of information and that. Then you'll get the salesman. Until at the end of the day you'll wonder what happened to it and what you've accomplished; what you've achieved.

That's the way the next day can go, and the next, and the one after that. Until you find a year has slipped by, and another, and another. And then suddenly, one day, you'll find everything you knew is out of date. That's when it's too late to change.

Listen to an old man who's been through it all, who made the mistake of falling behind. Don't let it happen to you! Lock yourself in a closet if you have to! Get away from the phone and the files and paper, and read and learn and listen and keep up to date. Then they can never touch you, never say, "He's finished, all washed up; he belongs to yesterday."

Arthur Hailey, *The Final Diagnosis*