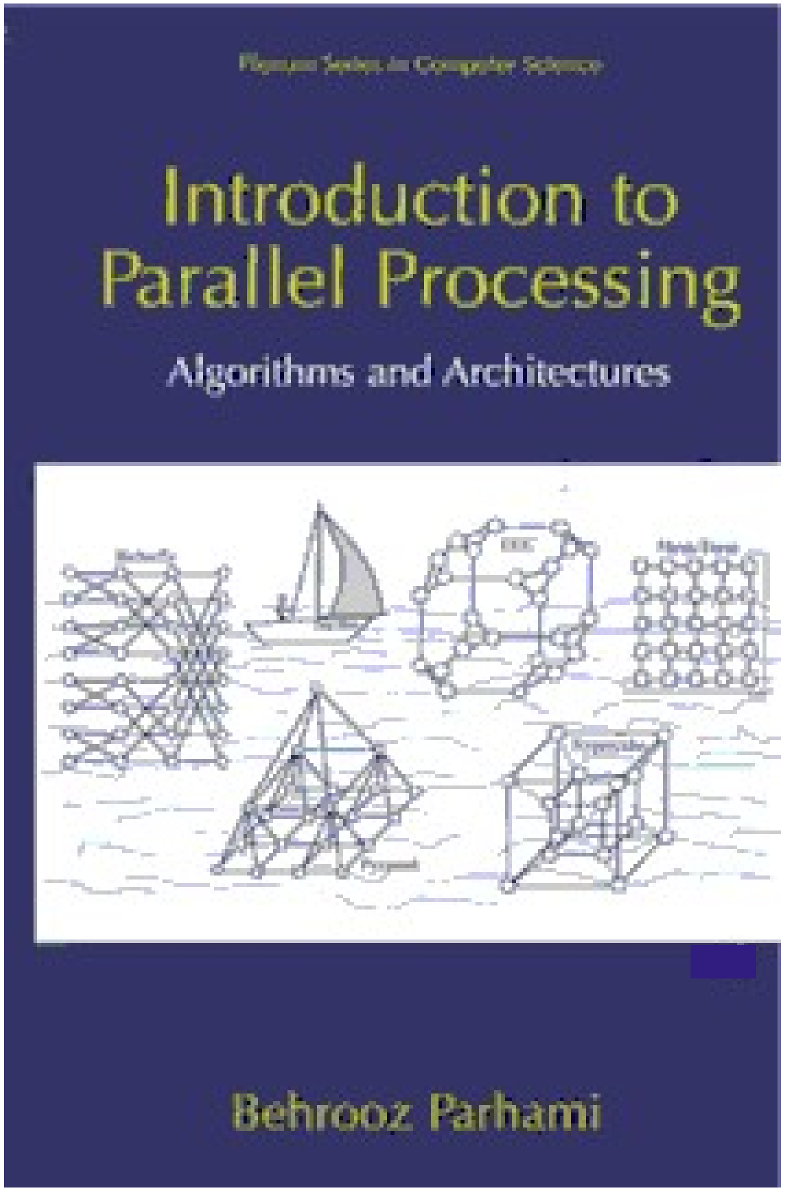

Plenum Series in Computer Science

Introduction to Parallel Processing

Algorithms and Architectures

Behrooz Parhami

Part II″
Circuit-Level Parallelism

	Part I: Fundamental Concepts	Background and Motivation	1. Introduction to Parallelism 2. A Taste of Parallel Algorithms
		Complexity and Models	3. Parallel Algorithm Complexity 4. Models of Parallel Processing
Architectural Variations	Part II″ Circuit-Level Parallelism	Sorting and Searching	7. Sorting and Selection Networks 8A. Search Acceleration Circuits
		Numerical Computations	6B. Arithmetic and Counting Circuits 6C. Fourier Transform Circuits
	Part III: Mesh-Based Architectures	Data Movement on 2D Arrays	9. Sorting on a 2D Mesh or Torus 10. Routing on a 2D Mesh or Torus
		Mesh Algorithms and Variants	11. Numerical 2D Mesh Algorithms 12. Other Mesh-Related Architectures
	Part IV: Low-Diameter Architectures	The Hypercube Architecture	13. Hypercubes and Their Algorithms 14. Sorting and Routing on Hypercubes
		Hypercubic and Other Networks	15. Other Hypercubic Architectures 16. A Sampler of Other Networks
	Part V: Some Broad Topics	Coordination and Data Access	17. Emulation and Scheduling 18. Data Storage, Input, and Output
		Robustness and Ease of Use	19. Reliable Parallel Processing 20. System and Software Issues
	Part VI: Implementation Aspects	Control-Parallel Systems	21. Shared-Memory MIMD Machines 22. Message-Passing MIMD Machines
		Data Parallelism and Conclusion	23. Data-Parallel SIMD Machines 24. Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Introduction to Parallel Processing: Algorithms and Architectures* (Plenum Press, 1999, ISBN 0-306-45970-1). It was prepared by the author in connection with teaching the graduate-level course ECE 254B: Advanced Computer Architecture: Parallel Processing, at the University of California, Santa Barbara. Instructors can use these slides in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

Edition	Released	Revised	Revised	Revised
First	Spring 2005	Spring 2006	Fall 2008	Fall 2010
		Winter 2013	Winter 2014	Winter 2016
		Winter 2019	Winter 2020	Winter 2021

II″ Circuit-Level Parallelism

Circuit-level specs: most realistic parallel computation model

- Concrete circuit model; incorporates hardware details
- Allows realistic speed and cost comparisons
- Useful for stand-alone systems or acceleration units

Topics in This Part	
Chapter 7	Sorting and Selection Networks
Chapter 8A	Search Acceleration Circuits
Chapter 8B	Arithmetic and Counting Circuits
Chapter 8C	Fourier Transform Circuits

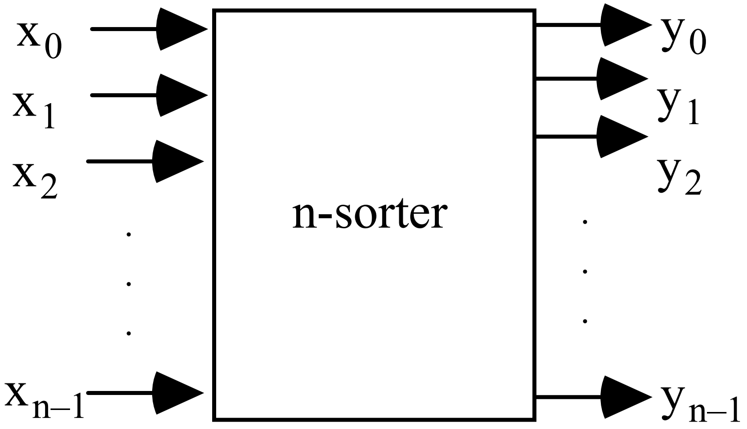
7 Sorting and Selection Networks

Become familiar with the circuit model of parallel processing:

- Go from algorithm to architecture, not vice versa
- Use a familiar problem to study various trade-offs

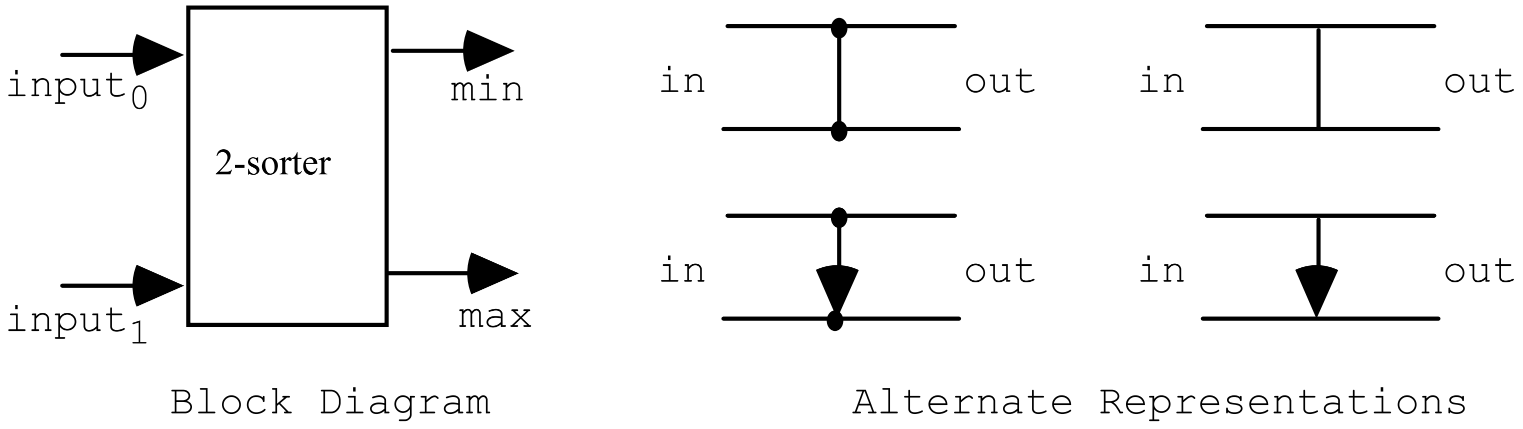
Topics in This Chapter
7.1 What is a Sorting Network?
7.2 Figures of Merit for Sorting Networks
7.3 Design of Sorting Networks
7.4 Batcher Sorting Networks
7.5 Other Classes of Sorting Networks
7.6 Selection Networks

7.1 What is a Sorting Network?



The outputs are a permutation of the inputs satisfying $y_0 \le y_1 \le \dots \le y_{n-1}$ (non-descending)

Fig. 7.1 An *n*-input sorting network or an *n*-sorter.

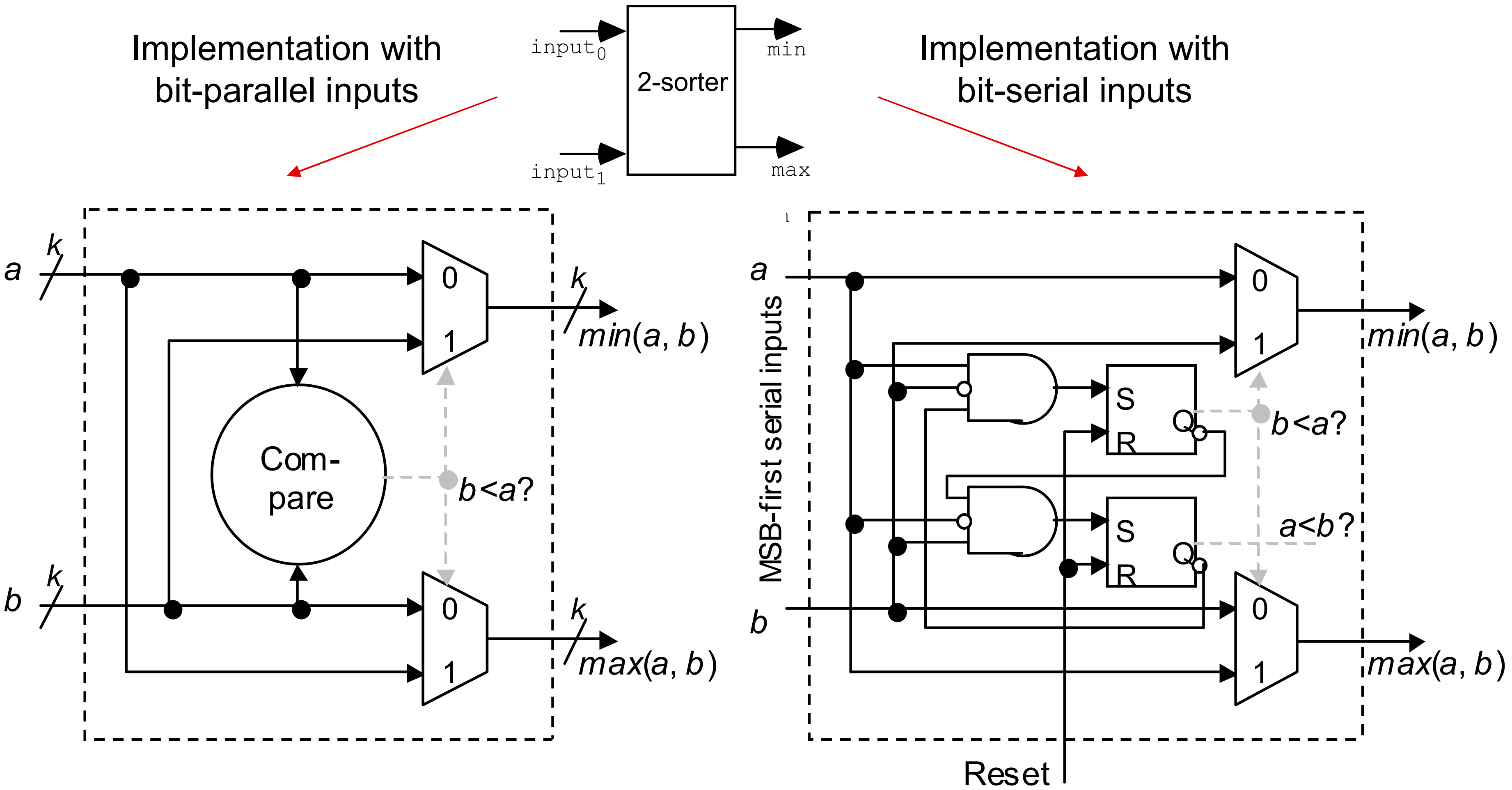


Block Diagram

Alternate Representations

Fig. 7.2 Block diagram and four different schematic representations for a 2-sorter.

Building Blocks for Sorting Networks



Implementation with bit-parallel inputs

Implementation with bit-serial inputs

Fig. 7.3 Parallel and bit-serial hardware realizations of a 2-sorter.

Proving a Sorting Network Correct

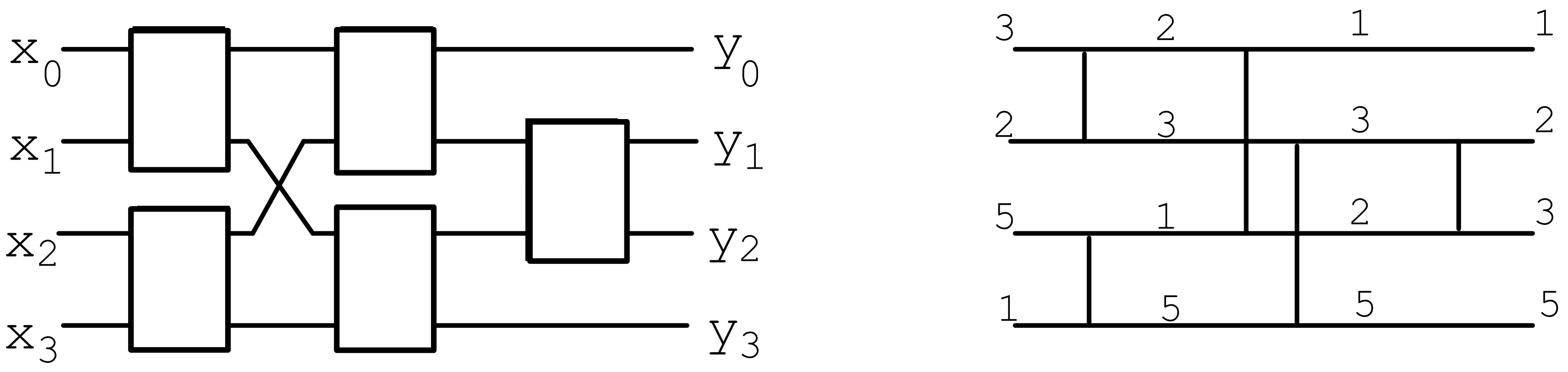


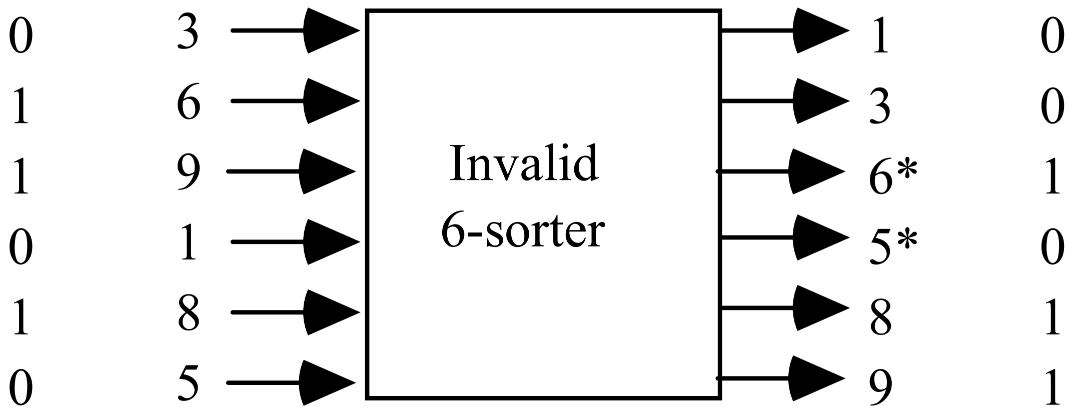
Fig. 7.4 Block diagram and schematic representation of a 4-sorter.

Method 1: Exhaustive test – Try all *n*! possible input orders

Method 2: Ad hoc proof – for the example above, note that y_0 is smallest, y_3 is largest, and the last comparator sorts the other two outputs

Method 3: Use the zero-one principle – A comparison-based sorting algorithm is correct iff it correctly sorts all 0-1 sequences (2^n tests)

Elaboration on the Zero-One Principle



Deriving a 0-1 sequence that is not correctly sorted, given an arbitrary sequence that is not correctly sorted.

Let outputs y_i and y_{i+1} be out of order, that is $y_i > y_{i+1}$

Replace inputs that are strictly less than y_i with 0s and all others with 1s

The resulting 0-1 sequence will not be correctly sorted either

7.2 Figures of Merit for Sorting Networks

Cost: Number of comparators

In the following example, we have 5 comparators

Delay: Number of levels

The following 4-sorter has 3 comparator levels on its critical path

Cost × Delay

The cost-delay product for this example is 15

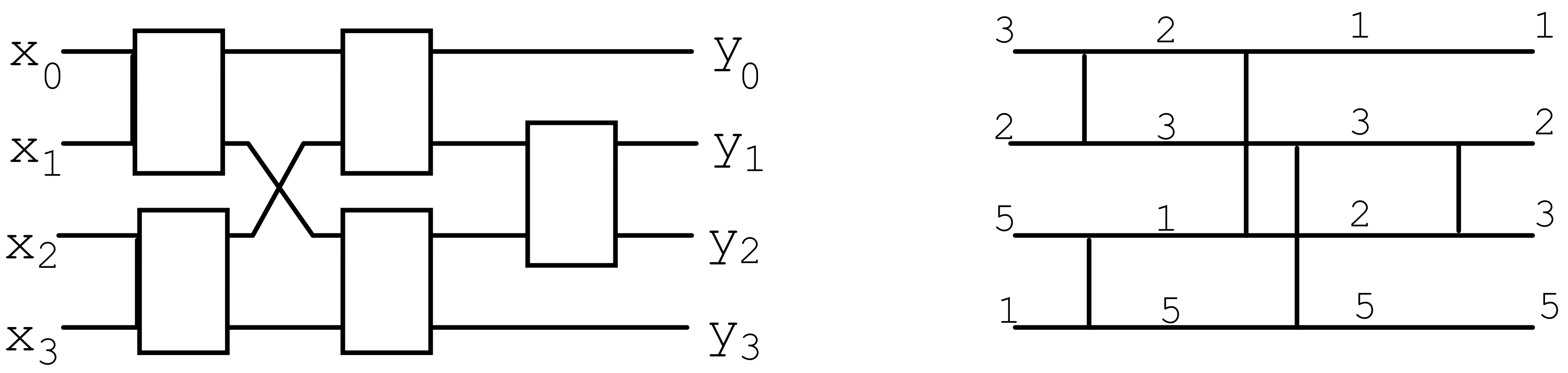
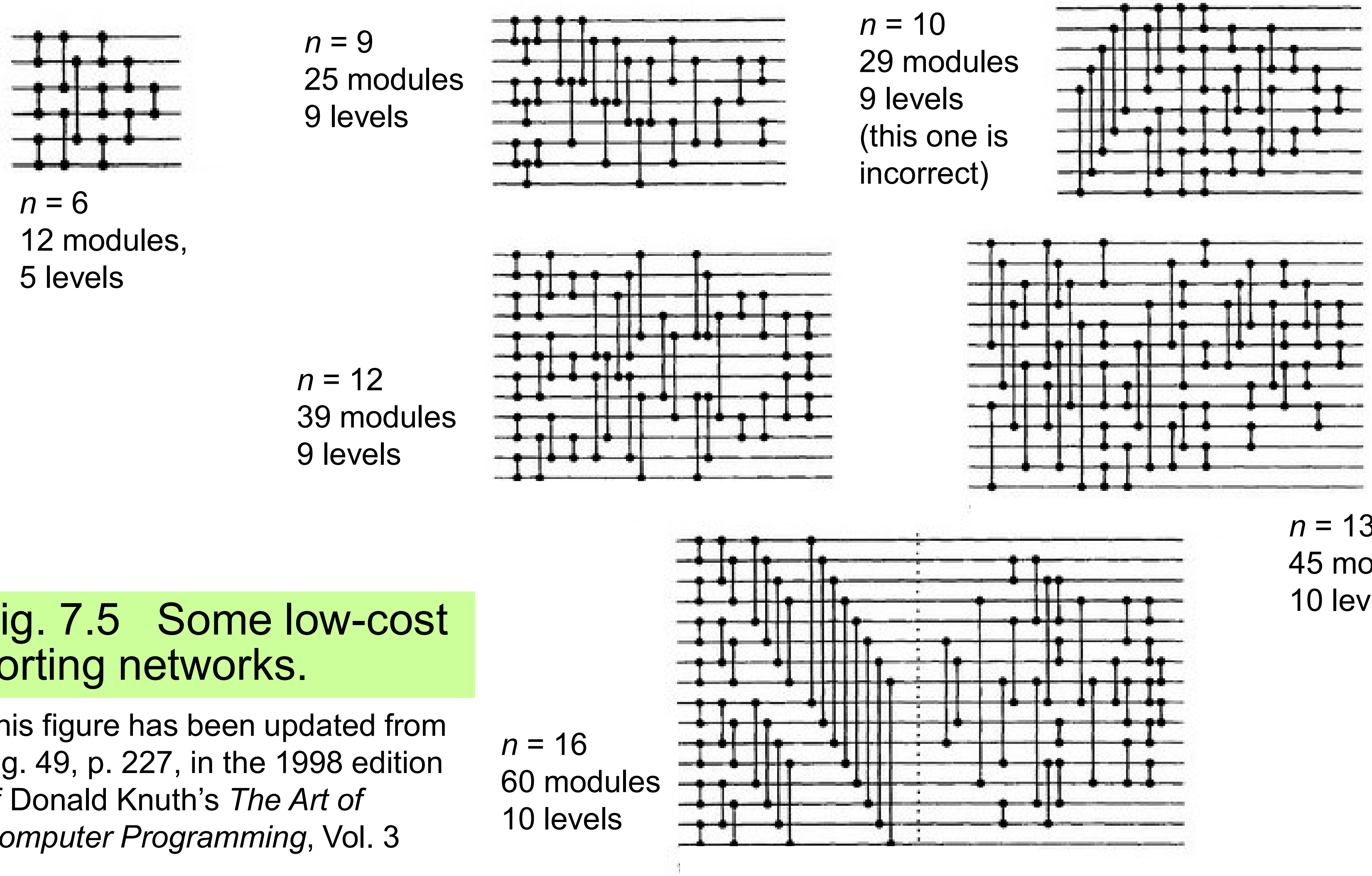


Fig. 7.4 Block diagram and schematic representation of a 4-sorter.

Cost as a Figure of Merit

Optimal size is known for *n* = 1 to 8: 0, 1, 3, 5, 9, 12, 16, 19



n = 6
12 modules,
5 levels

n = 9
25 modules
9 levels

n = 10
29 modules
9 levels
(this one is incorrect)

n = 12
39 modules
9 levels

n = 13
45 modules
10 levels

n = 16
60 modules
10 levels

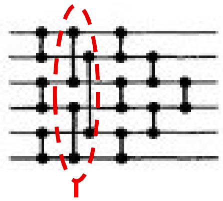
Fig. 7.5 Some low-cost sorting networks.

This figure has been updated from Fig. 49, p. 227, in the 1998 edition of Donald Knuth's *The Art of Computer Programming*, Vol. 3

Delay as a Figure of Merit

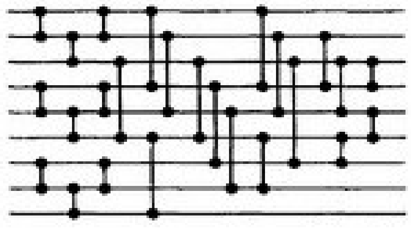
Optimal delay is known for *n* = 1 to 10: 0, 1, 3, 3, 5, 5, 6, 6, 7, 7

n = 6
12 modules,
5 levels

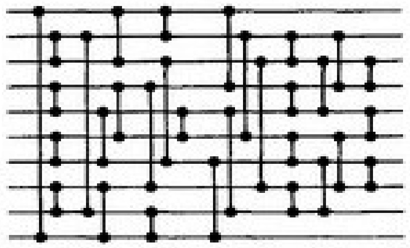


These 3 comparators constitute one level

n = 9
25 modules
7 ~~8~~ levels
(this one is incorrect)



n = 10
31 modules
7 levels



n = 12
40 modules
8 levels

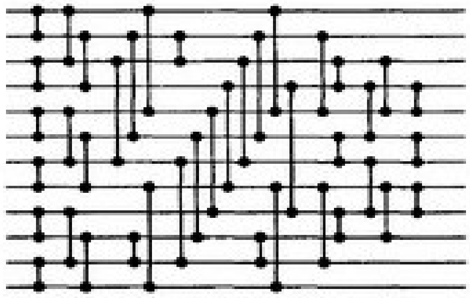
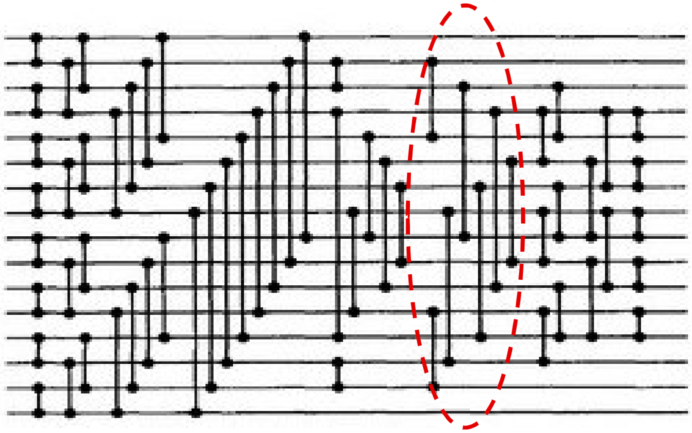


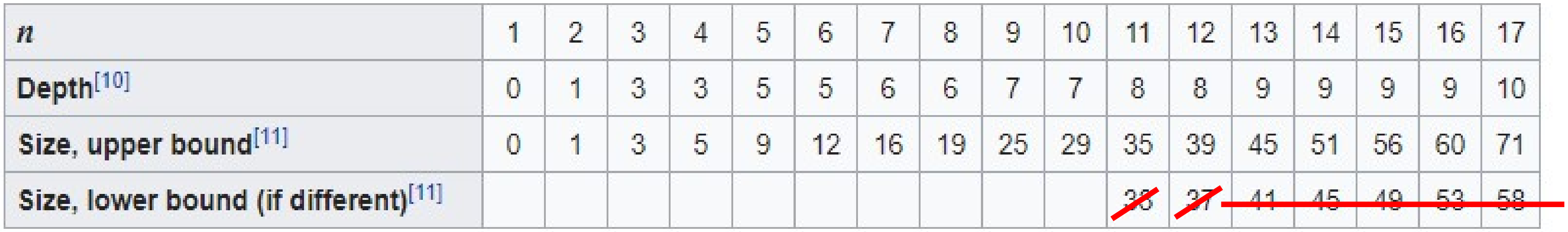
Fig. 7.6 Some fast sorting networks.

This figure has been updated from Fig. 51, p. 229, in the 1998 edition of Donald Knuth's *The Art of Computer Programming*, Vol. 3



n = 16
61 modules
9 levels

Best Sorting Networks Known



n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth[10]	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9	10
Size, upper bound[11]	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60	71
Size, lower bound (if different)[11]											~~33~~	~~37~~	~~41~~	~~45~~	~~49~~	~~53~~	~~58~~

Source: Wikipedia

43 47 51 55 60

[2021 updates]

References:

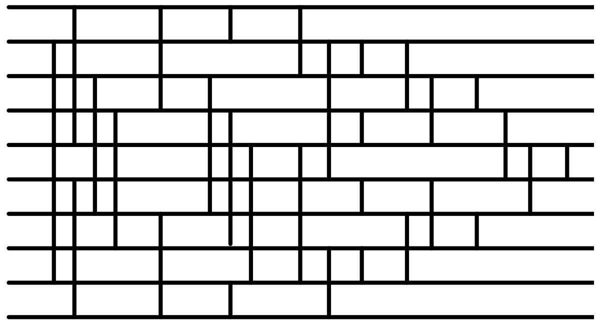
[10] Codish, Michael, Luis Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp, “Sorting Networks: To the End and Back Again,” 2015, arXiv:1507.01428

[11] Codish, Michael, Luis Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp, “Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten),” *Proc. Int'l Conf. Tools with AI*, pp. 186-193, 2014. arXiv:1405.5754

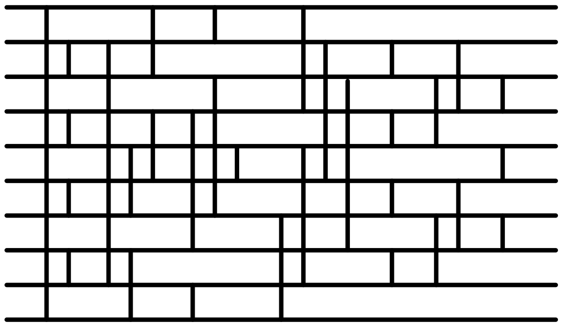
The problem of determining whether a given candidate network is a sorting network is co-NP-complete

[13] Parberry, Ian, On the Computational Complexity of Optimal Sorting Network Verification, *Proc. PARLE '91: Parallel Architectures and Languages Europe*, Volume I: Parallel Architectures and Algorithms, 1991, pp. 252-269.

Cost-Delay Product as a Figure of Merit



n = 10, 29 modules, 9 levels



n = 10, 31 modules, 7 levels

Low-cost 10-sorter from Fig. 7.5

Fast 10-sorter from Fig. 7.6

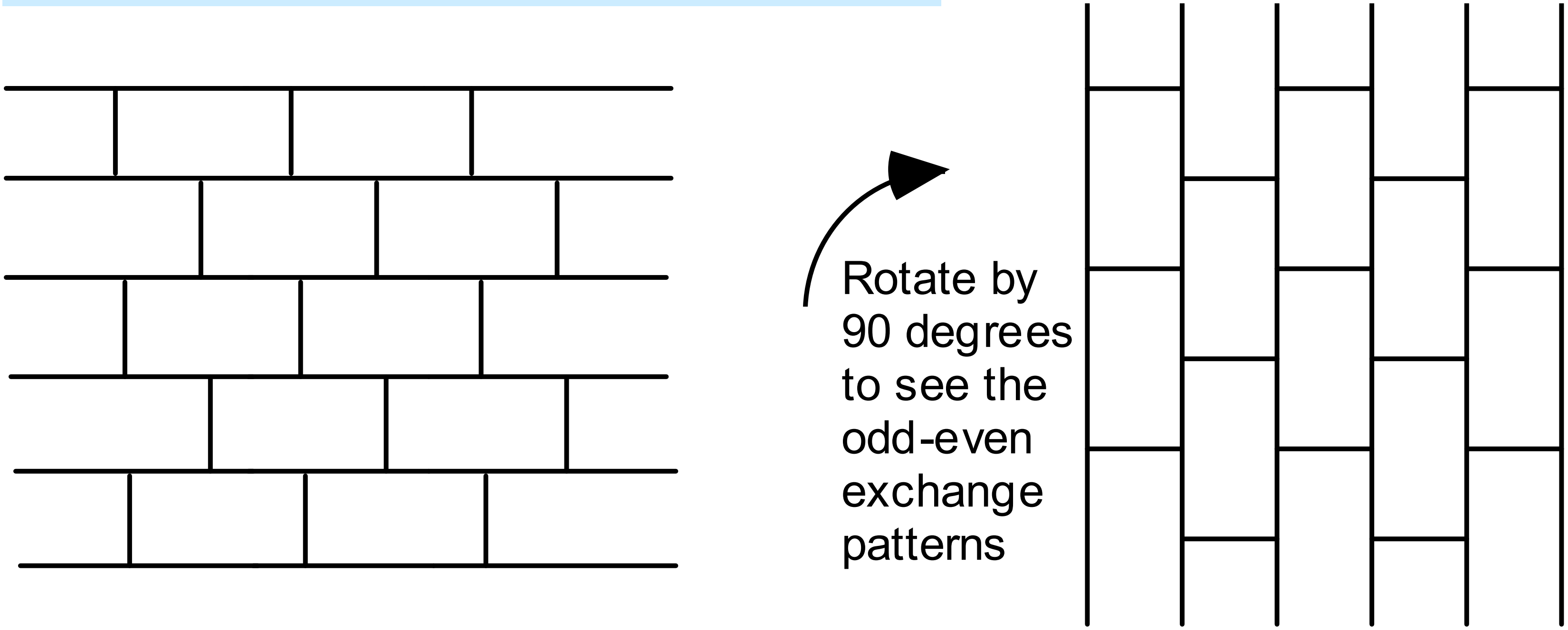
Cost × Delay = 29 × 9 = 261

Cost × Delay = 31 × 7 = 217

The most cost-effective *n*-sorter may be neither the fastest design, nor the lowest-cost design

7.3 Design of Sorting Networks

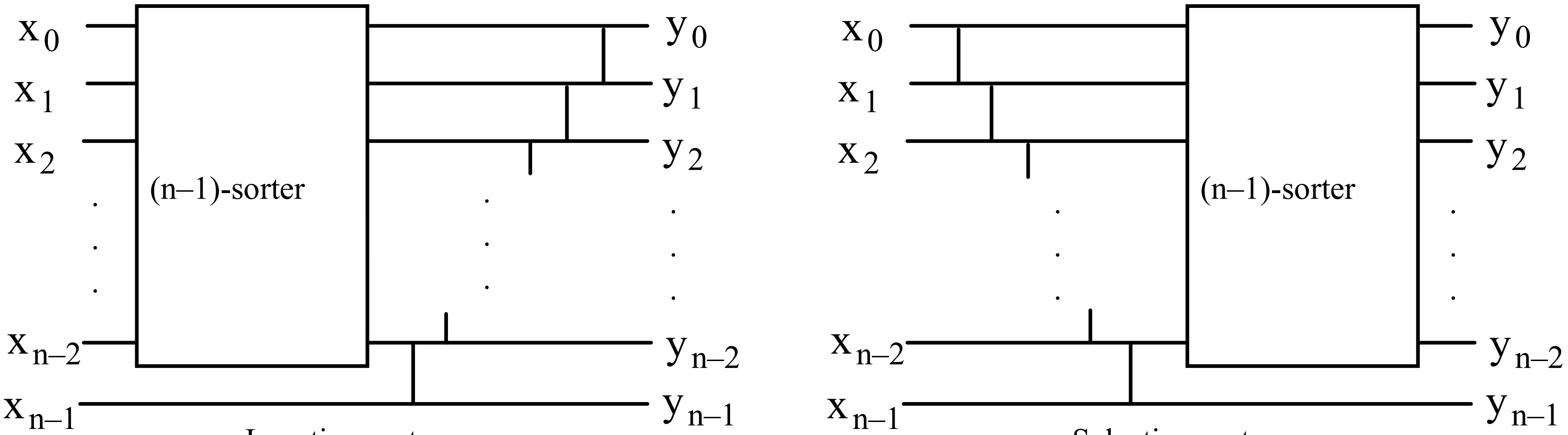
$$\begin{aligned} C(n) &= n(n-1)/2 \\ D(n\,) &= n \\ \text{Cost} \times \text{Delay} &= n^2(n-1)/2 \;=\; \Theta(n^3) \end{aligned}$$



Rotate by 90 degrees to see the odd-even exchange patterns

Fig. 7.7 Brick-wall 6-sorter based on odd–even transposition.

Insertion Sort and Selection Sort



Insertion sort

Selection sort

Parallel insertion sort = Parallel selection sort = Parallel bubble sort!

$C(n) = n(n-1)/2$

$D(n) = 2n-3$

Cost × Delay $= \Theta(n^3)$

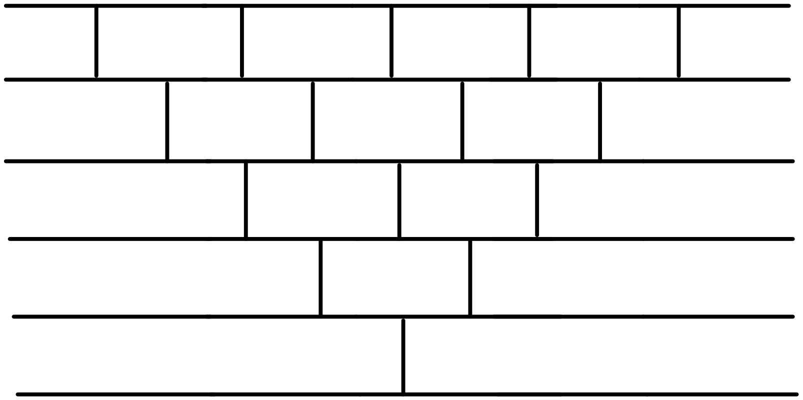
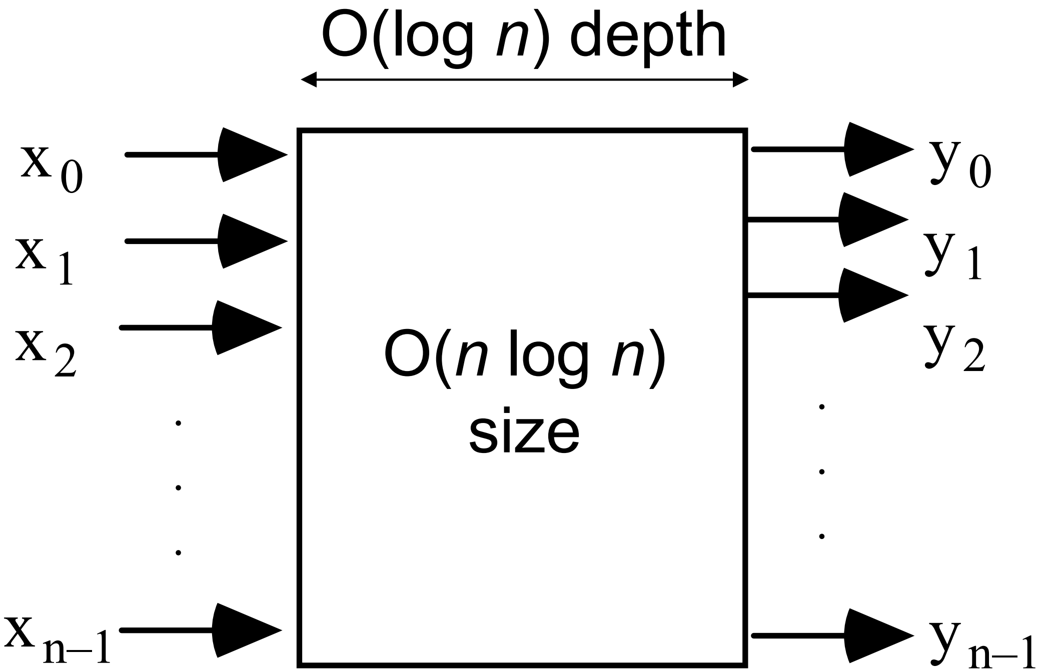


Fig. 7.8 Sorting network based on insertion sort or selection sort.

Theoretically Optimal Sorting Networks



AKS sorting network
(Ajtai, Komlos, Szemeredi: 1983)

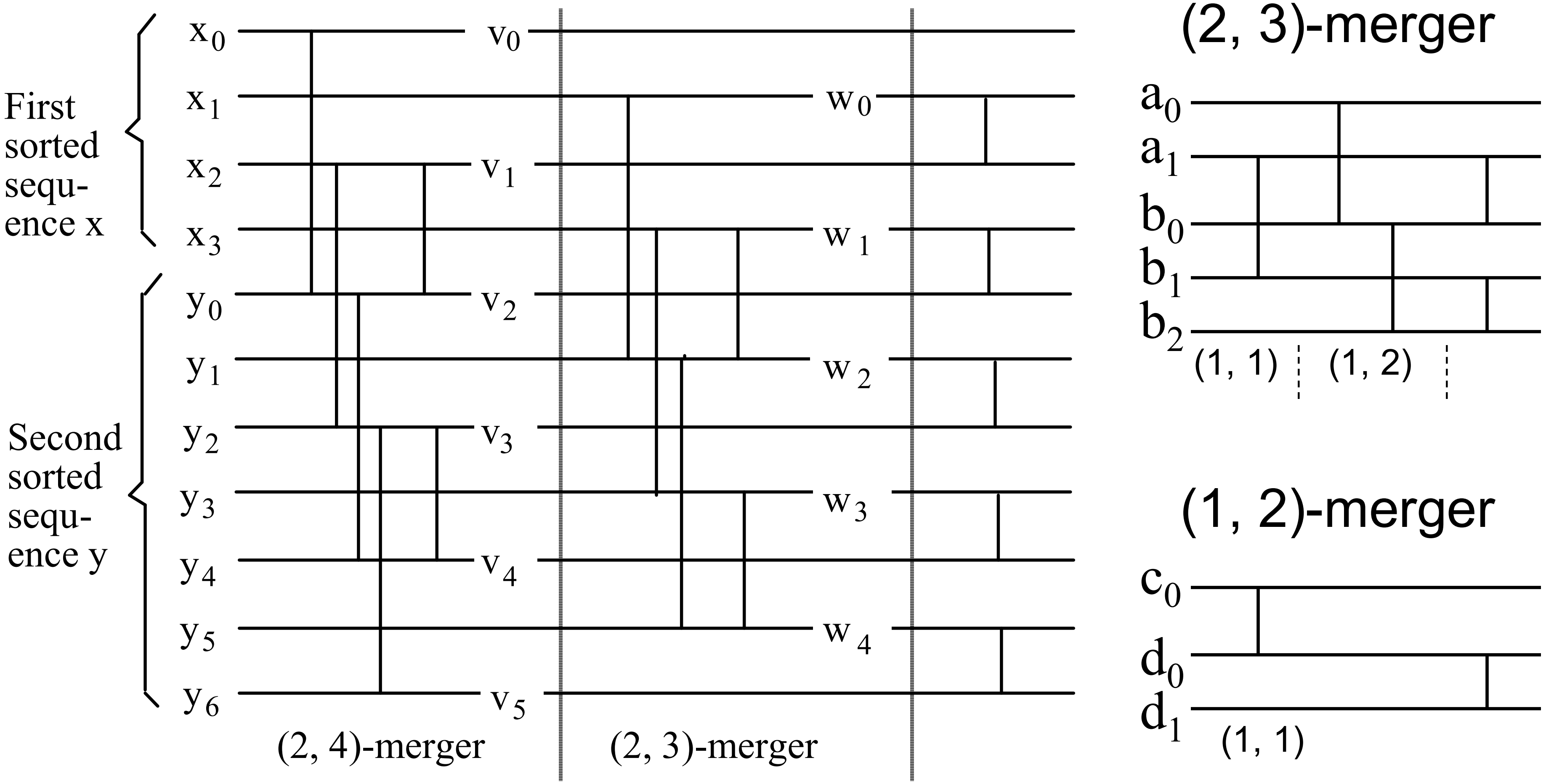
Note that even for these optimal networks, delay-cost product is suboptimal; but this is the best we can do

Existing sorting networks have O($\log^2 n$) latency and O($n \log^2 n$) cost

Given that $\log_2 n$ is only 20 for n = 1 000 000, the latter are more practical

Unfortunately, AKS networks are not practical owing to large (4-digit) constant factors involved; improvements since 1983 not enough

7.4 Batcher Sorting Networks

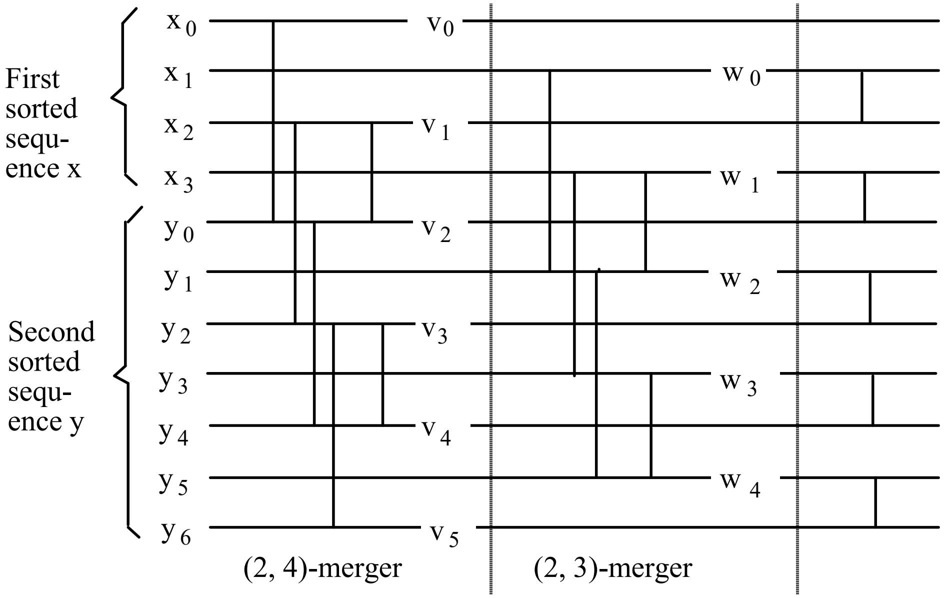


(2, 3)-merger

(1, 2)-merger

Fig. 7.9   Batcher’s even–odd merging network for 4 + 7 inputs.

Proof of Batcher's Even-Odd Merge



Use the zero-one principle

Assume: x has k 0s
y has k' 0s

v has $k_{\text{even}} = \lceil k/2 \rceil + \lceil k'/2 \rceil$ 0s

w has $k_{\text{odd}} = \lfloor k/2 \rfloor + \lfloor k'/2 \rfloor$ 0s

Case a: $k_{\text{even}} = k_{\text{odd}}$

v 0 0 0 0 0 0 1 1 1 1 1 1
w 0 0 0 0 0 0 1 1 1 1 1

Case b: $k_{\text{even}} = k_{\text{odd}}+1$

v 0 0 0 0 0 0 0 1 1 1 1 1
w 0 0 0 0 0 0 1 1 1 1 1

Case c: $k_{\text{even}} = k_{\text{odd}}+2$

v 0 0 0 0 0 0 0 0 1 1 1 1
w 0 0 0 0 0 0 1 1 1 1 1

Out of order

Batcher's Even-Odd Merge Sorting

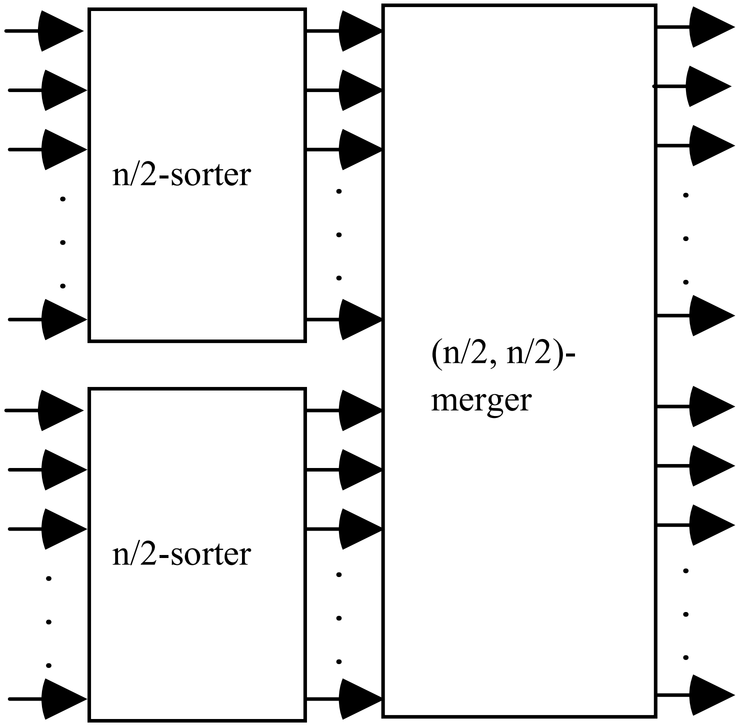


Fig. 7.10 The recursive structure of Batcher's even–odd merge sorting network.

Batcher's (m, m) even-odd merger, for m a power of 2:

$$\begin{aligned} C(m) &= 2C(m/2) + m - 1 \\ &= (m-1) + 2(m/2-1) + 4(m/4-1) + \ldots \\ &= m \log_2 m + 1 \end{aligned}$$

$$D(m) = D(m/2) + 1 = \log_2 m + 1$$

$$\text{Cost} \times \text{Delay} = \Theta(m \log^2 m)$$

Batcher sorting networks based on the even-odd merge technique:

$$\begin{aligned} C(n) &= 2C(n/2) + (n/2)(\log_2(n/2)) + 1 \\ &\cong n(\log_2 n)^2/\,2 \end{aligned}$$

$$\begin{aligned} D(n) &= D(n/2) + \log_2(n/2) + 1 \\ &= D(n/2) + \log_2 n \\ &= \log_2 n\,(\log_2 n + 1)/2 \end{aligned}$$

$$\text{Cost} \times \text{Delay} = \Theta(n \log^4 n)$$

Example Batcher's Even-Odd 8-Sorter

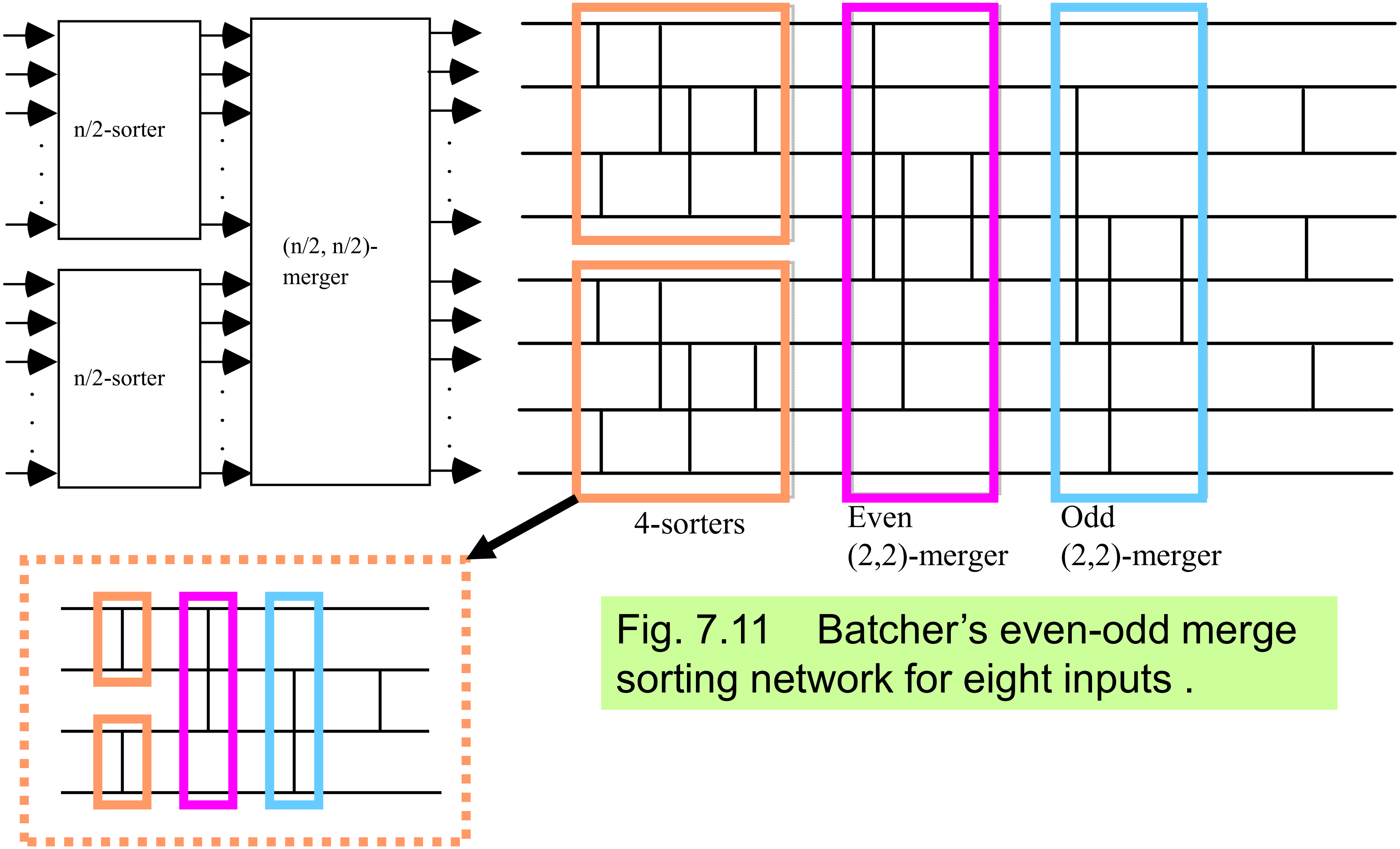
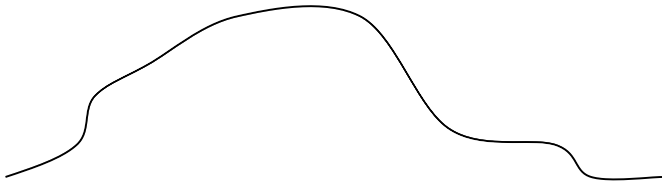


Fig. 7.11 Batcher's even-odd merge sorting network for eight inputs .

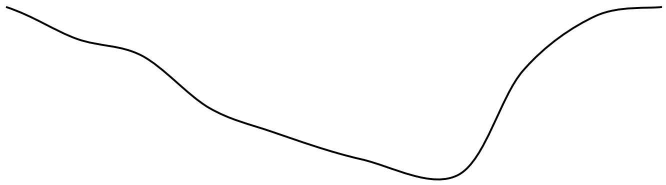
Bitonic-Sequence Sorter

Bitonic sequence:

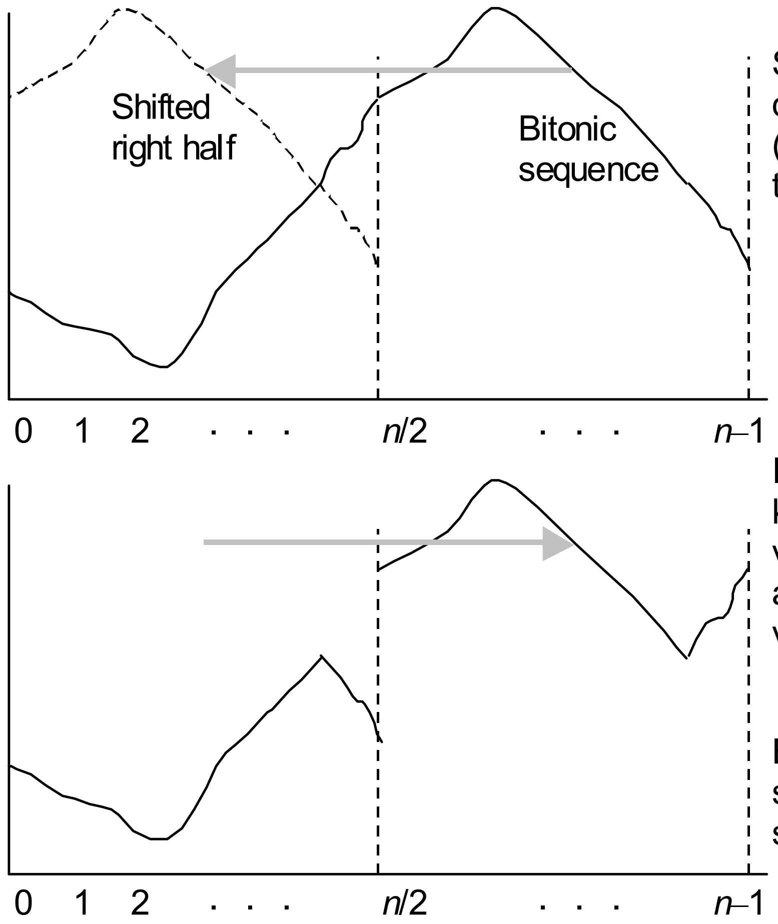
1 3 3 4 6 6 6 2 2 1 0 0
Rises, then falls



8 7 7 6 6 6 5 4 6 8 8 9
Falls, then rises



8 9 8 7 7 6 6 6 5 4 6 8
The previous sequence, right-rotated by 2



Shift right half of data to left half (superimpose the two halves)

In each position, keep the smaller value of each pair and ship the larger value to the right

Each half is a bitonic sequence that can be sorted independently

Fig. 14.2 Sorting a bitonic sequence on a linear array.

Batcher's Bitonic Sorting Networks

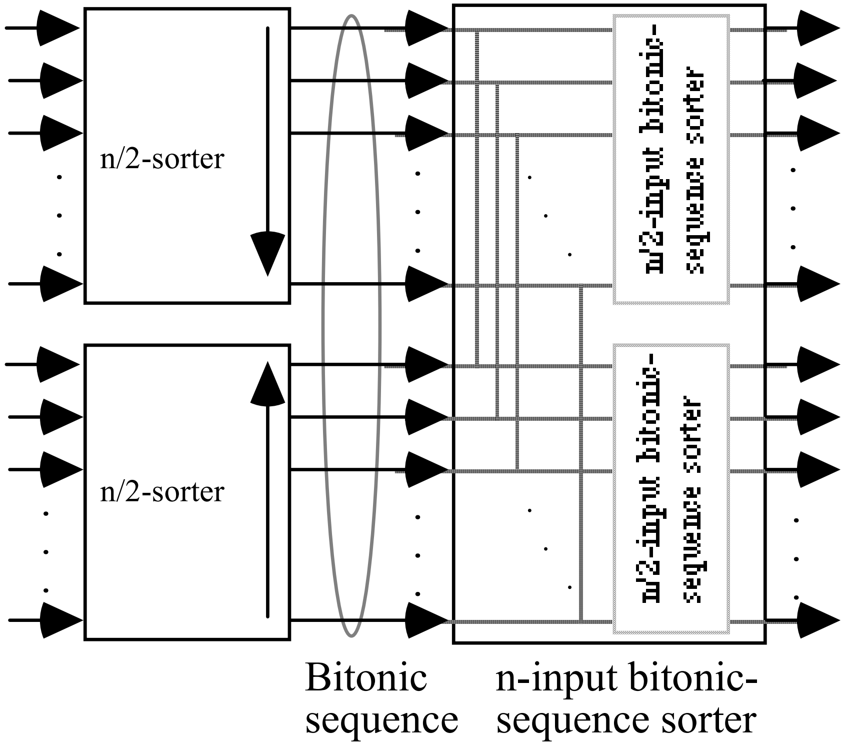

n/2-sorter
n/2-sorter
n/2-input bitonic-sequence sorter
n/2-input bitonic-sequence sorter
Bitonic sequence
n-input bitonic-sequence sorter

Fig. 7.12 The recursive structure of Batcher's bitonic sorting network.

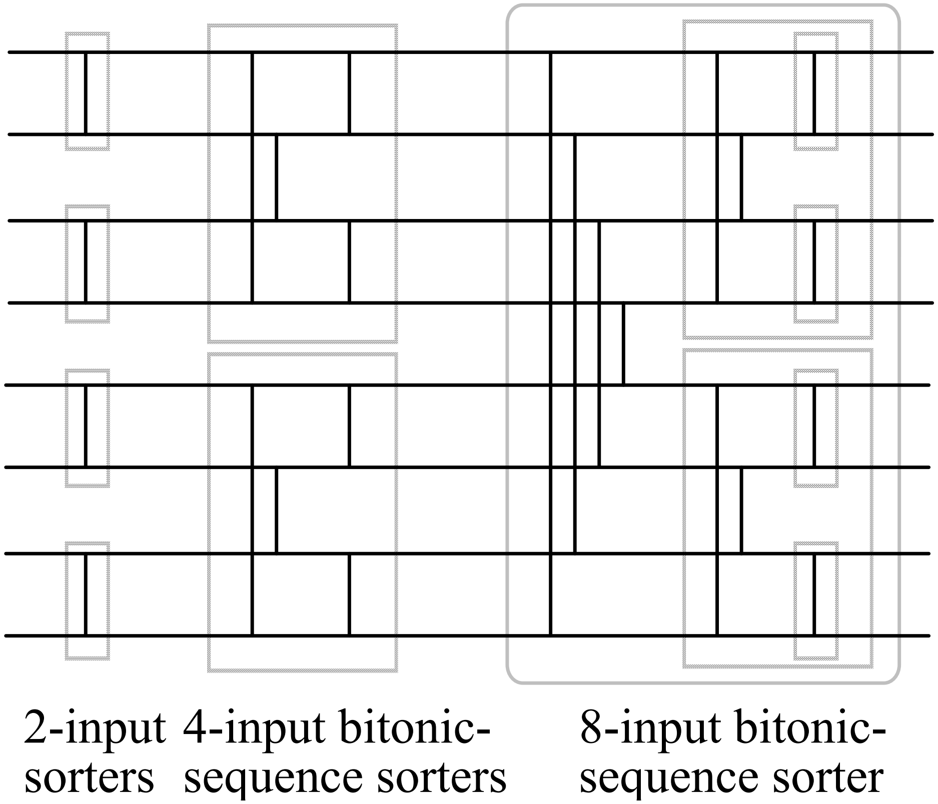

2-input sorters
4-input bitonic-sequence sorters
8-input bitonic-sequence sorter

Fig. 7.13 Batcher's bitonic sorting network for eight inputs.

7.5 Other Classes of Sorting Networks

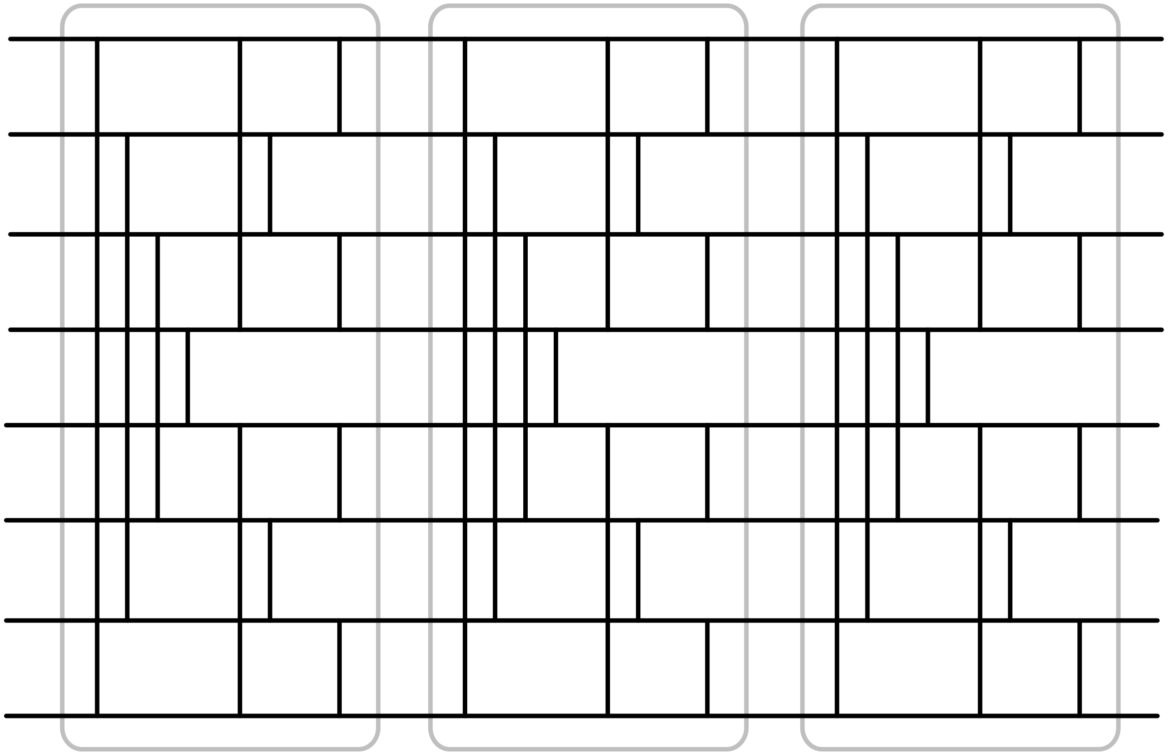


Fig. 7.14 Periodic balanced sorting network for eight inputs.

Desirable properties:

a. Regular / modular (easier VLSI layout).

b. Simpler circuits via reusing the blocks

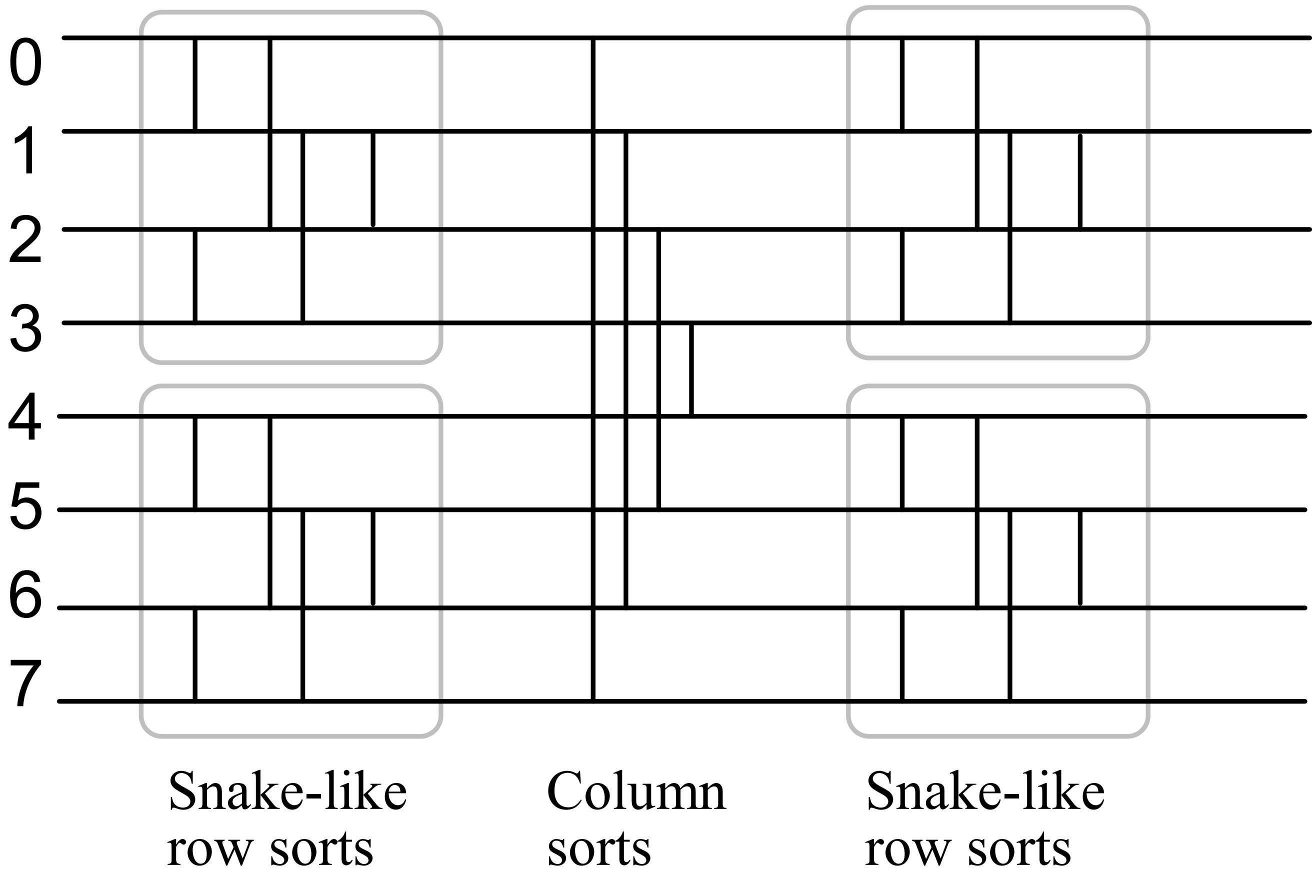
c. With an extra block tolerates some faults (missed exchanges)

d. With 2 extra blocks provides tolerance to single faults (a missed or incorrect exchange)

e. Multiple passes through faulty network (graceful degradation)

f. Single-block design becomes fault-tolerant by using an extra stage

Shearsort-Based Sorting Networks (1)

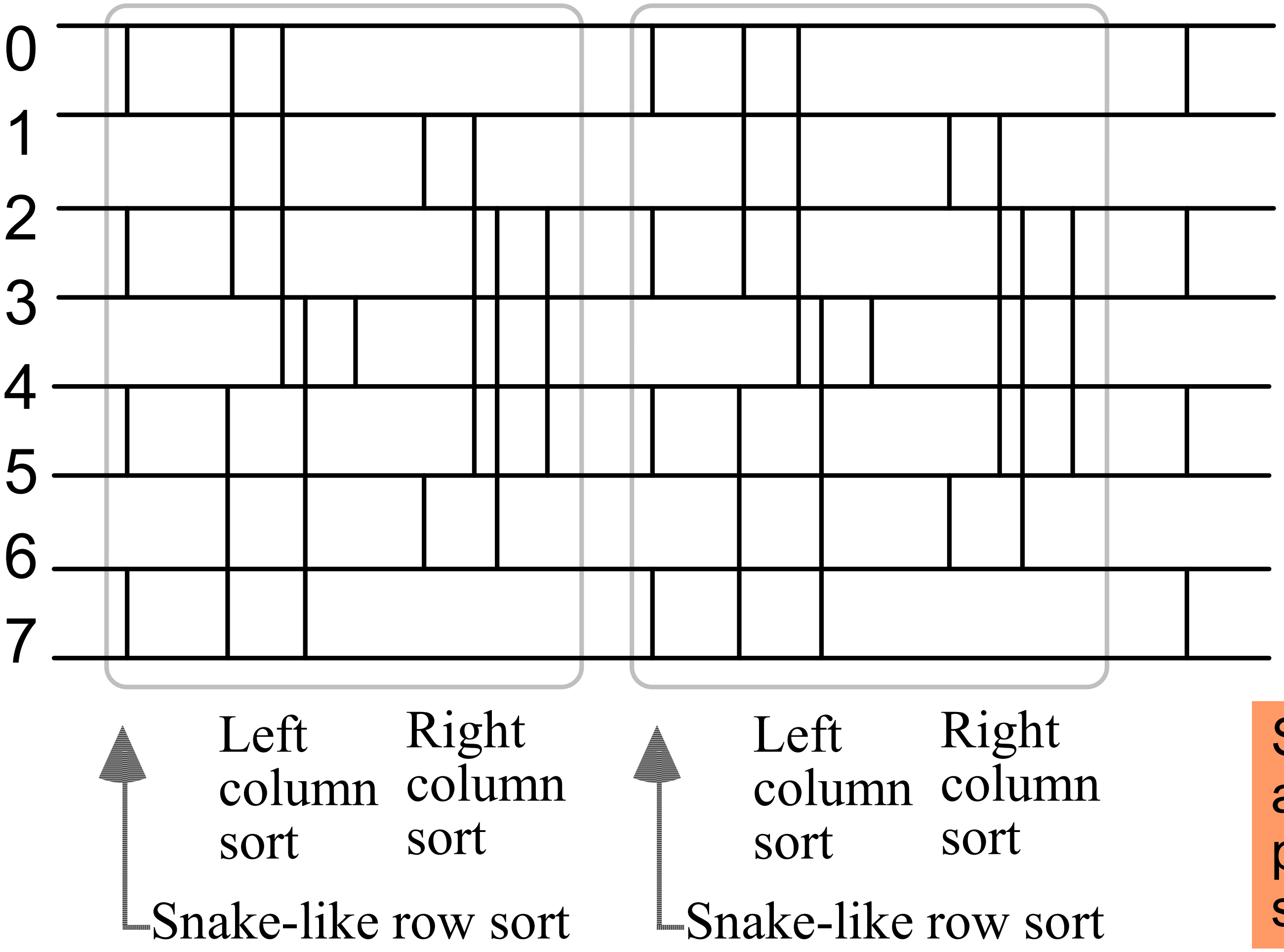


0	1	2	3
7	6	5	4

Corresponding 2-D mesh

Fig. 7.15 Design of an 8-sorter based on shearsort on 2×4 mesh.

Shearsort-Based Sorting Networks (2)



0	1
3	2
4	5
7	6

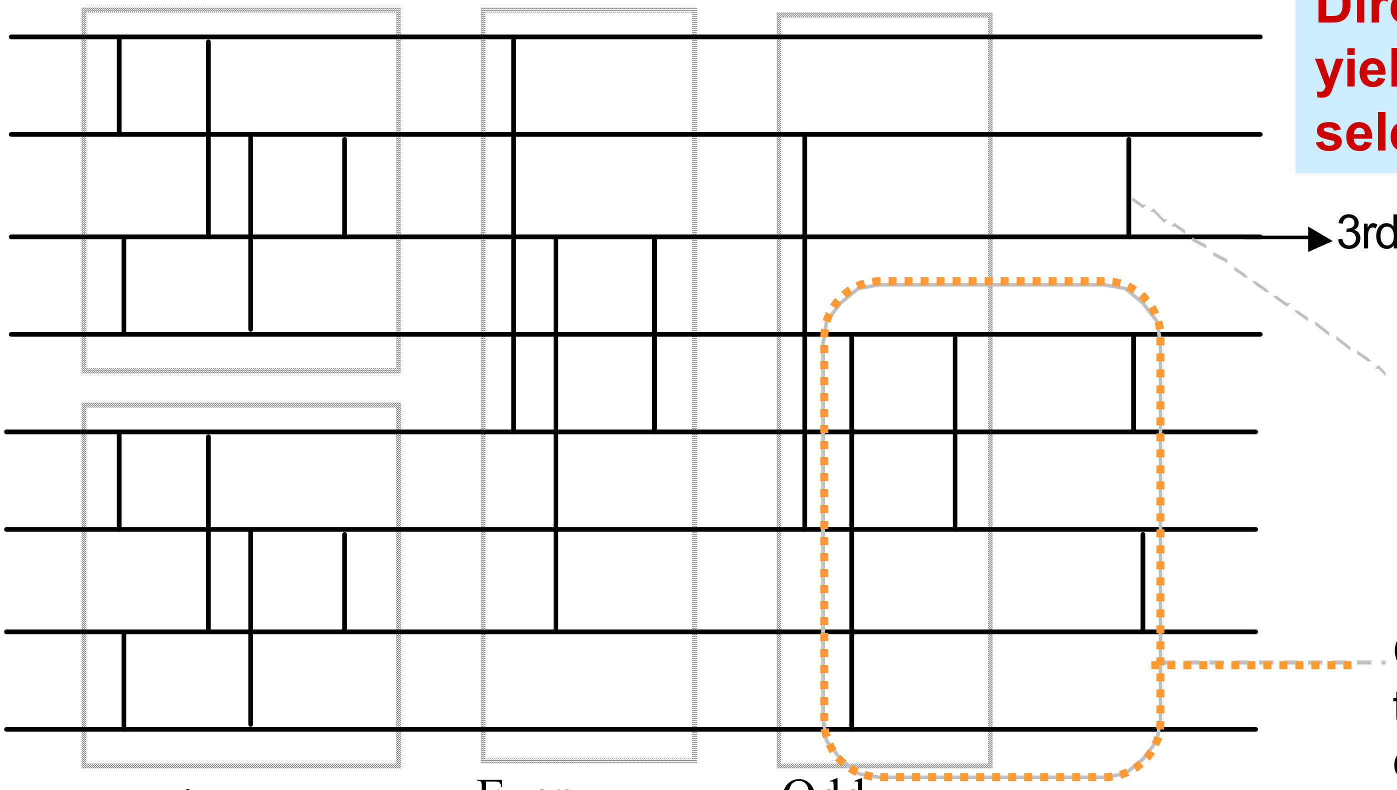
Corresponding 2-D mesh

Some of the same advantages as periodic balanced sorting networks

Fig. 7.16 Design of an 8-sorter based on shearsort on 2×4 mesh.

7.6 Selection Networks

Direct design may yield simpler/faster selection networks



3rd smallest element

Can remove this block if smallest three inputs needed

Can remove these four comparators

4-sorters

Even (2,2)-merger

Odd (2,2)-merger

Deriving an (8, 3)-selector from Batcher’s even-odd merge 8-sorter.

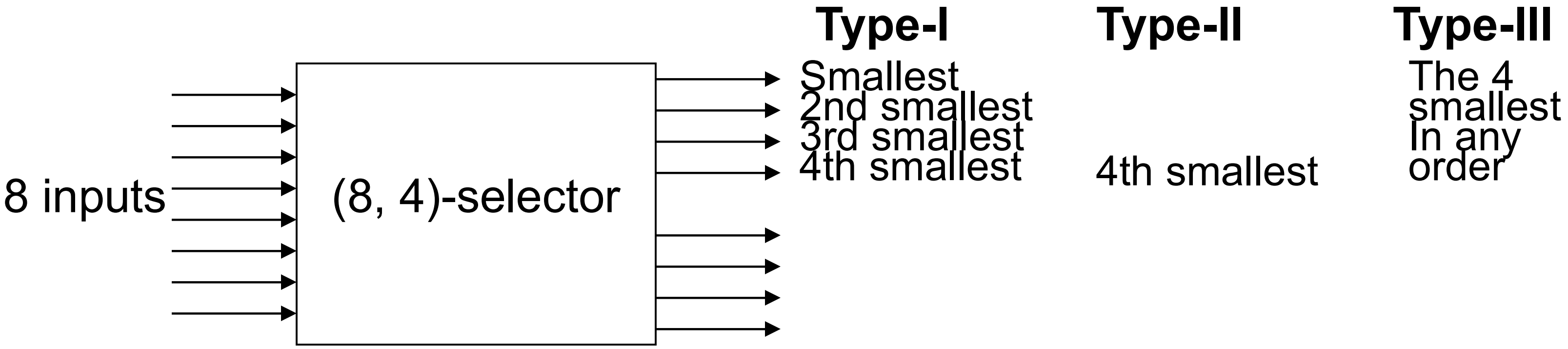
Categories of Selection Networks

Unfortunately we know even less about selection networks than we do about sorting networks.

One can define three selection problems [Knut81]:

I. Select the *k* smallest values; present in sorted order
II. Select *k*th smallest value
III. Select the *k* smallest values; present in any order

Circuit and time complexity: (I) hardest, (III) easiest



8 inputs

(8, 4)-selector

Type-I

Smallest
2nd smallest
3rd smallest
4th smallest

Type-II

4th smallest

Type-III

The 4 smallest In any order

Type-III Selection Networks

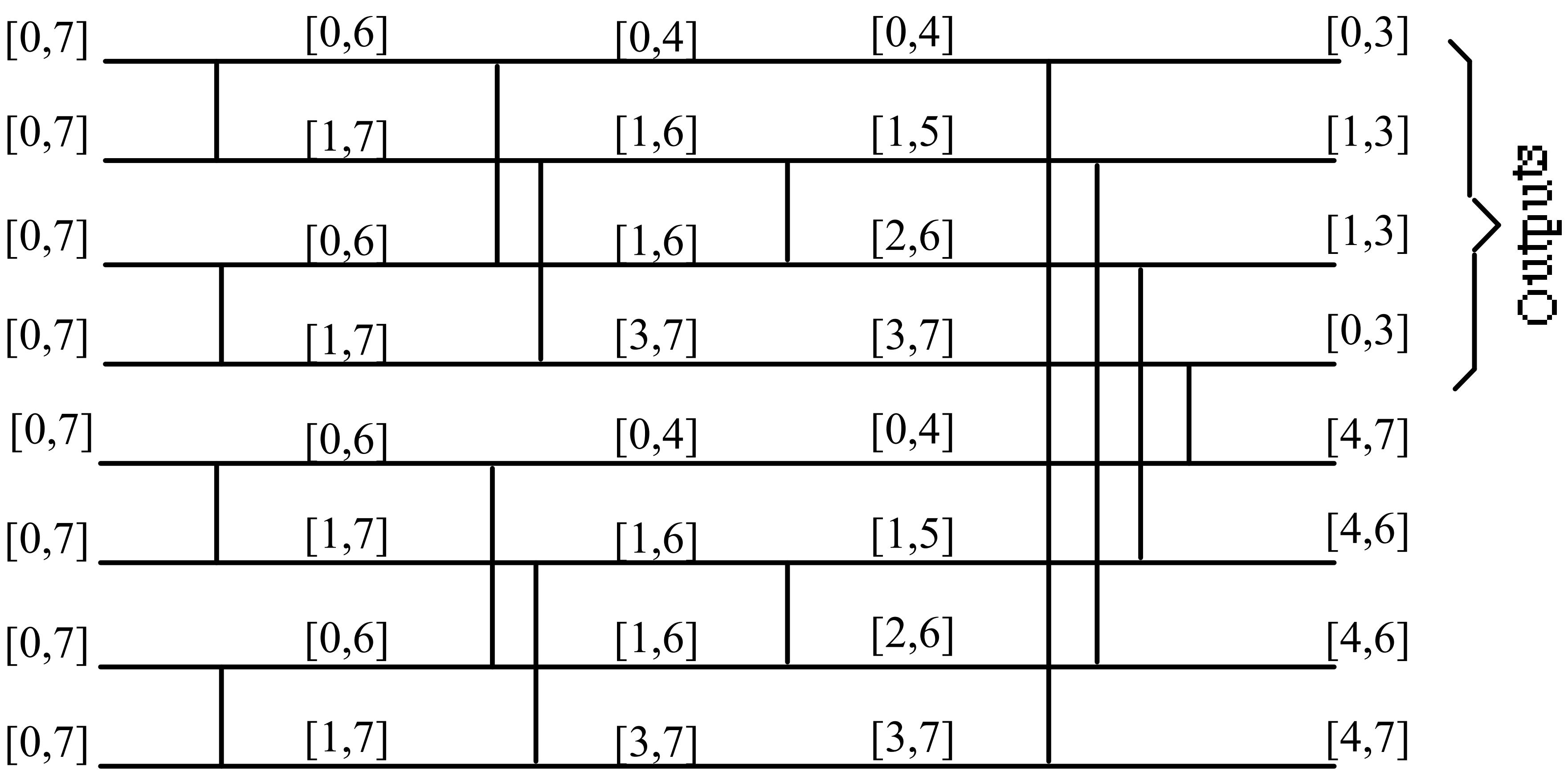
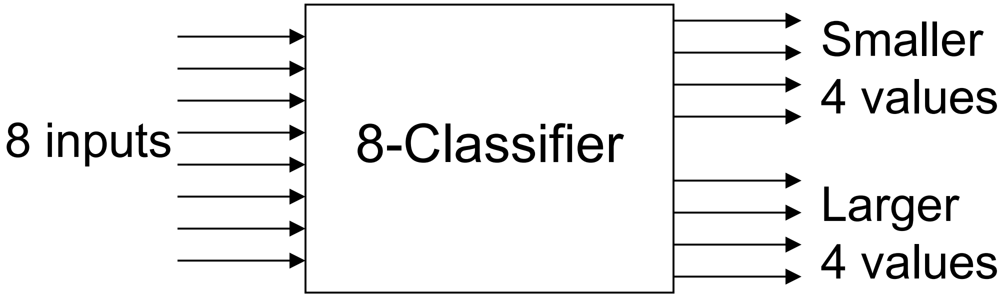


Figure 7.17 A type III (8, 4)-selector.

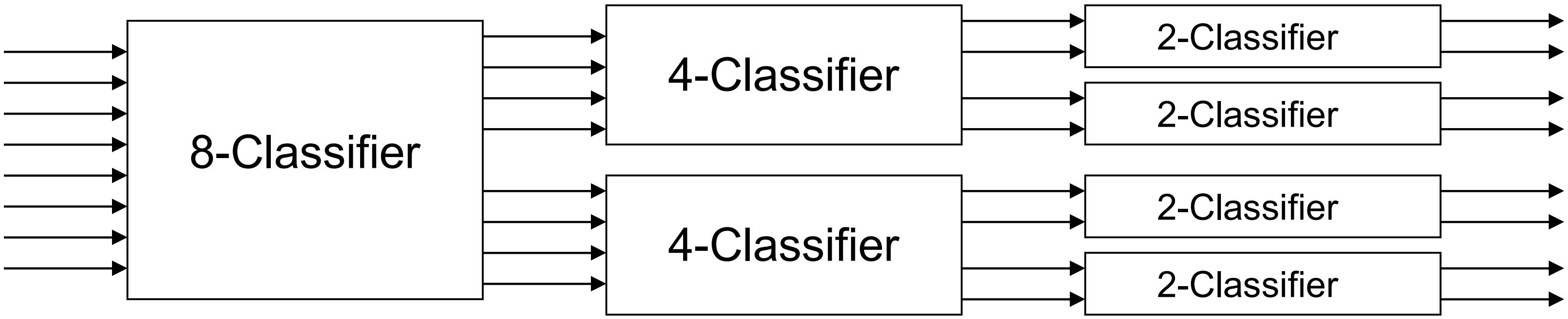
8-Classifier

Classifier Networks

Classifiers:
Selectors that separate the smaller half of values from the larger half



Use of classifiers for building sorting networks



Problem: Given O(log *n*)-time and O(*n* log *n*)-cost *n*-classifier designs, what are the delay and cost of the resulting sorting network?

8A Search Acceleration Circuits

Much of sorting is done to facilitate/accelerate searching

- Simple search can be speeded up via special circuits
- More complicated searches: range, approximate-match

Topics in This Chapter	
8A.1	Systolic Priority Queues
8A.2	Searching and Dictionary Operations
8A.3	Tree-Structured Dictionary Machines
8A.4	Associative Memories
8A.5	Associative Processors
8A.6	VLSI Trade-offs in Search Processors

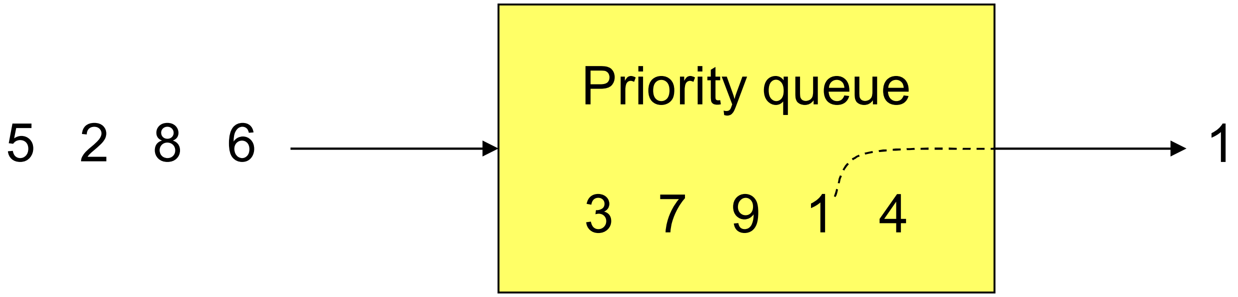
8A.1 Systolic Priority Queues

Problem: We want to maintain a large list of keys, so that we can add new keys into it (insert operation) and obtain the smallest key (extract operation) whenever desired.

Unsorted list: Constant-time insertion / Linear-time extraction

Sorted list: Linear-time insertion / Constant-time extraction

Can both insert and extract operations (priority-queue operations) be performed in constant time, independent of the size of the list?



First Attempt: Via a Linear-Array Sorter

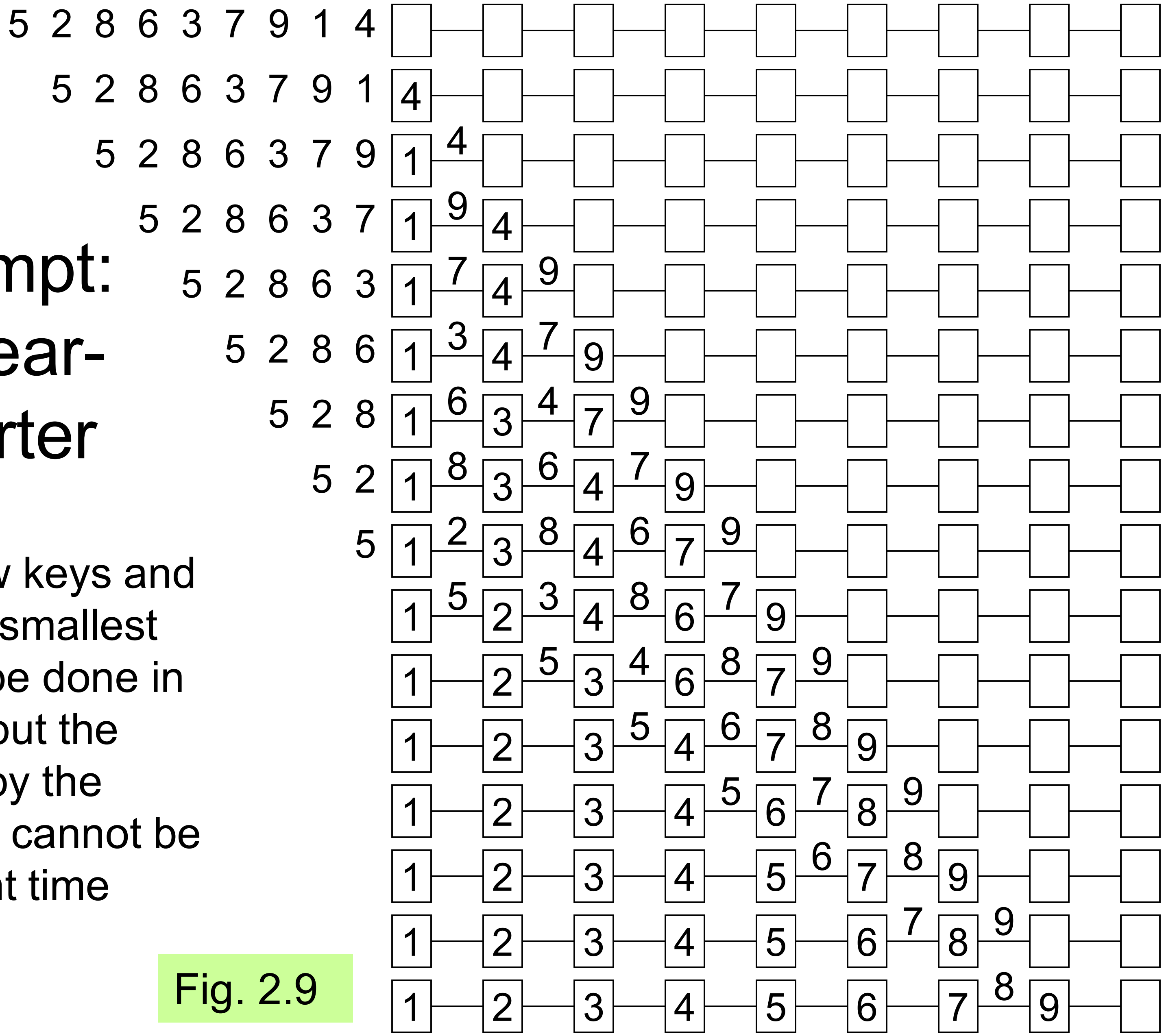
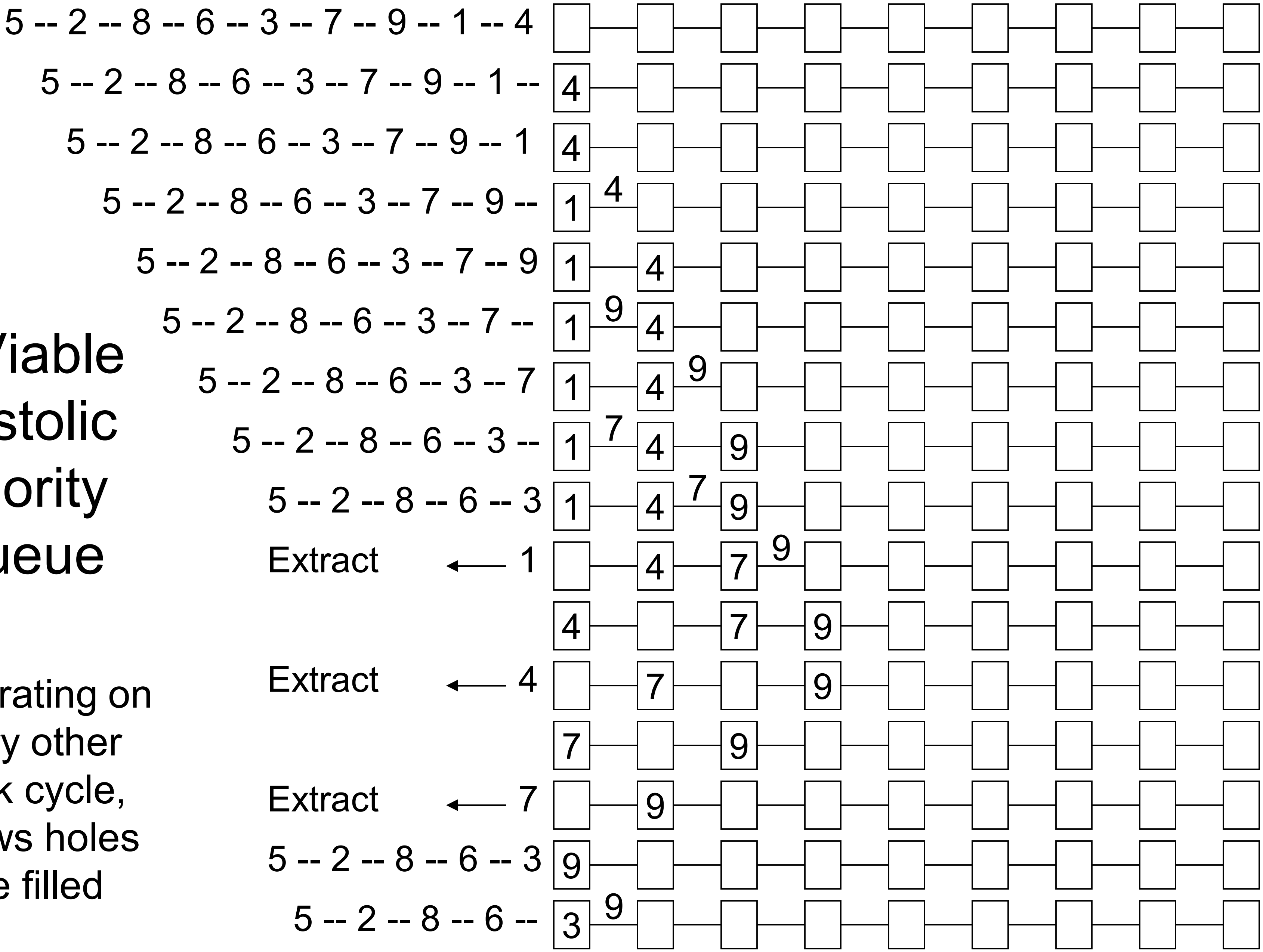


Fig. 2.9

Insertion of new keys and read-out of the smallest key value can be done in constant time, but the “hole” created by the extracted value cannot be filled in constant time

A Viable Systolic Priority Queue

Operating on every other clock cycle, allows holes to be filled



5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 9 -- 1 -- 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 9 -- 1 -- 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 9 -- 1 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 9 -- 1 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 9 1 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 -- 1 9 4

5 -- 2 -- 8 -- 6 -- 3 -- 7 1 4 9

5 -- 2 -- 8 -- 6 -- 3 -- 1 7 4 9

5 -- 2 -- 8 -- 6 -- 3 1 4 7 9

Extract ← 1 4 7 9

4 7 9

Extract ← 4 7 9

7 9

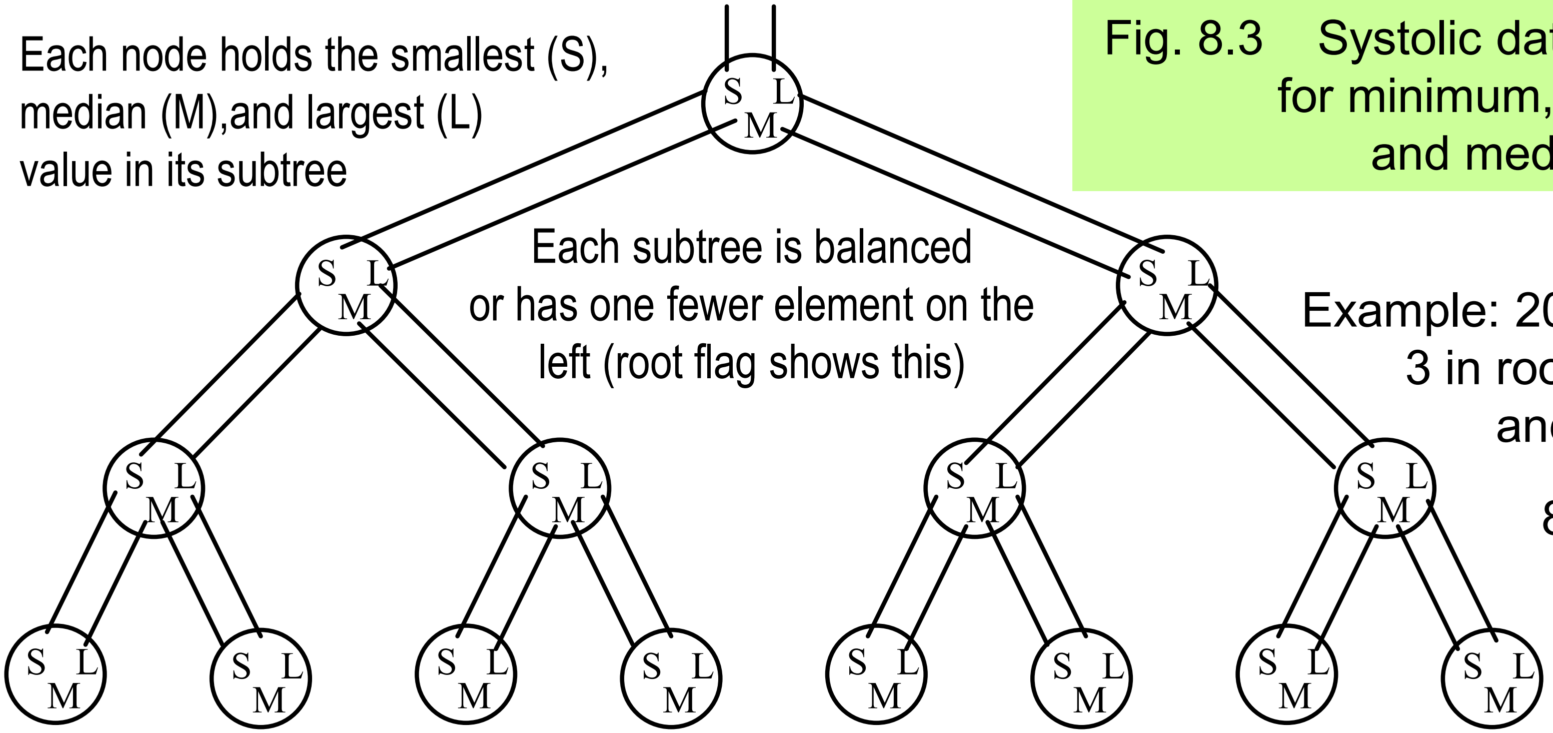
Extract ← 7 9

5 -- 2 -- 8 -- 6 -- 3 9

5 -- 2 -- 8 -- 6 -- 3 9

Systolic Data Structures

Each node holds the smallest (S), median (M),and largest (L) value in its subtree

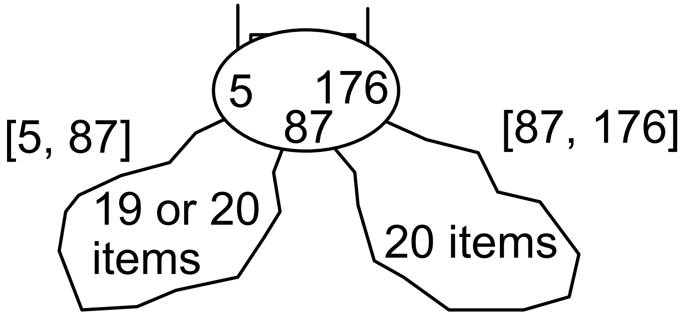


Each subtree is balanced or has one fewer element on the left (root flag shows this)

Fig. 8.3 Systolic data structure for minimum, maximum, and median finding.

Example: 20 elements, 3 in root, 8 on left, and 9 on right

8 elements: 3 + 2 + 3



Insert 2
Insert 20
Insert 127
Insert 195

Extractmin
Extractmed
Extractmax

Update/access examples for the systolic data structure of Fig. 8.3

8A.2 Searching and Dictionary Operations

Parallel (p + 1)-ary search on PRAM

$\log_{p+1}(n + 1)$
$= \log_2(n + 1) / \log_2(p + 1)$
$= \Theta(\log n / \log p)$ steps

Speedup $\cong \log p$

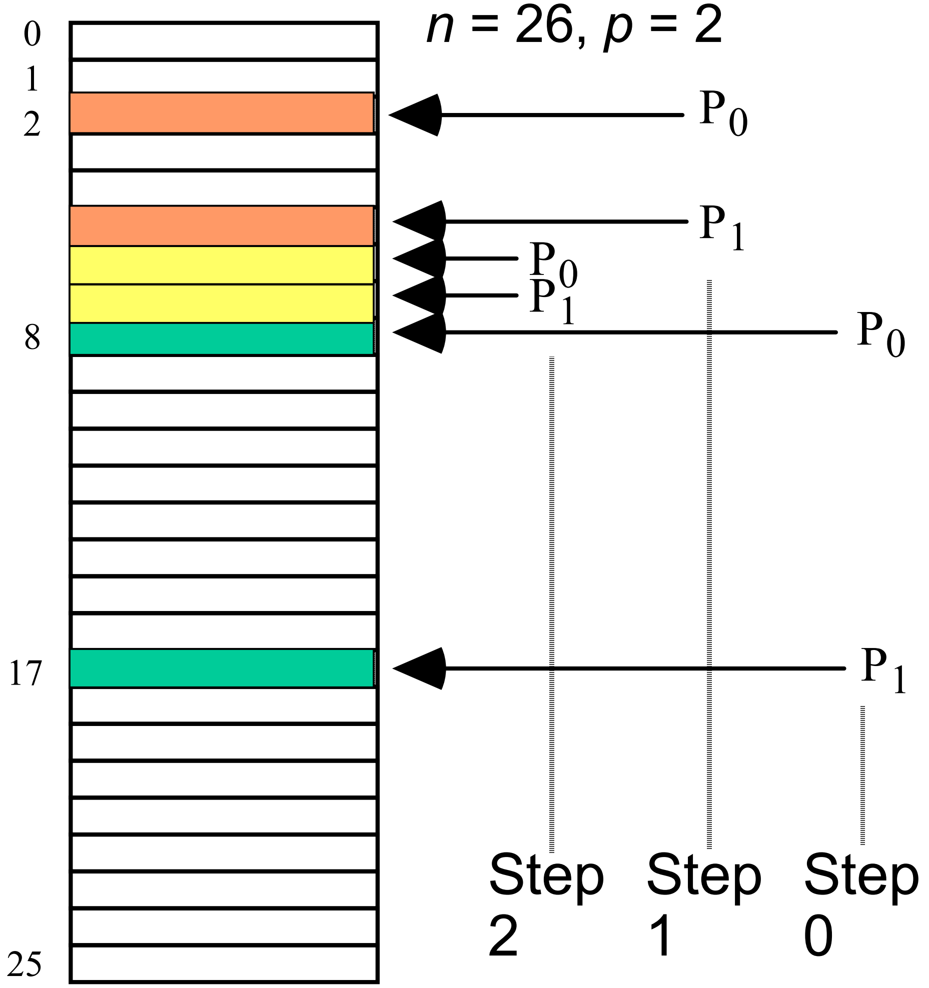
Optimal: no comparison-based search algorithm can be faster

A single search in a sorted list can't be significantly speeded up through parallel processing, but all hope is not lost:

Dynamic data (sorting overhead)

Batch searching (multiple lookups)

Example:
$n = 26$, $p = 2$



Dictionary Operations

Basic dictionary operations: record keys x_0, x_1, . . . , x_{n-1}

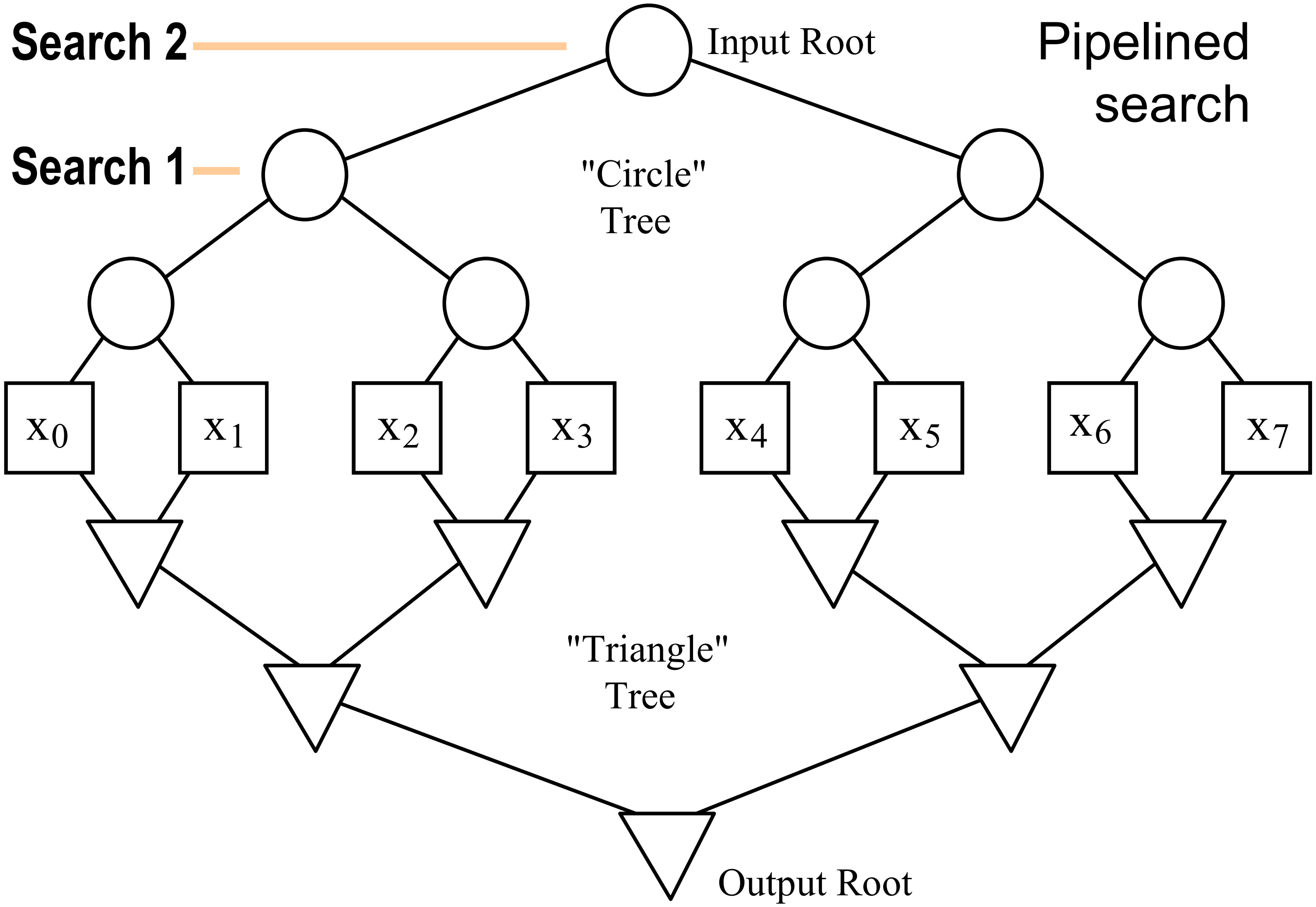
search(*y*)	Find record with key *y*; return its associated data
insert(*y*, *z*)	Augment list with a record: key = *y*, data = *z*
delete(*y*)	Remove record with key *y*; return its associated data

Some or all of the following operations might also be of interest:

findmin	Find record with smallest key; return data
findmax	Find record with largest key; return data
findmed	Find record with median key; return data
findbest(*y*)	Find record with key “nearest” to *y*
findnext(*y*)	Find record whose key is right after *y* in sorted order
findprev(*y*)	Find record whose key is right before *y* in sorted order
extractmin	Remove record(s) with min key; return data
extractmax	Remove record(s) with max key; return data
extractmed	Remove record(s) with median key value; return data

Priority queue operations: *findmin*, *extractmin* (or *findmax*, *extractmax*)

8A.3 Tree-Structured Dictionary Machines



Pipelined search

Fig. 8.1 A tree-structured dictionary machine.

Combining in the triangular nodes

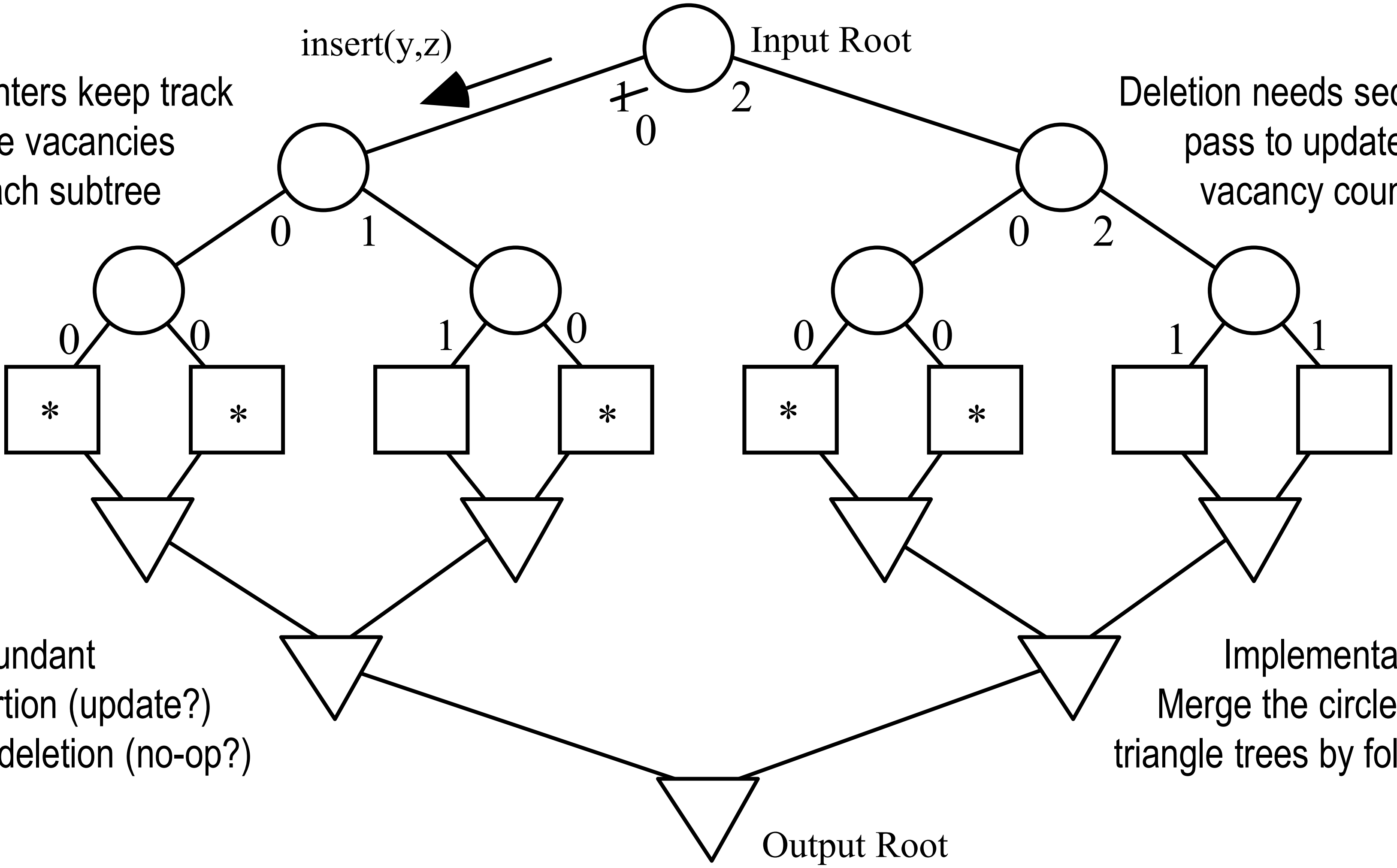
search(*y*): Pass OR of match signals & data from “yes” side

findmin / *findmax*: Pass smaller / larger of two keys & data

findmed: Not supported here

findbest(*y*): Pass the larger of two match-degree indicators along with associated record

Insertion and Deletion in the Tree Machine



Counters keep track of the vacancies in each subtree

Deletion needs second pass to update the vacancy counters

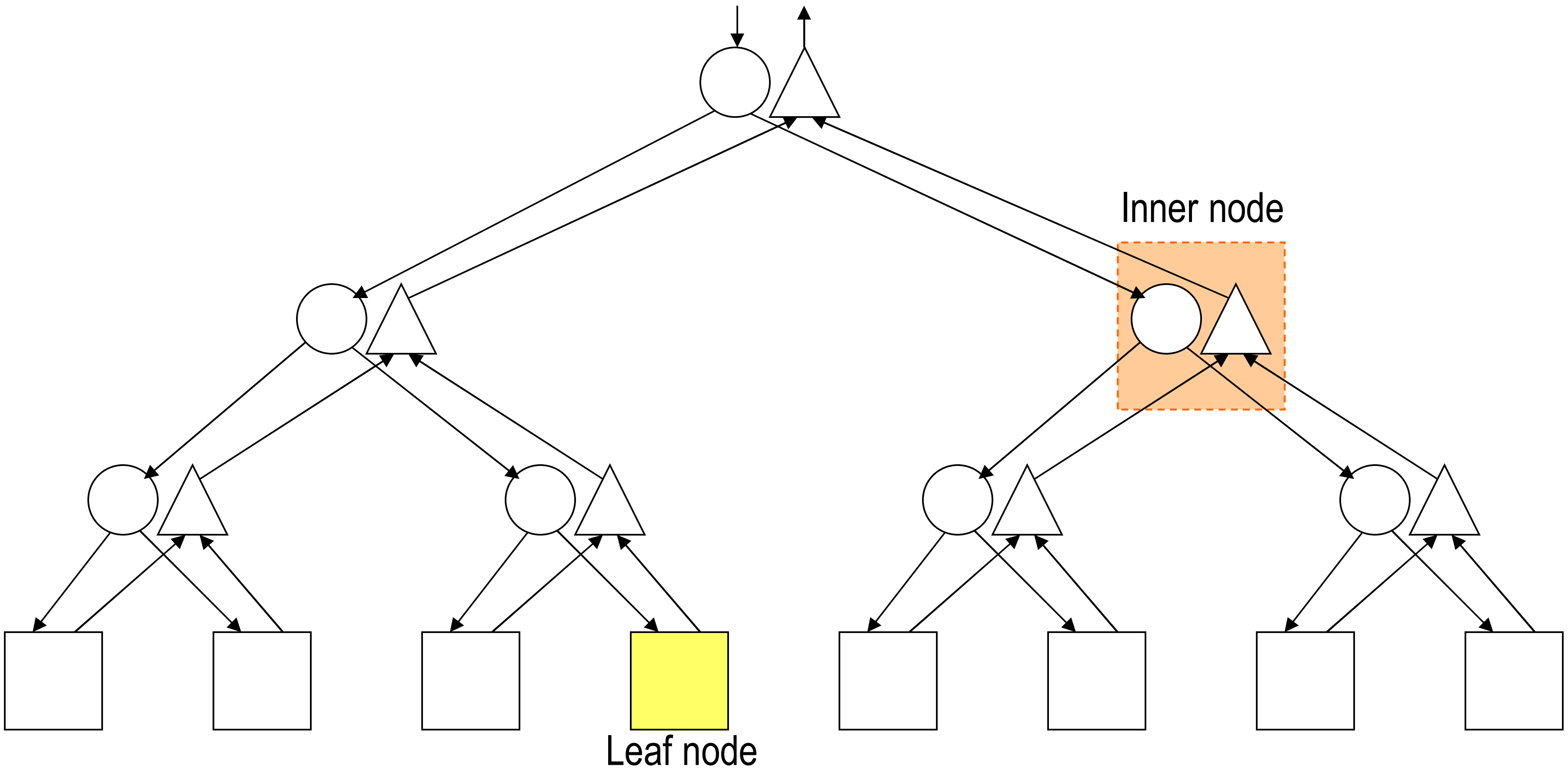
Redundant insertion (update?) and deletion (no-op?)

Implementation: Merge the circle and triangle trees by folding

Figure 8.2 Tree machine storing 5 records and containing 3 free slots.

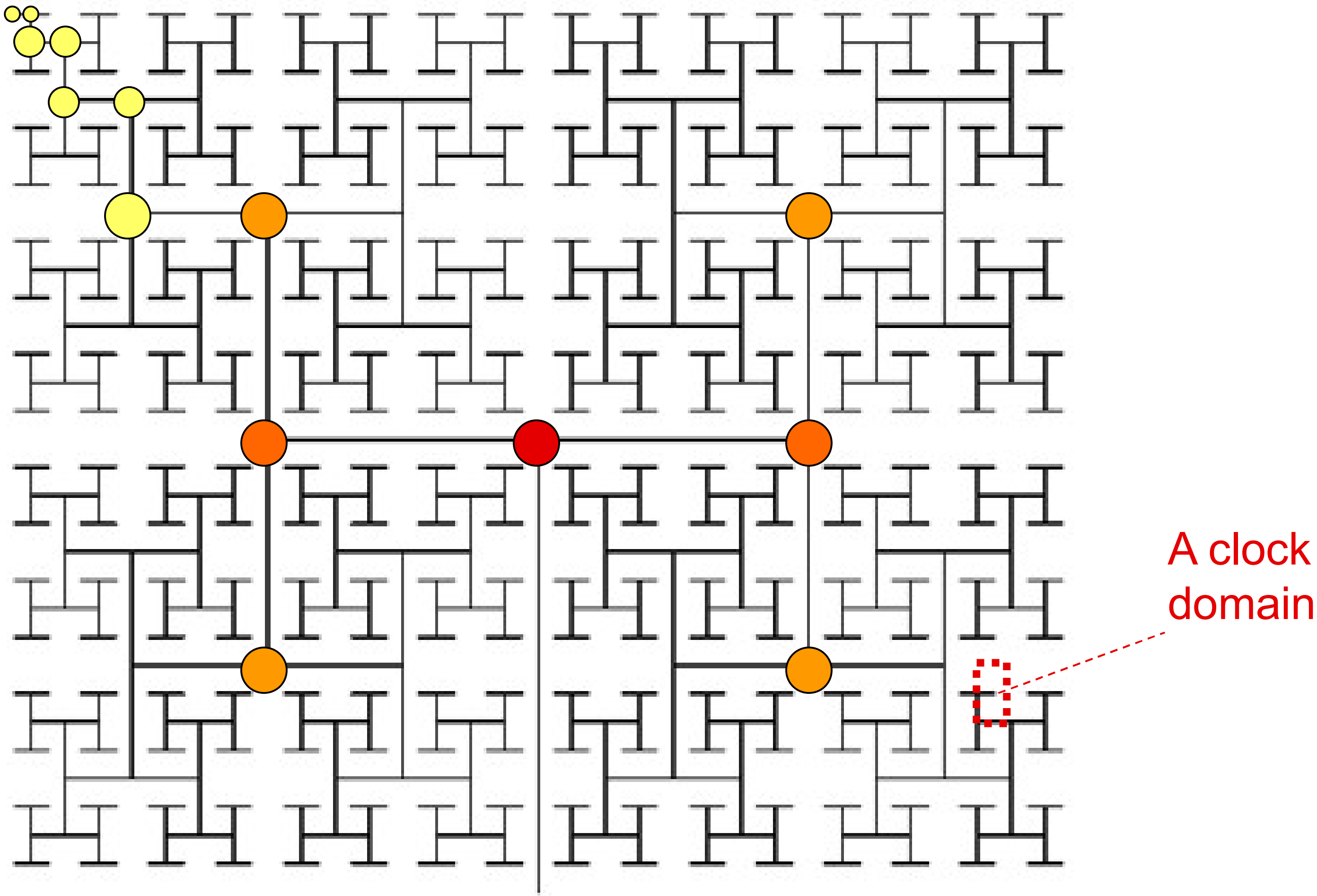
Physical Realization of a Tree Machine

Tree machine in folded form



VLSI Layout of a Tree

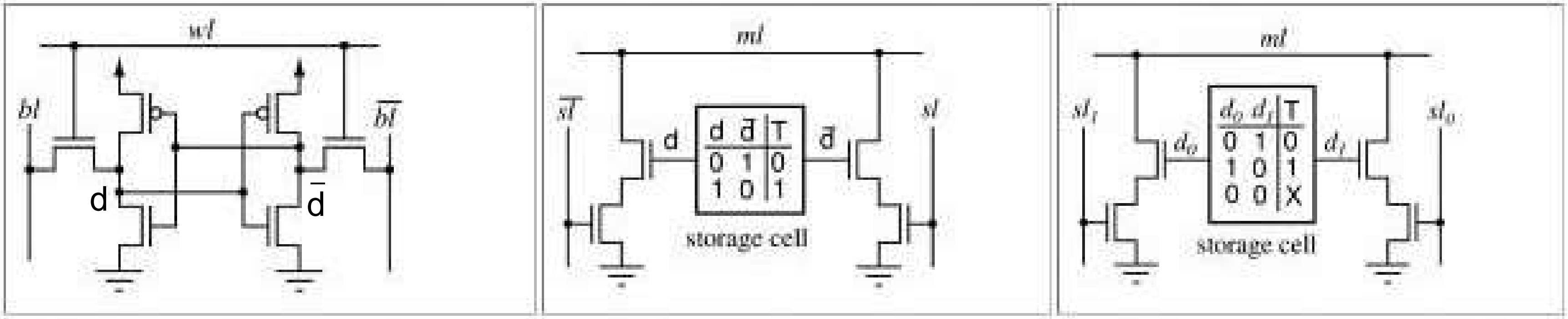
H-tree layout (used, e.g., for clock distribution network in high-performance microchips)



8A.4 Associative Memories

Associative or content-addressable memories (AMs, CAMs)
Binary (BCAM) vs. ternary (TCAM)

Figure 3: Conventional SRAM storage cell, binary CAM and ternary CAM cells. The CAM cells omit the read/write access circuitry for clarity.



(a) 6-transistor SRAM cell. **(b) Binary CAM cell.** **(c) Ternary CAM cell**

Image source: http://www.pagiamtzis.com/cam/camintro.html

Mismatch in cell connects the match line (*ml*) to ground
If all cells in the word match the input pattern, a word match is indicated

Word Match Circuitry

The match line is precharged and then pulled down by any mismatch

Figure 4: The matchline of a NOR-based CAM.

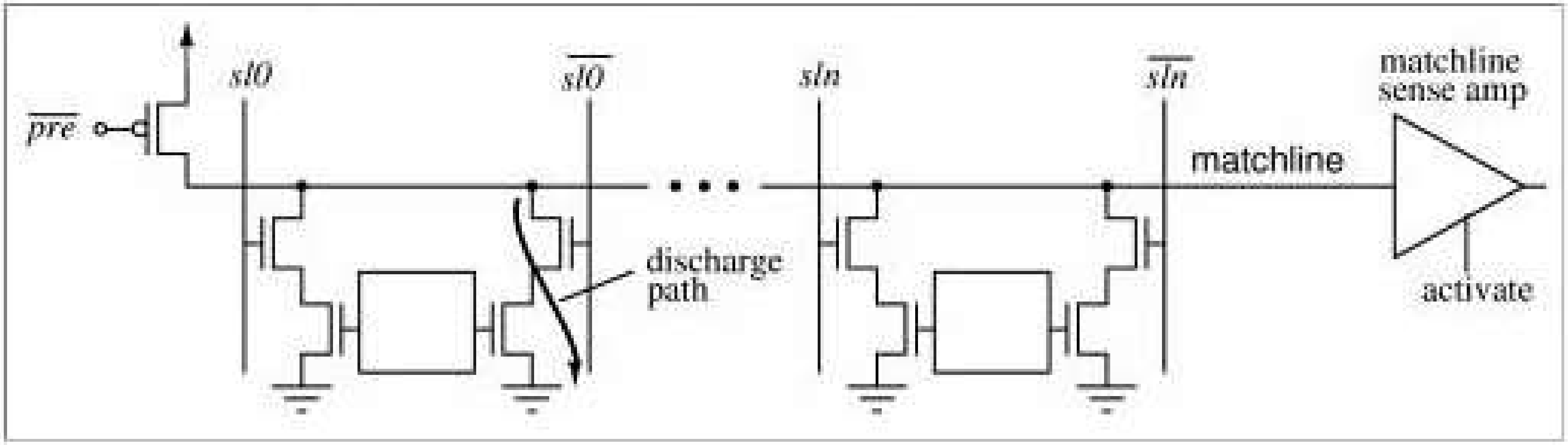


Image source: http://www.pagiamtzis.com/cam/camintro.html

Note that each CAM cell is nearly twice as complex as an SRAM cell
More transistors, more wires

CAM Array Operation

Figure 1: NOR-based CAM architecture (adapted from [Sch96])

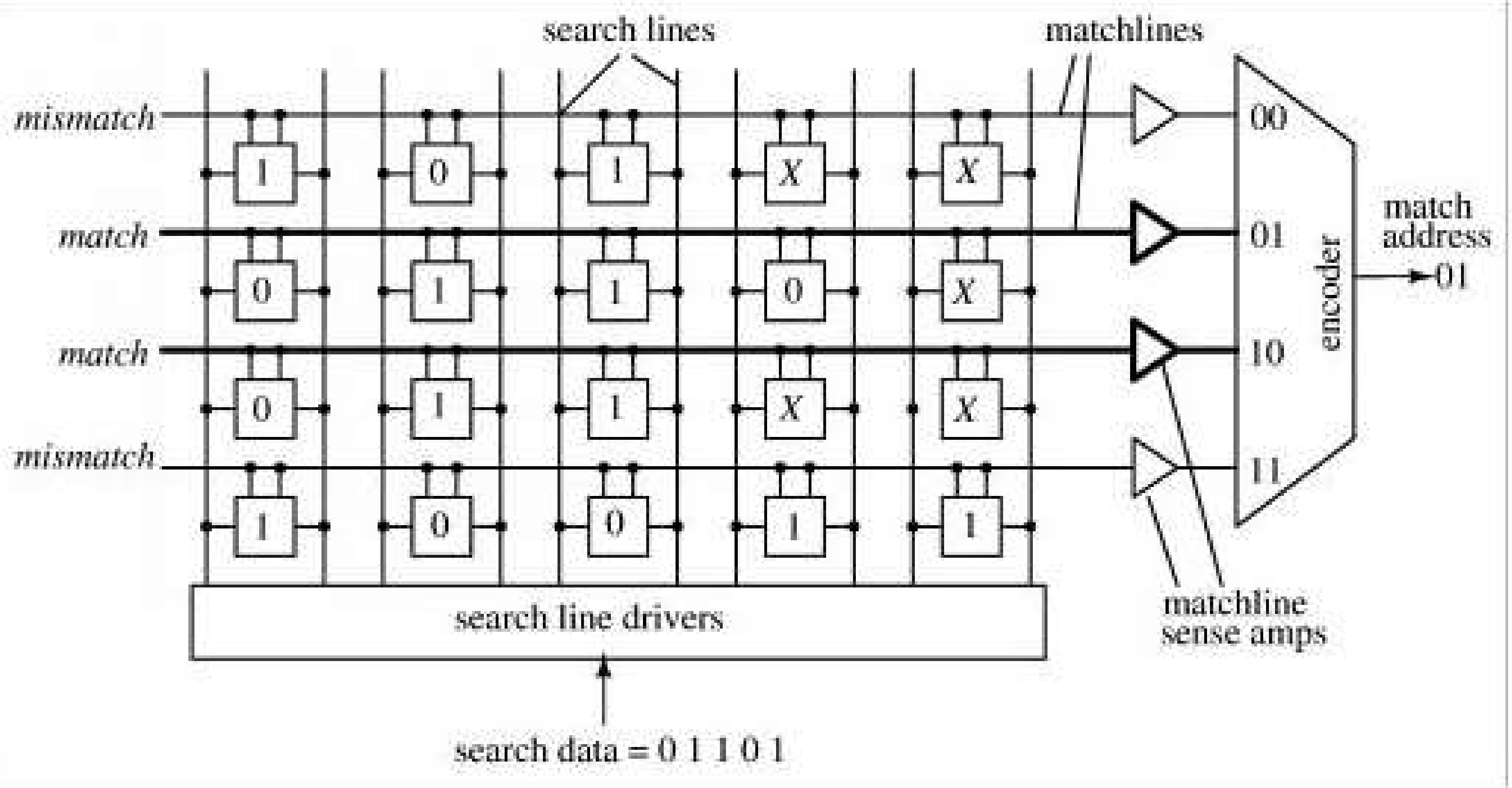


Image source: http://www.pagiamtzis.com/cam/camintro.html

Current CAM Applications

Packet forwarding
Routing tables specify the path to be taken by matching an incoming destination address with stored address prefixes
Prefixes must be stored in order of decreasing length (difficult updating)

Packet classification
Determine packet category based on information in multiple fields
Different classes of packets may be treated differently

Associative caches / TLBs
Main processor caches are usually not fully associative (too large)
Smaller specialized caches and TLBs benefit from full associativity

Data compression
Frequently used substrings are identified and replaced by short codes
Substring matching is accelerated by CAM

History of Associative Processing

Associative memory
- Parallel masked search of all words
- Bit-serial implementation with RAM

Associative processor
- Add more processing logic to PEs

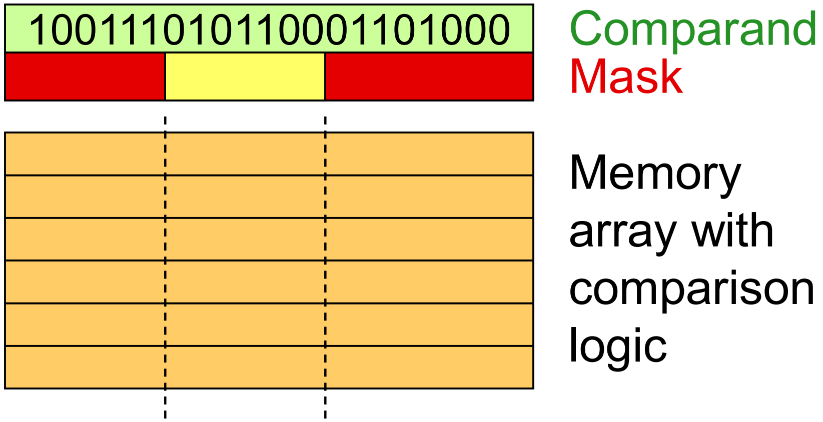


Table 4.1 Entering the second half-century of associative processing

Decade	Events and Advances	Technology	Performance
1940s	Formulation of need & concept	Relays	
1950s	Emergence of cell technologies	Magnetic, Cryogenic	Mega-bit-OPS
1960s	Introduction of basic architectures	Transistors	
1970s	Commercialization & applications	ICs	Giga-bit-OPS
1980s	Focus on system/software issues	VLSI	Tera-bit-OPS
1990s	Scalable & flexible architectures	ULSI, WSI	Peta-bit-OPS

8A.5 Associative Processors

Associative or content-addressable memories/processors constituted early forms of SIMD parallel processing

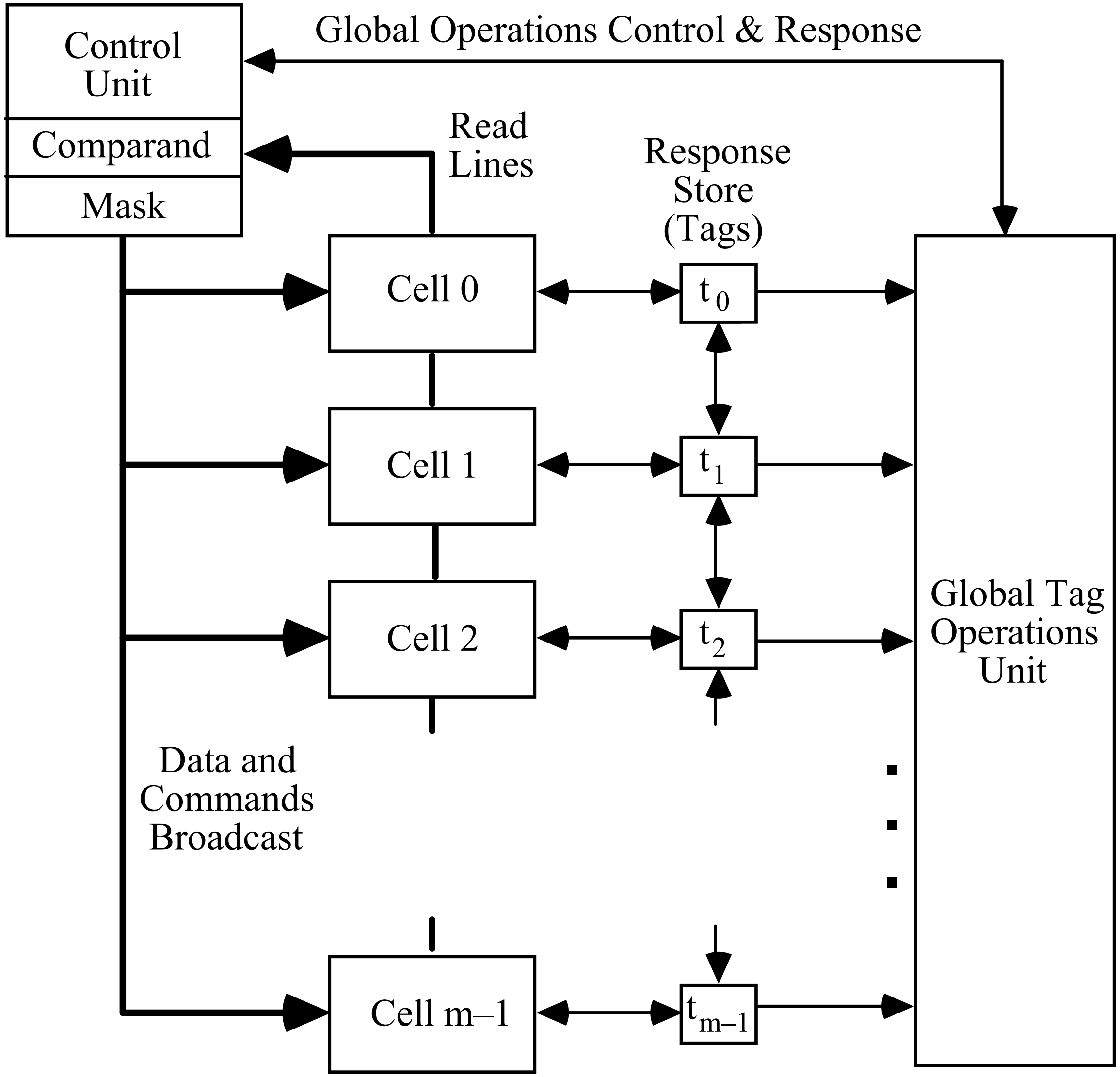


Fig. 23.1 Functional view of an associative memory/processor.

Search Functions in Associative Devices

Exact match: Locating data based on partial knowledge of contents

Inexact match: Finding numerically or logically proximate values

Membership: Identifying all members of a specified set

Relational: Determining values that are less than, less than or equal, etc.

Interval: Marking items that are between or outside given limits

Extrema: Finding the maximum, minimum, next higher, or next lower

Rank-based: Selecting *k*th or *k* largest/smallest elements

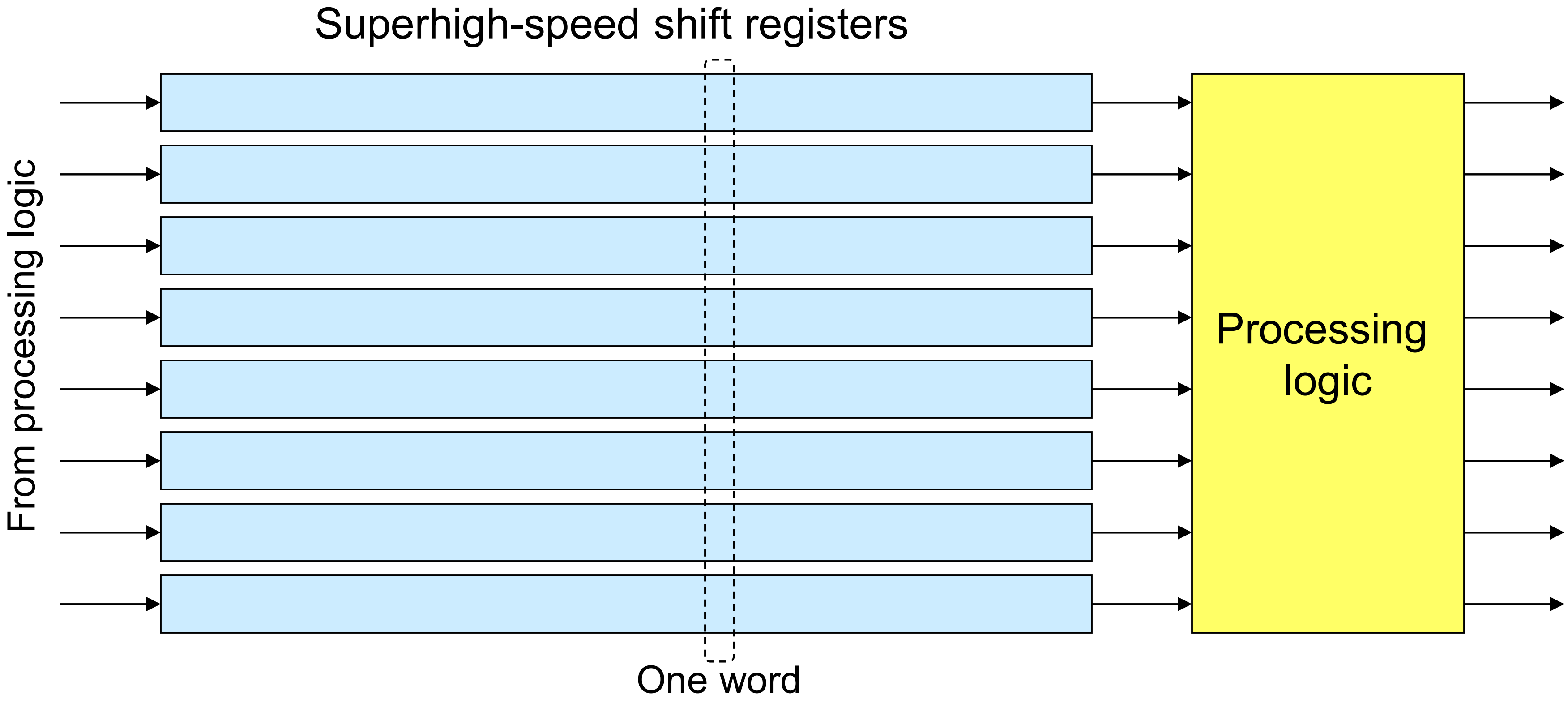
Ordered retrieval: Repeated max- or min-finding with elimination (sorting)

Classification of Associative Devices

		Handling of bits within words	
		Parallel	Serial
Handling of words	Parallel	WPBP: Fully parallel	WPBS: Bit-serial
	Serial	WSBP: Word-serial	WSBS: Fully serial

WSBP: Word-Serial Associative Devices

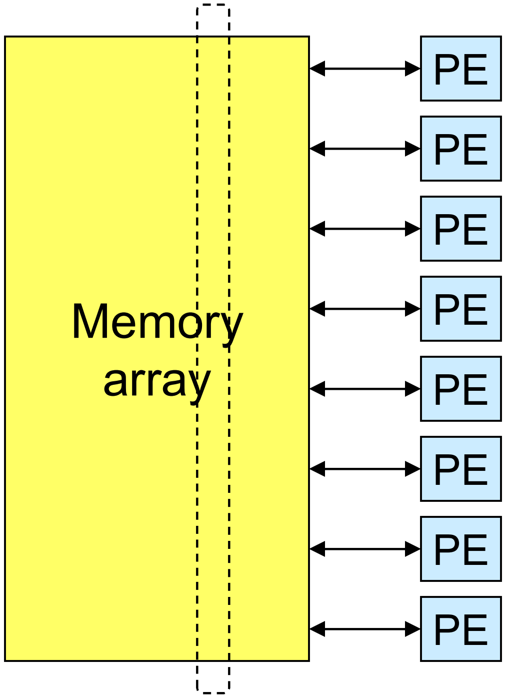
Strictly speaking, this is not a parallel processor, but with superhigh-speed shift registers and deeply pipelined processing logic, it behaves like one



WPBS: Bit-Serial Associative Devices

One bit of every word is processed in one device cycle

Advantages: 1. Can be implemented with conventional memory
2. Easy to add other capabilities beyond search



One bit-slice

Example: Adding field A to field B in every word, storing the sum in field S

```
Loop:
    Read next bit slice of A
    Read next bit slice of B
      (carry from previous slice is in PE flag C)
    Find sum bits; store in next bit slice of S
    Find new carries; store in PE flag
Endloop
```

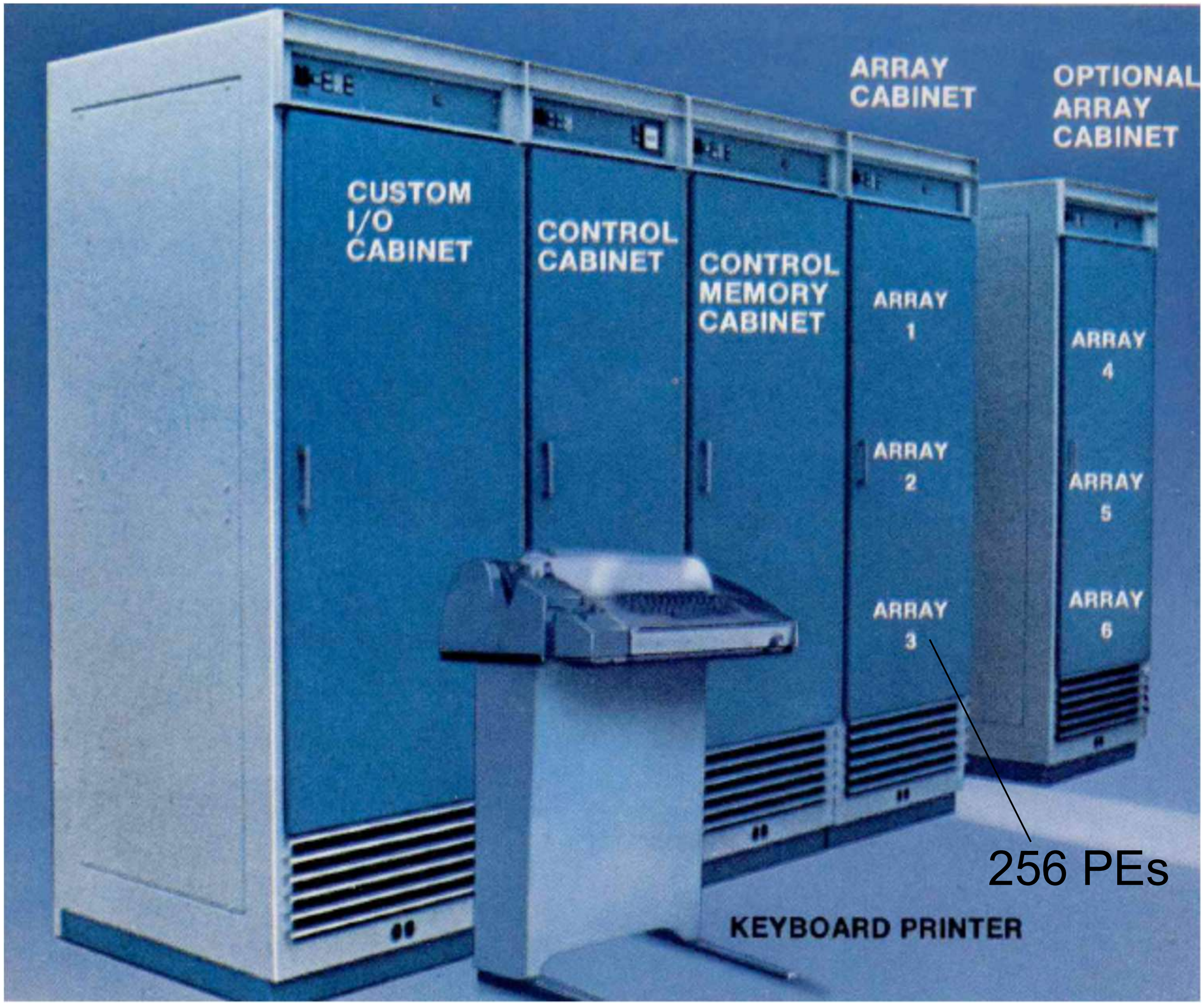
Goodyear STARAN Associative Processor

First computer based on associative memory (1972)

Aimed at air traffic control applications

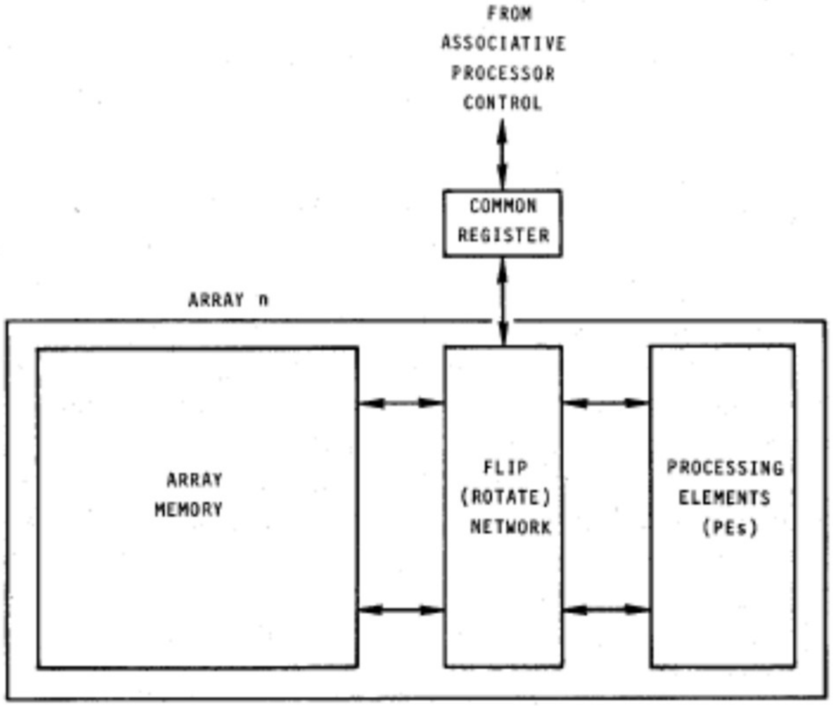
Aircraft conflict detection is an O(n^2) operation

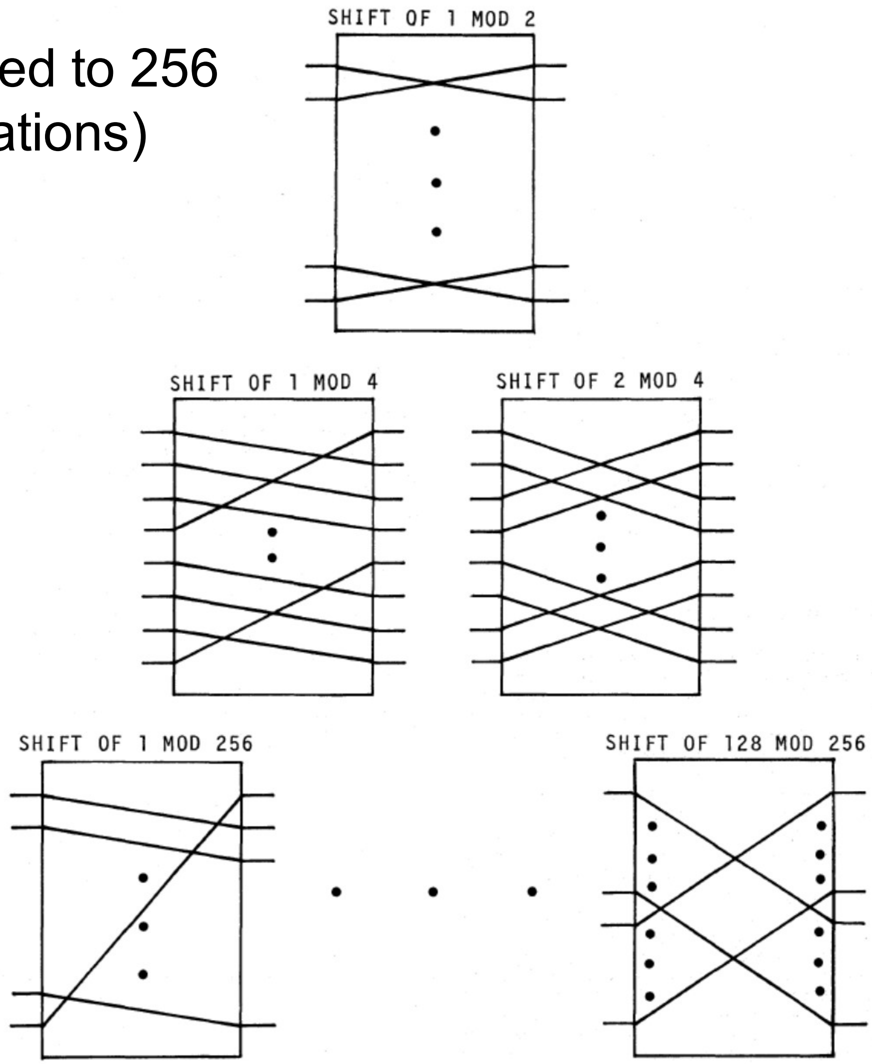
AM can do it in O(n) time



Flip Network Permutations in the Goodyear STARAN

The 256 bits in a bit-slice could be routed to 256 PEs in different arrangements (permutations)





Figs. in this slide from J. Potter, “The STARAN Architecture and Its Applications …,” 1978 NCC

Distributed Array Processor (DAP)

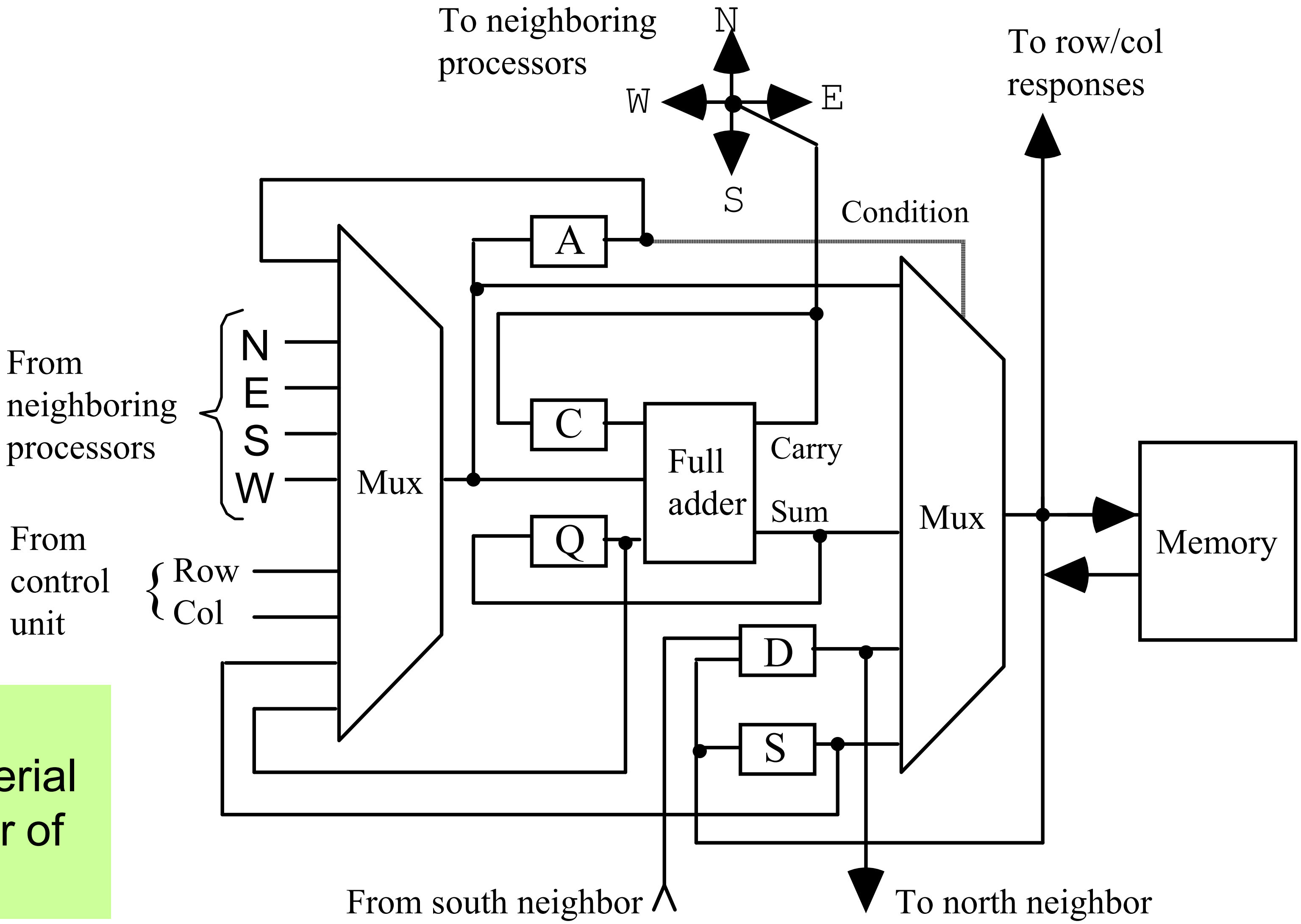


Fig. 23.6 The bit-serial processor of DAP.

DAP’s High-Level Structure

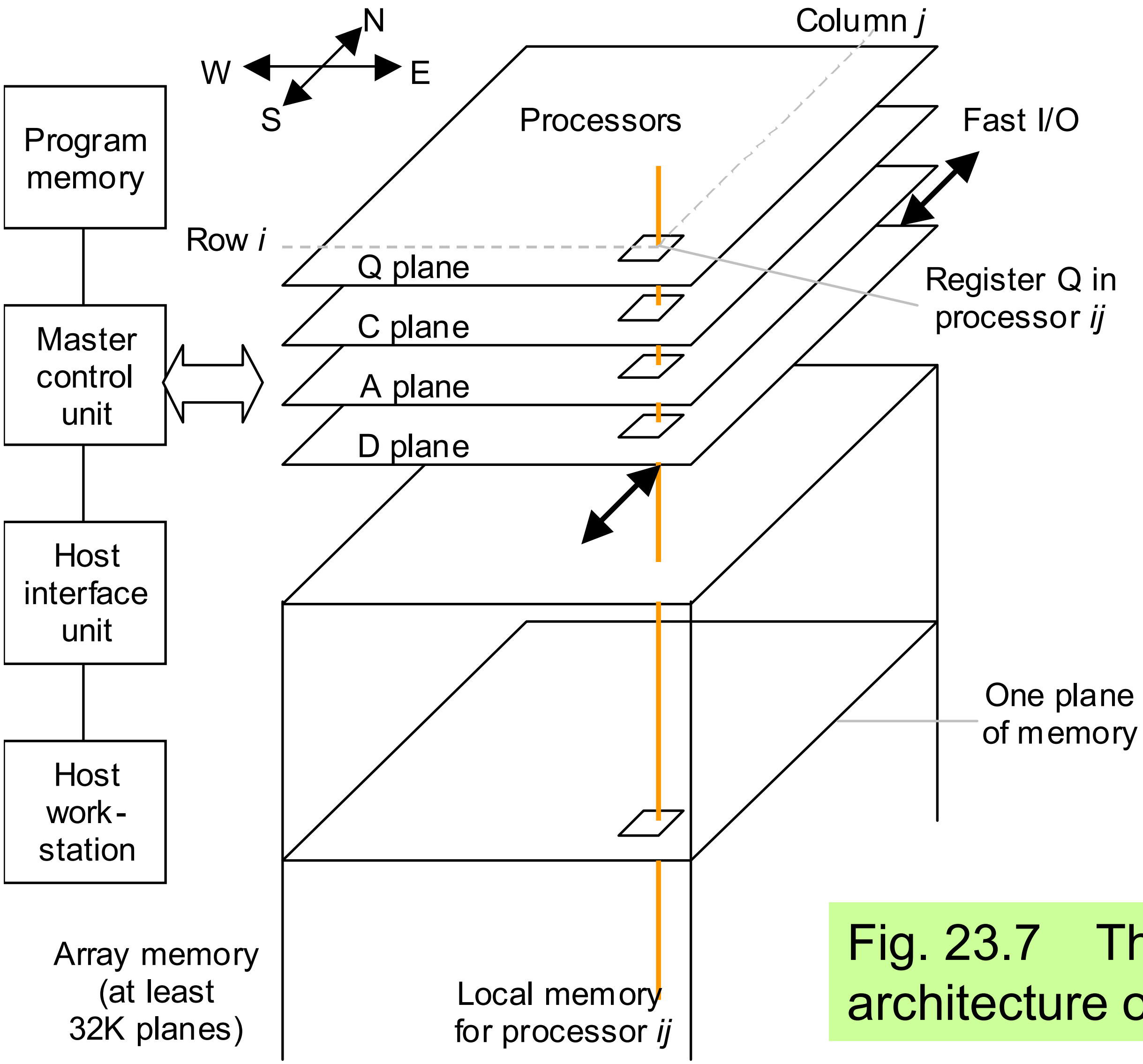


Fig. 23.7 The high-level architecture of DAP system.

8A.6 VLSI Trade-offs in Search Processors

This section has not been written yet

References:

[Parh90] B. Parhami, "Massively Parallel Search Processors: History and Modern Trends," *Proc. 4th Int'l Parallel Processing Symp.*, pp. 91-104, 1990.

[Parh91] B. Parhami, "Scalable Architectures for VLSI-Based Associative Memories," in *Parallel Architectures*, ed. by N. Rishe, S. Navathe, and D. Tal, IEEE Computer Society Press, 1991, pp. 181-200.

8B Arithmetic and Counting Circuits

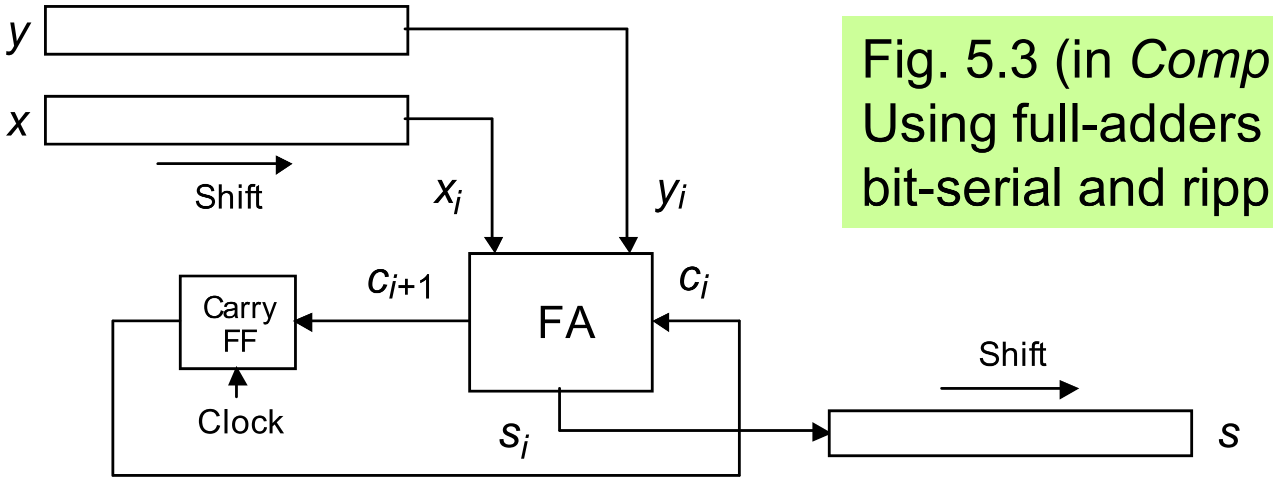
Many parallel processing techniques originate from, or find applications in, designing high-speed arithmetic circuits

- Counting, addition/subtraction, multiplication, division
- Limits on performance and various VLSI trade-offs

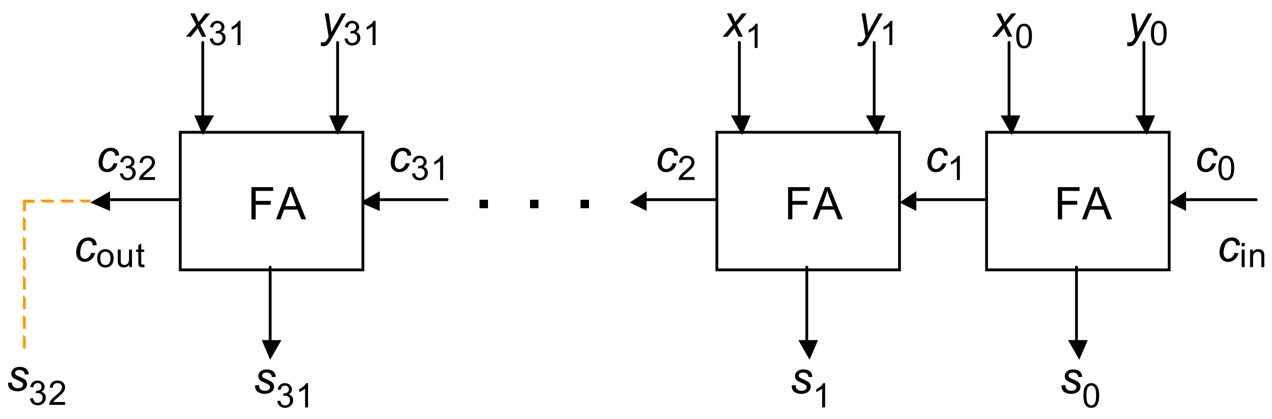
Topics in This Chapter
8B.1 Basic Addition and Counting
8B.2 Circuits for Parallel Counting
8B.3 Addition as a Prefix Computation
8B.4 Parallel Prefix Networks
8B.5 Multiplication and Squaring Circuits
8B.6 Division and Square-Rooting Circuits

8B.1 Basic Addition and Counting

Fig. 5.3 (in *Computer Arithmetic*) Using full-adders in building bit-serial and ripple-carry adders.



(a) Bit-serial adder.



(b) Ripple-carry adder.

Ideal latency:
O(log k)

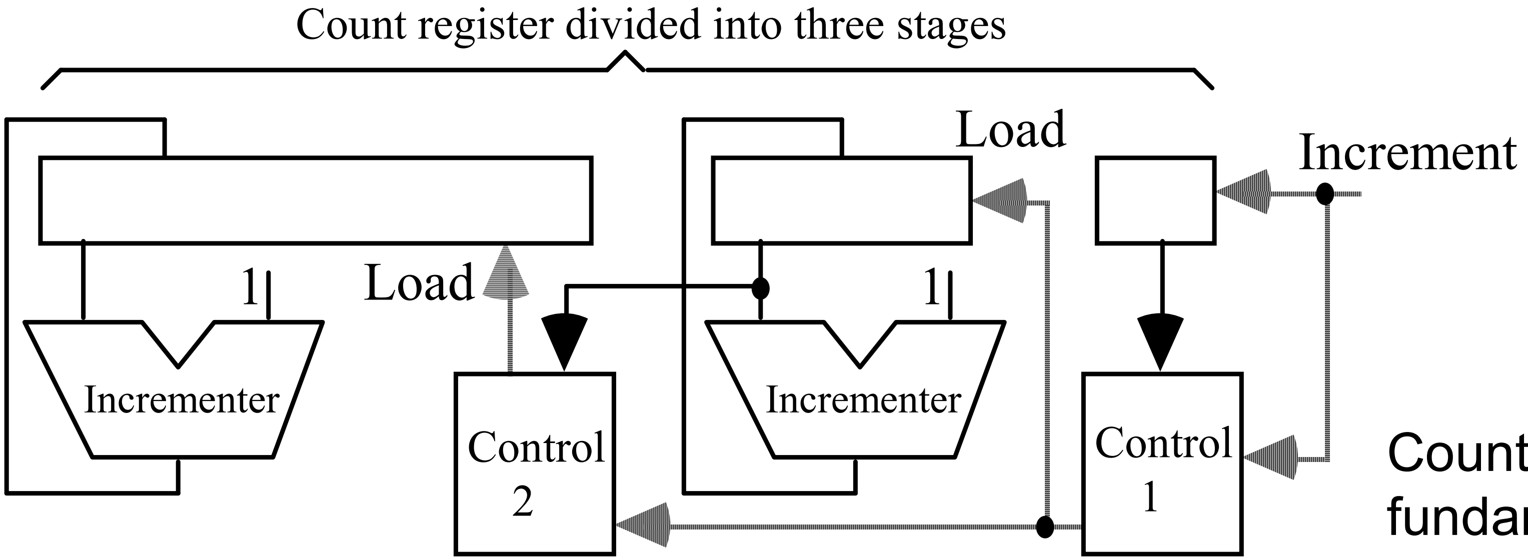
Ideal cost:
O(k)

Can these be achieved simultaneously?

Constant-Time Counters

Any fast adder design can be specialized and optimized to yield a fast counter (carry-lookahead, carry-skip, etc.)

One can use redundant representation to build a constant-time counter, but a conversion penalty must be paid during read-out

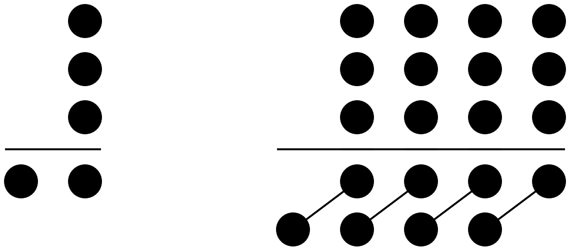


Counting is fundamentally simpler than addition

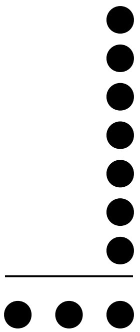
Fig. 5.12 (in *Computer Arithmetic*)
Fast (constant-time) three-stage up counter.

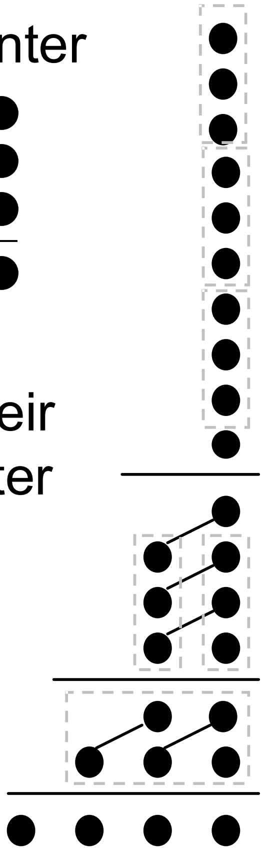
8B.2 Circuits for Parallel Counting

1-bit full-adder = (3; 2)-counter



Circuit reducing 7 bits to their
3-bit sum = (7; 3)-counter





Circuit reducing *n* bits to their
$\lceil \log_2(n + 1) \rceil$-bit sum
$= (n; \lceil \log_2(n+1) \rceil)$-counter

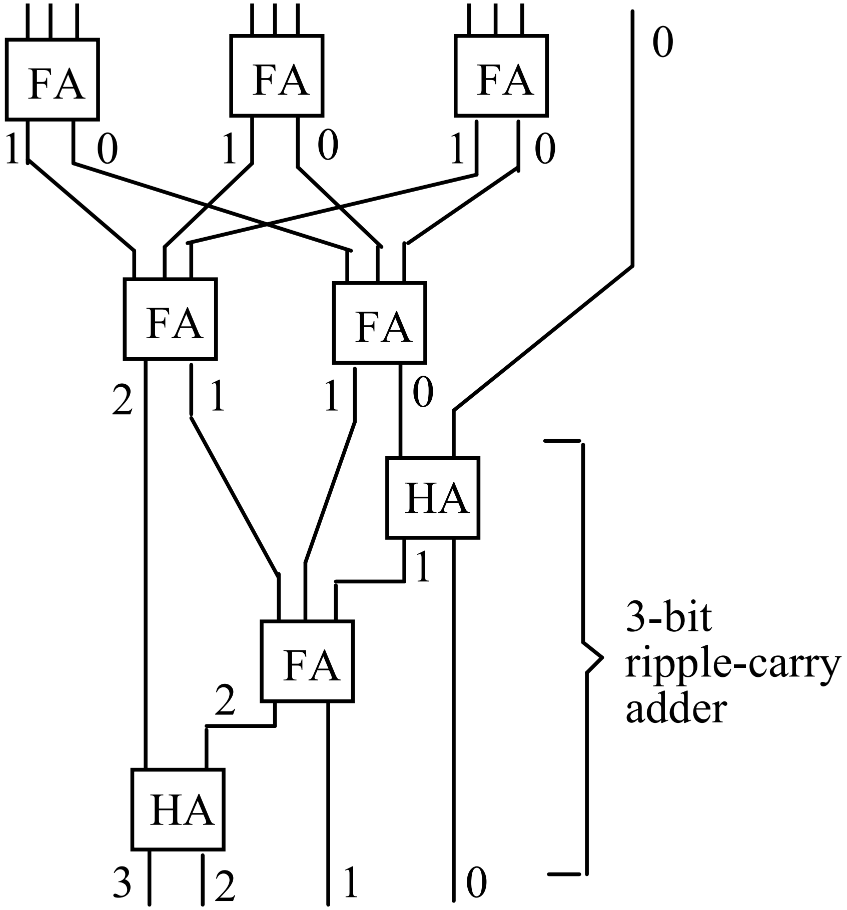
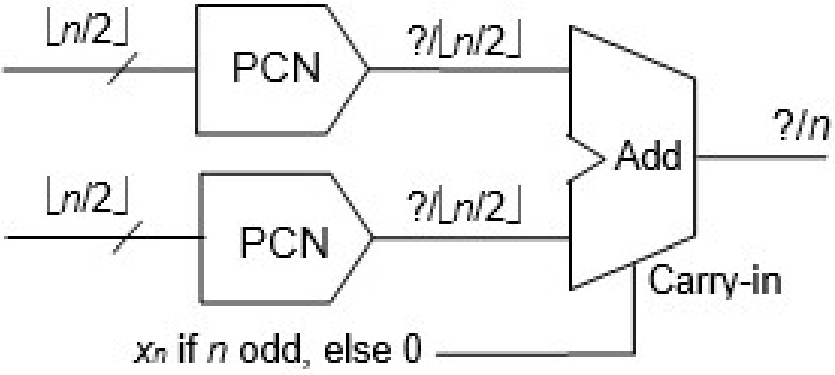
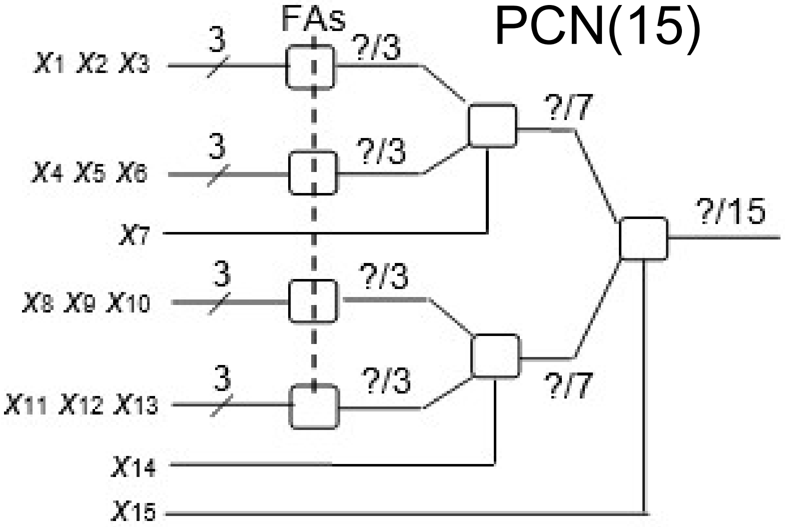


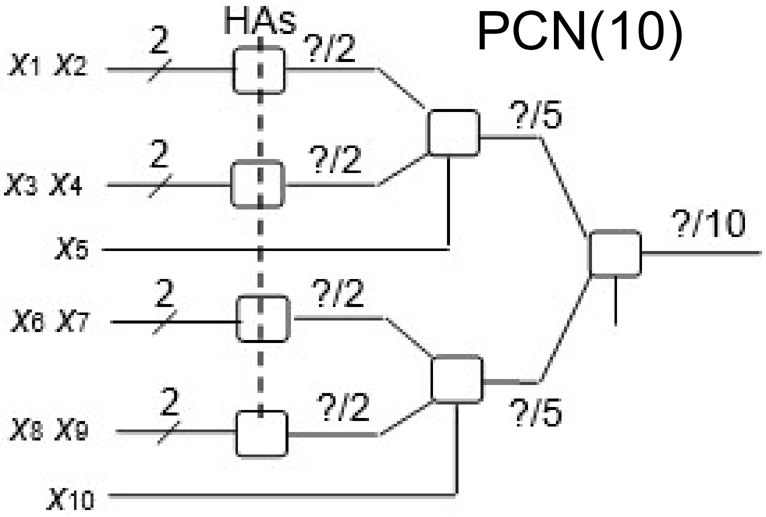
Fig. 8.16 (in *Computer Arithmetic*)
A 10-input parallel counter also known as a (10; 4)-counter.

Recursive Construction of Parallel Counters

An *n*-input parallel counting network (PCN) can be built from two ⌊*n*/2⌋-bit parallel counting networks and a ⌊$\log_2 n$⌋-bit adder

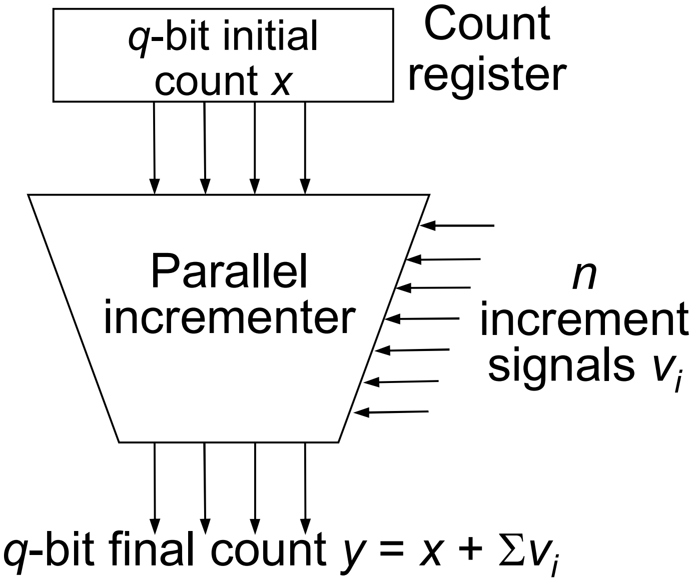






Accumulative Parallel Counters

True generalization of sequential counters



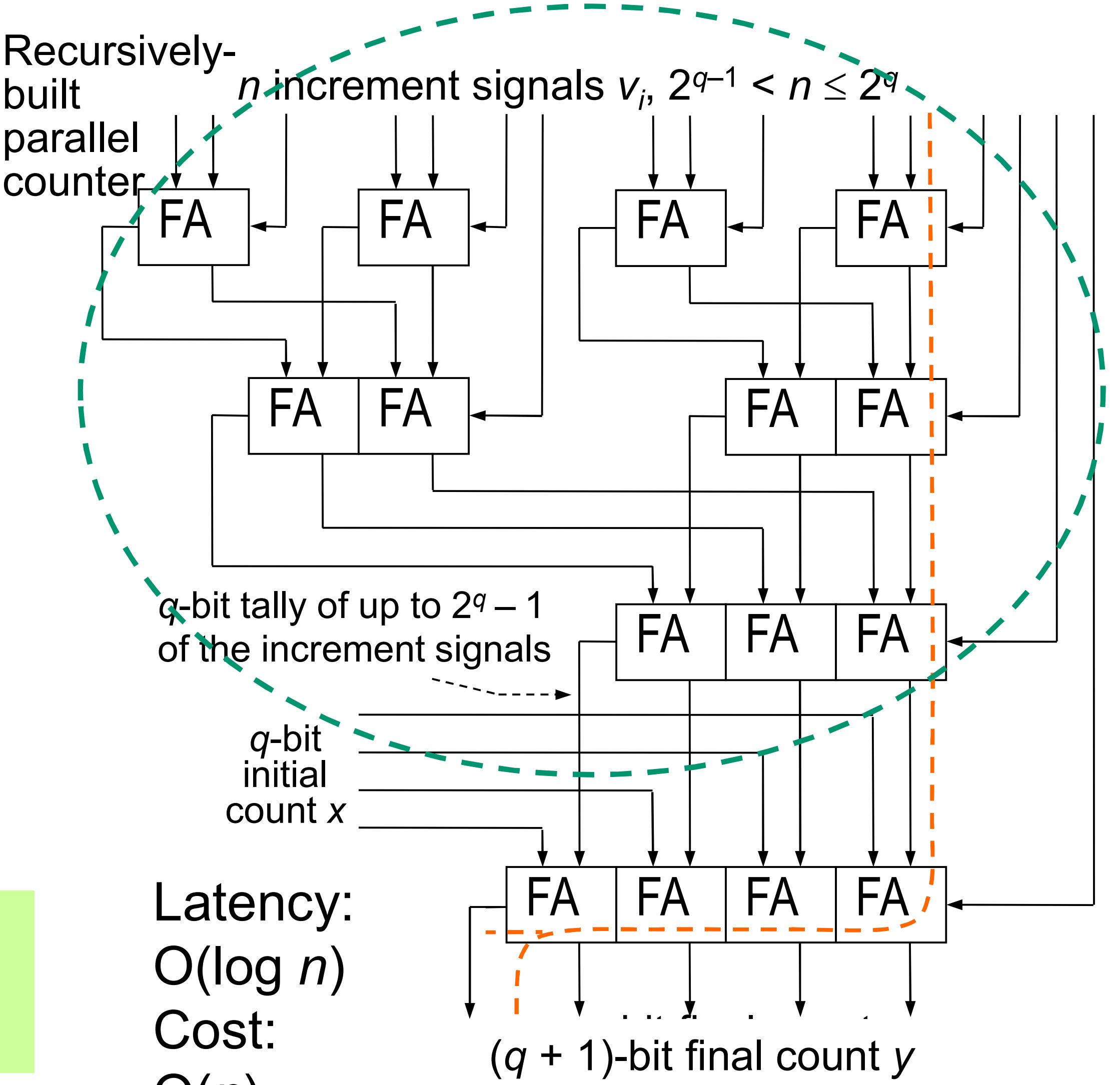
q-bit initial count *x*

Count register

Parallel incrementer

n increment signals v_i

q-bit final count $y = x + \Sigma v_i$



Recursively-built parallel counter

n increment signals v_i, $2^{q-1} < n \le 2^q$

q-bit tally of up to $2^q - 1$ of the increment signals

q-bit initial count *x*

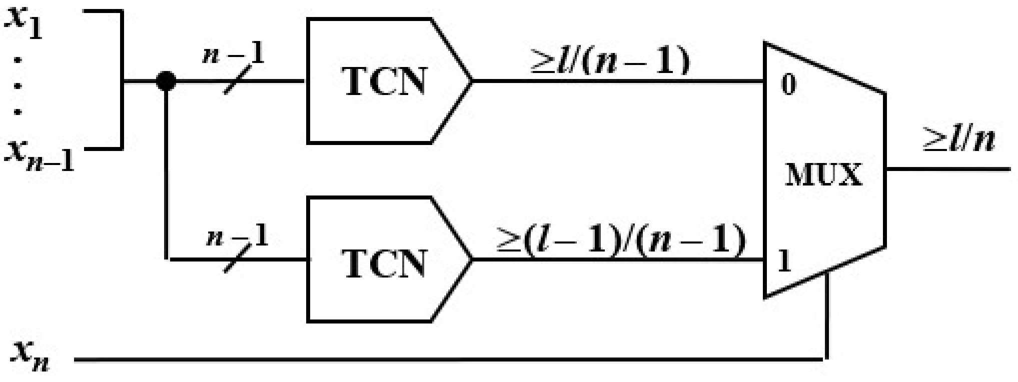
(q + 1)-bit final count *y*

Latency: O(log *n*)

Cost: O(*n*)

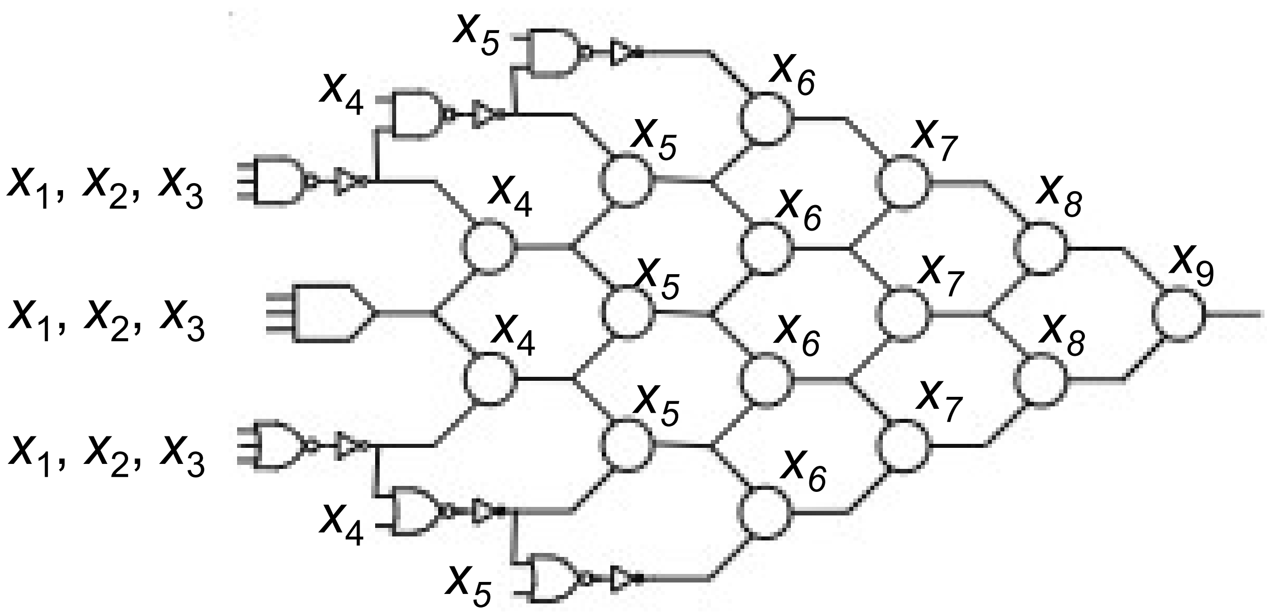
Possible application: Compare Hamming weight of a vector to a constant

Threshold Counting Networks



At-least-*l*-out-of-*n* threshold counting network built from a multiplexer and two smaller threshold counting networks

Recursively-built 5-out-of-9 voter



8B.3 Addition as a Prefix Computation

Example: Prefix sums

x_0	x_1	x_2	. . .	x_i
x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	. . .	$x_0 + x_1 + . . . + x_i$
s_0	s_1	s_2	. . .	s_i

Sequential time with one processor is O(n)

Simple pipelining does not help

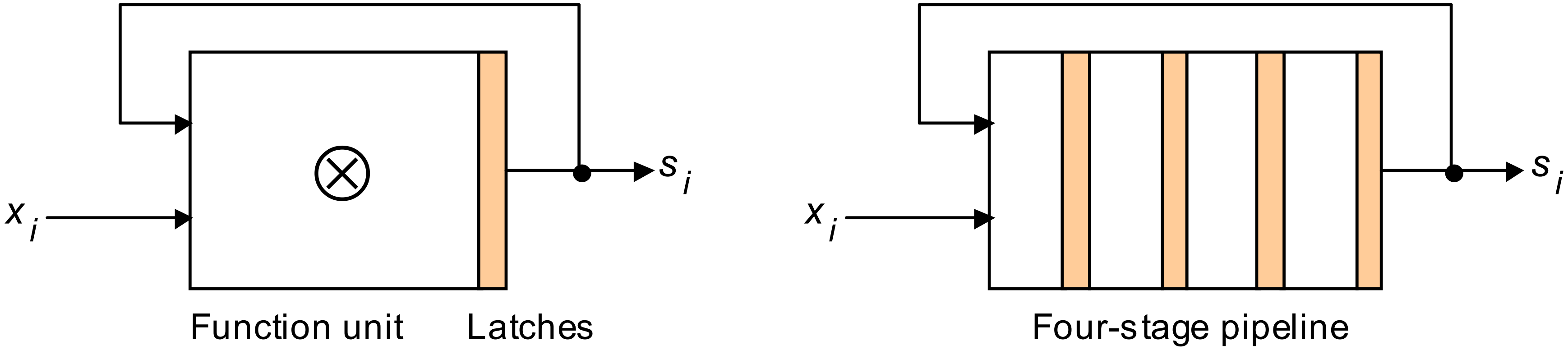
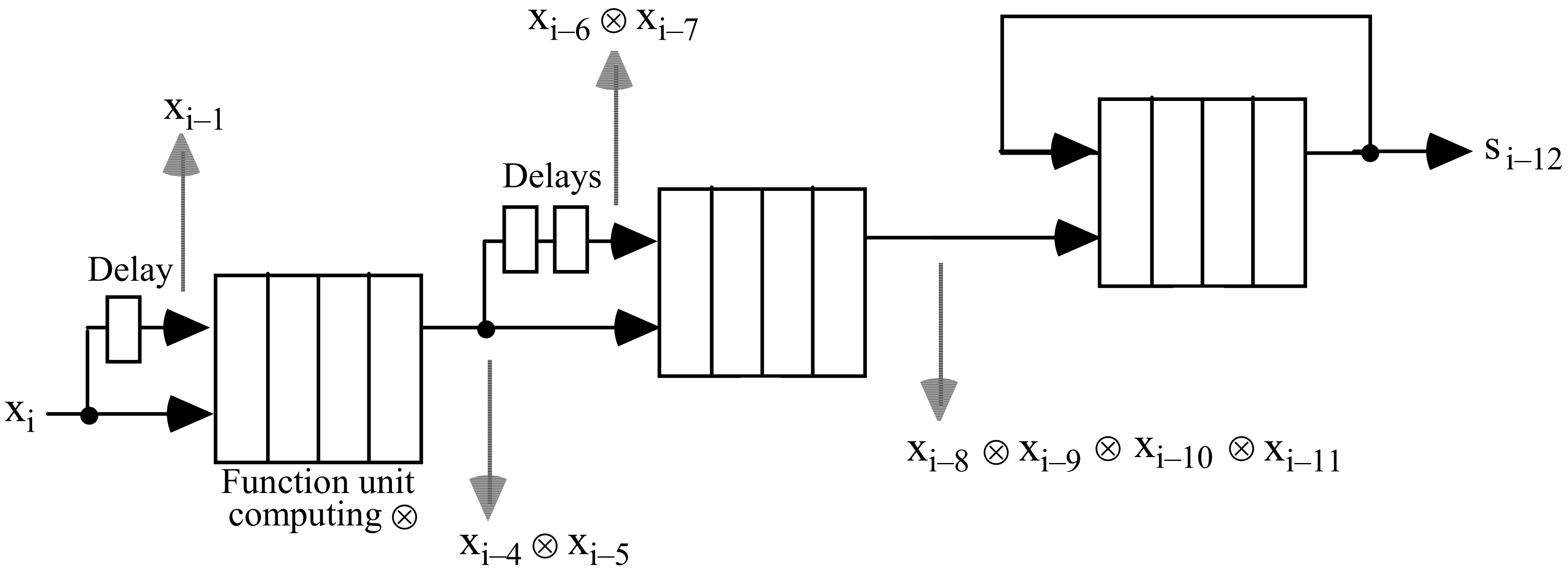


Fig. 8.4 Prefix computation using a latched or pipelined function unit.

Improving the Performance with Pipelining

Ignoring pipelining overhead, it appears that we have achieved a speedup of 4 with 3 “processors.” Can you explain this anomaly? (Problem 8.6a)

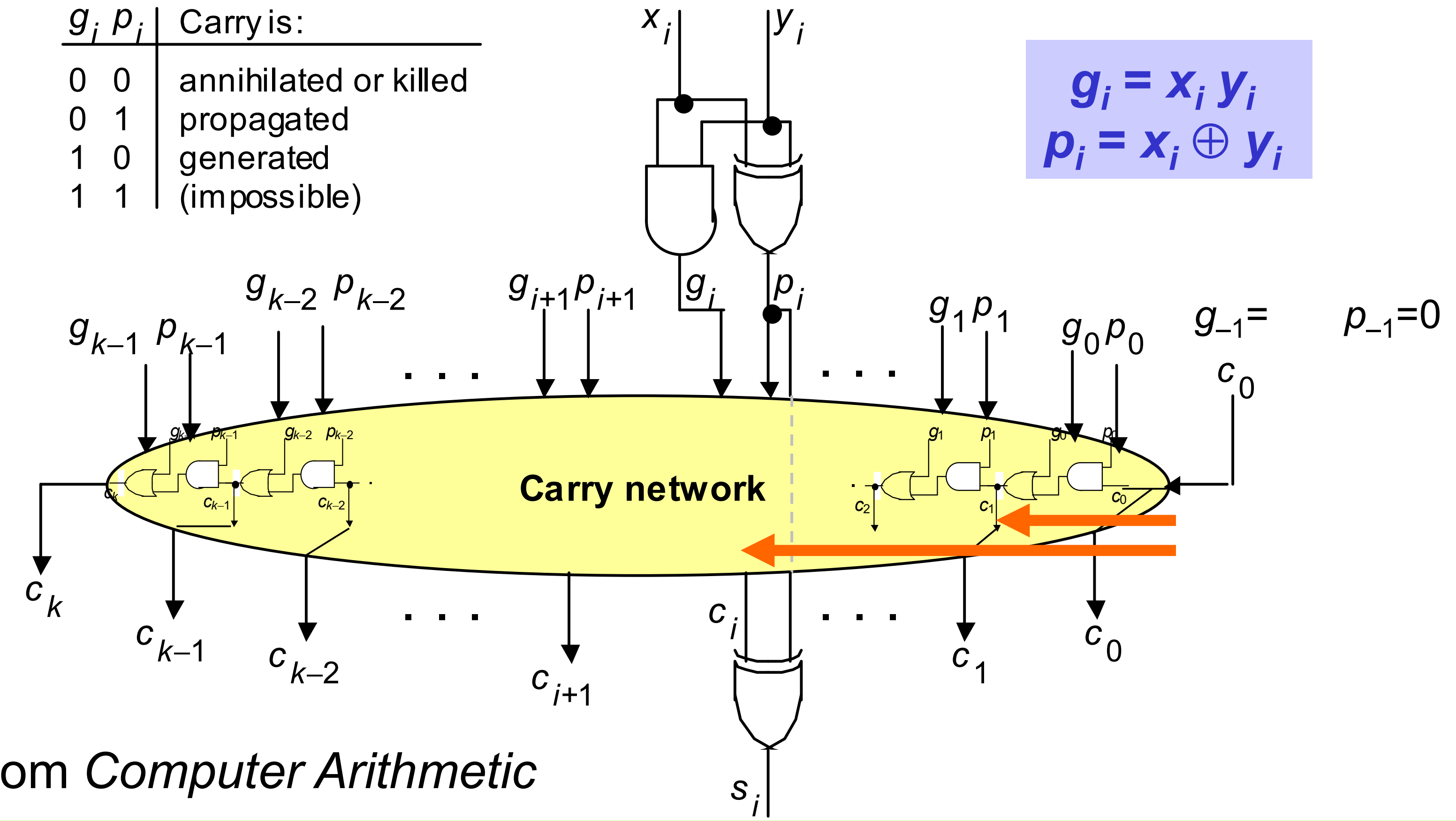


$x_{i-6} \otimes x_{i-7}$
x_{i-1}
Delays
s_{i-12}
Delay
x_i
Function unit computing ⊗
$x_{i-4} \otimes x_{i-5}$
$x_{i-8} \otimes x_{i-9} \otimes x_{i-10} \otimes x_{i-11}$

Fig. 8.5 High-throughput prefix computation using a pipelined function unit.

Carry Determination as a Prefix Computation

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)



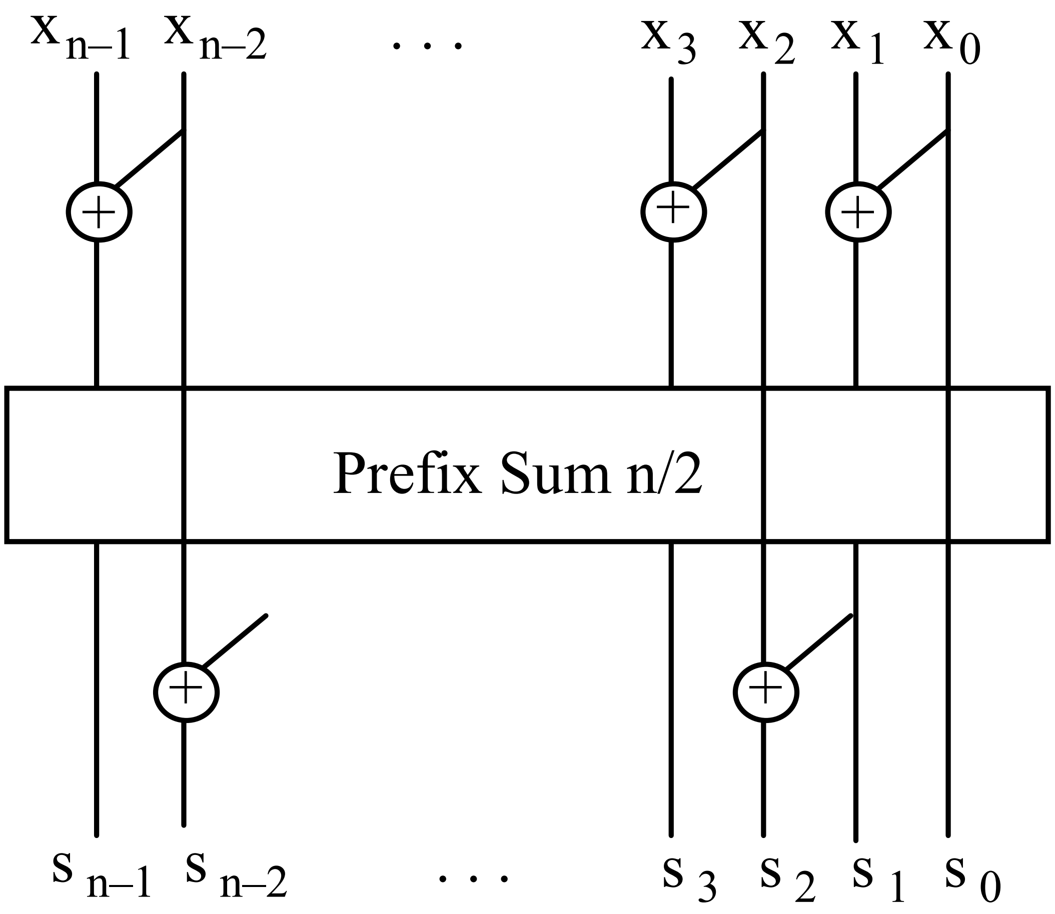
$g_i = x_i\, y_i$

$p_i = x_i \oplus y_i$

Figure from *Computer Arithmetic*

Fig. 5.15 (ripple-carry network) superimposed on Fig. 5.14 (generic adder).

8B.4 Parallel Prefix Networks



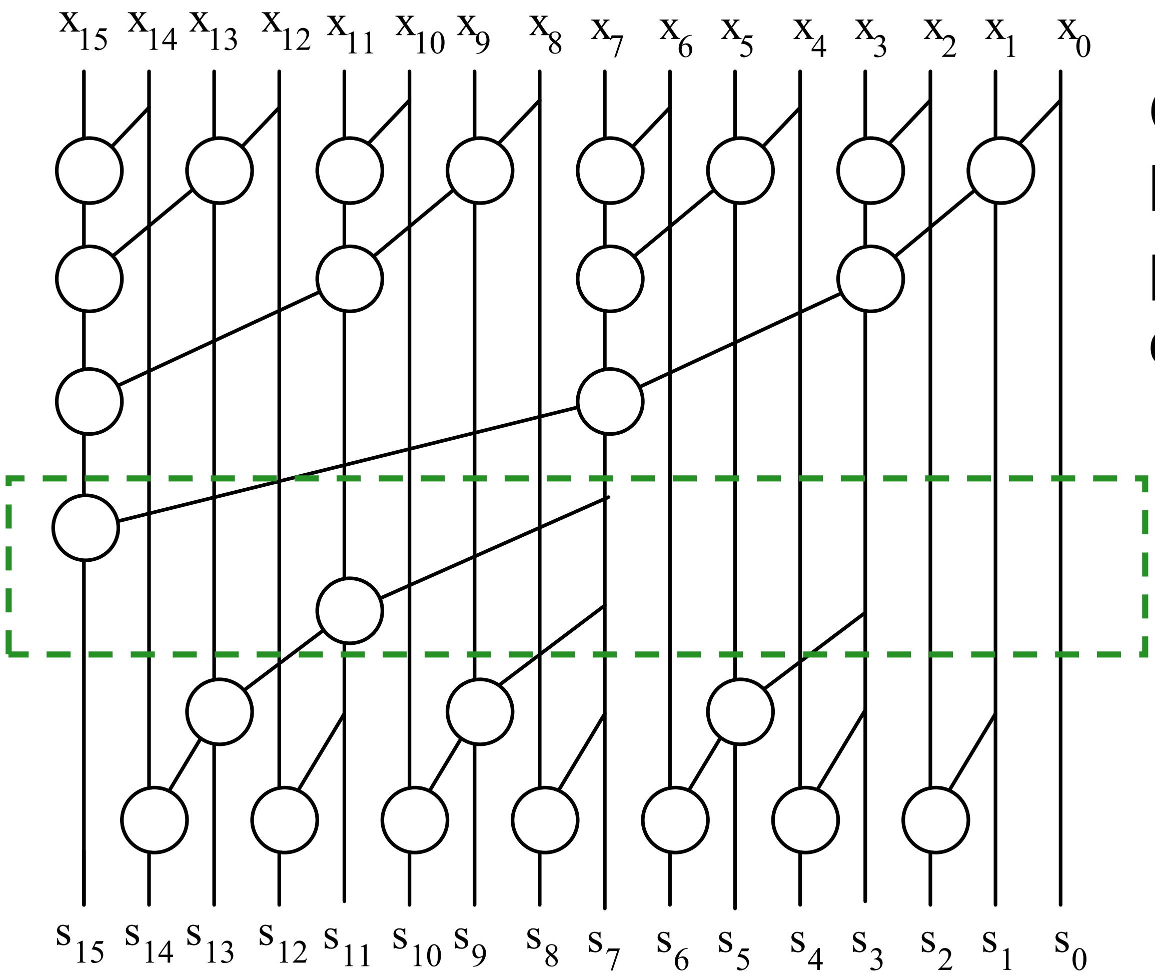
$$T(n) = T(n/2) + 2$$
$$= 2 \log_2 n - 1$$

$$C(n) = C(n/2) + n - 1$$
$$= 2n - 2 - \log_2 n$$

This is the Brent-Kung Parallel prefix network (its delay is actually $2 \log_2 n - 2$)

Fig. 8.6 Prefix sum network built of one $n/2$-input network and $n - 1$ adders.

Example of Brent-Kung Parallel Prefix Network



x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Originally developed by Brent and Kung as part of a VLSI-friendly carry lookahead adder

One level of latency

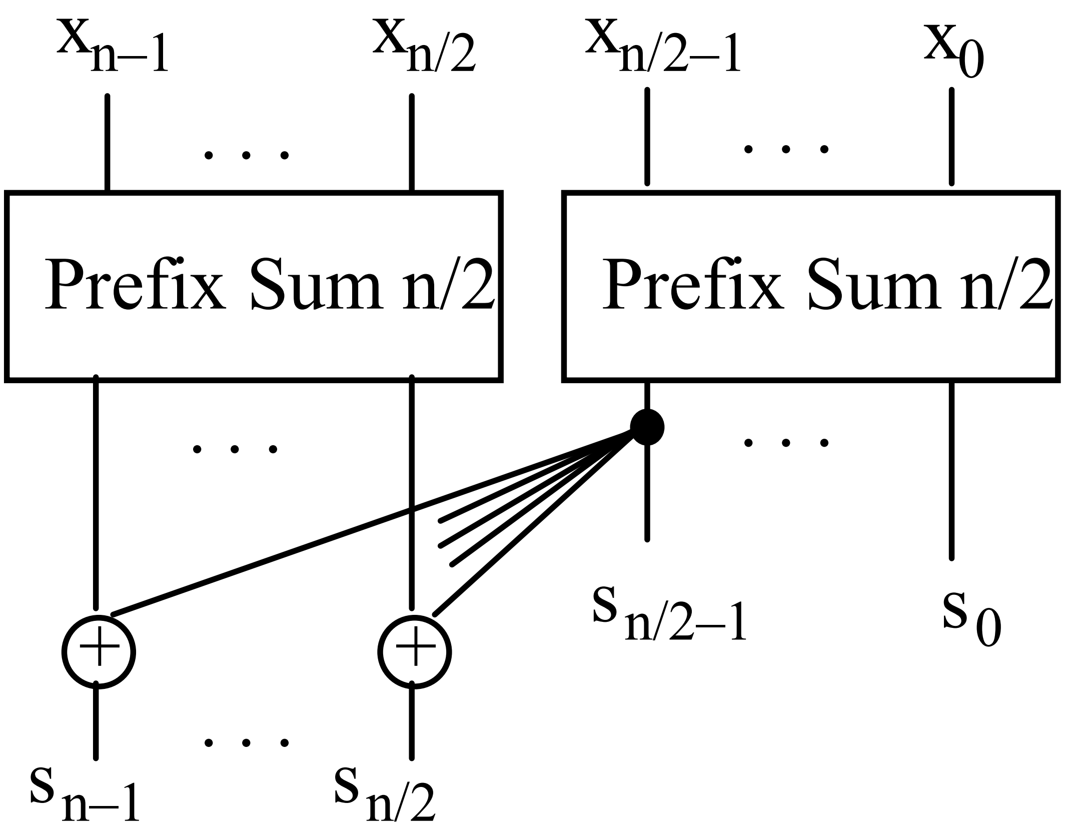
$$T(n) = 2 \log_2 n - 2$$

$$C(n) = 2n - 2 - \log_2 n$$

Fig. 8.8 Brent–Kung parallel prefix graph for *n* = 16.

Another Divide-and-Conquer Design

Ladner-Fischer construction



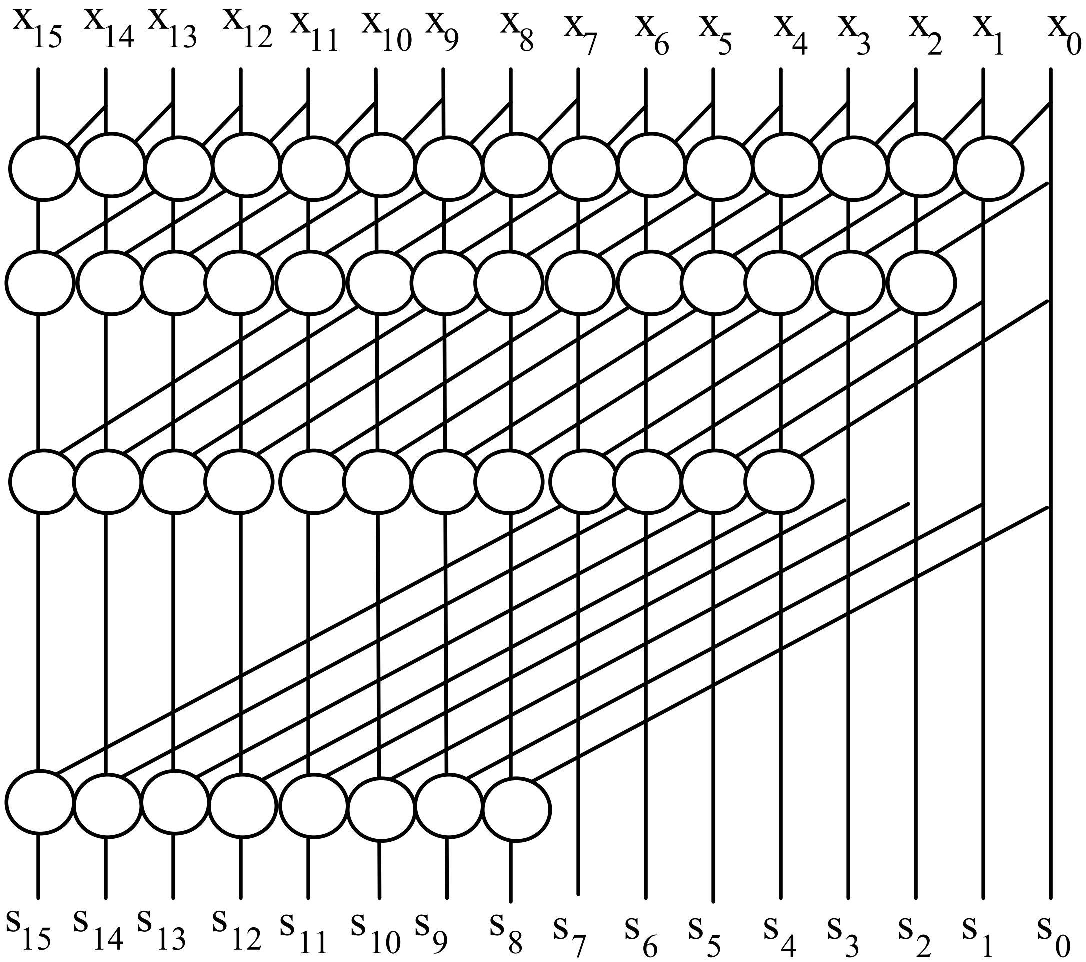
$$T(n) = T(n/2) + 1 = \log_2 n$$

$$C(n) = 2C(n/2) + n/2 = (n/2) \log_2 n$$

Simple Ladner-Fisher Parallel prefix network (its delay is optimal, but has fan-out issues if implemented directly)

Fig. 8.7 Prefix sum network built of two *n*/2-input networks and *n*/2 adders.

Example of Kogge-Stone Parallel Prefix Network


x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

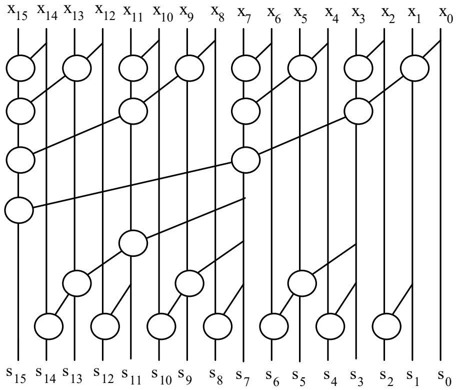
$$T(n) = \log_2 n$$

$$\begin{aligned} C(n) &= (n-1) + (n-2) \\ &\quad + (n-4) + \ldots + n/2 \\ &= n \log_2 n - n - 1 \end{aligned}$$

Optimal in delay,
but too complex
in number of cells
and wiring pattern

Fig. 8.9 Kogge-Stone parallel prefix graph for $n = 16$.

Comparison and Hybrid Parallel Prefix Networks


x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Brent/Kung
6 levels
26 cells

Kogge/Stone
4 levels
49 cells

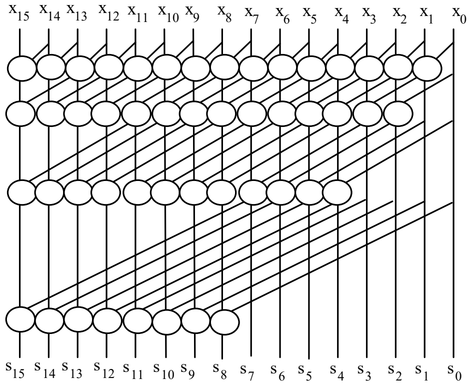
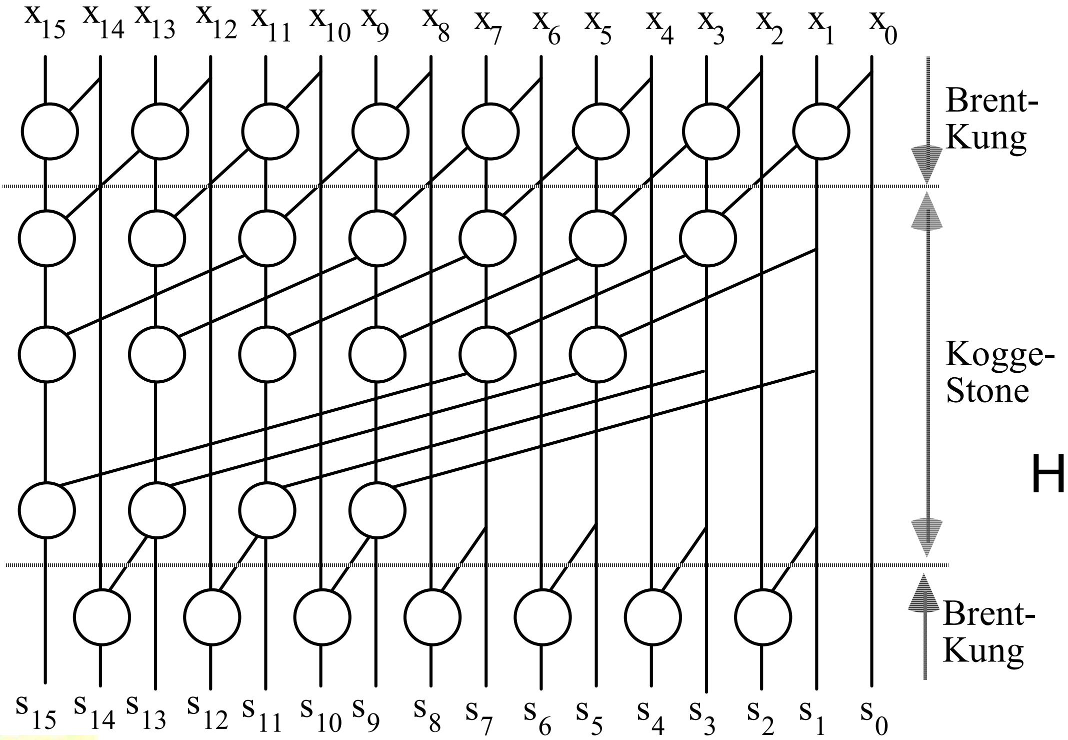

x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Fig. 8.10 A hybrid Brent–Kung / Kogge–Stone parallel prefix graph for *n* = 16.


x_{15} x_{14} x_{13} x_{12} x_{11} x_{10} x_9 x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Brent-Kung
Kogge-Stone
Brent-Kung
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

Han/Carlson
5 levels
32 cells

Linear-Cost, Optimal Ladner-Fischer Networks

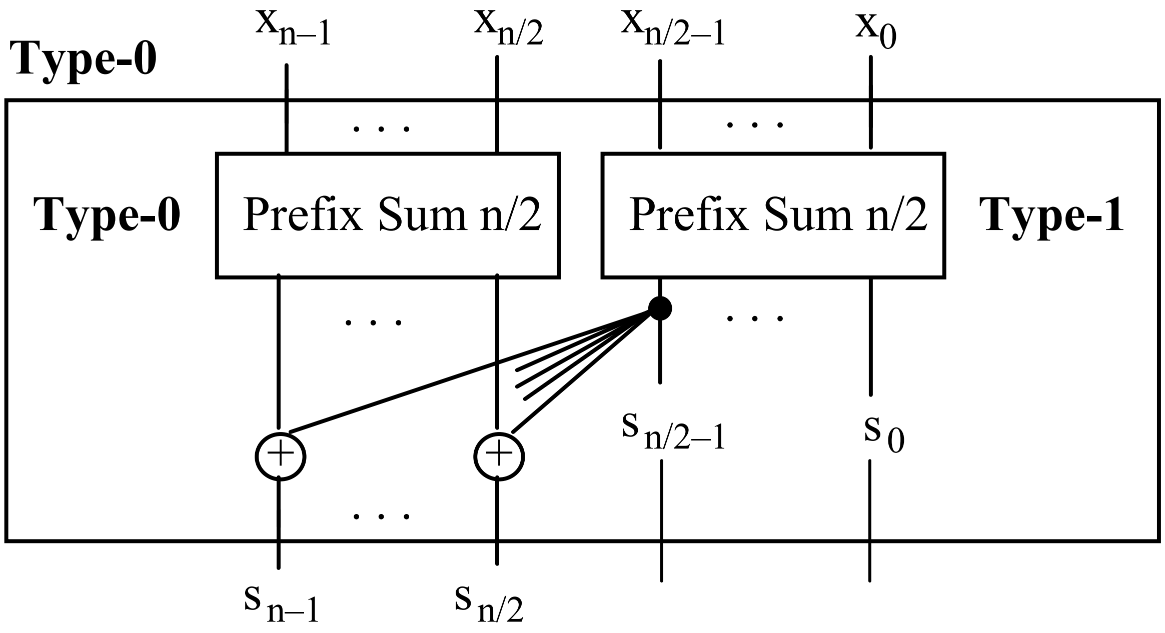
Define a type-*x* parallel prefix network as one that:

Produces the leftmost output in optimal $\log_2 n$ time

Yields all other outputs with at most *x* additional delay

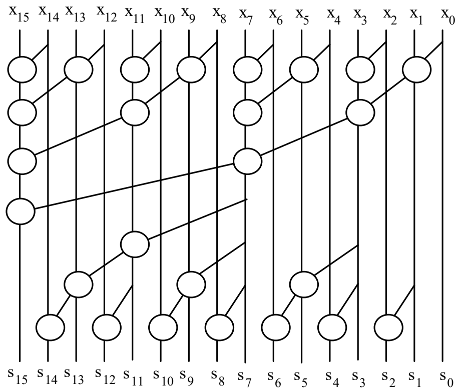
Note that even the Brent-Kung network produces the leftmost output in optimal time

We are interested in building a type-0 overall network, but can use type-*x* networks (*x* > 0) as component parts



Recursive construction of the fastest possible parallel prefix network (type-0)

Examples of Type-0, 1, 2 Parallel Prefix Networks



Brent/Kung:
16-input type-2
network

Kogge/Stone
16-input type-0
network

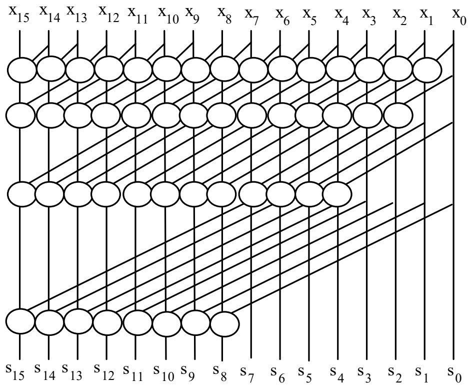
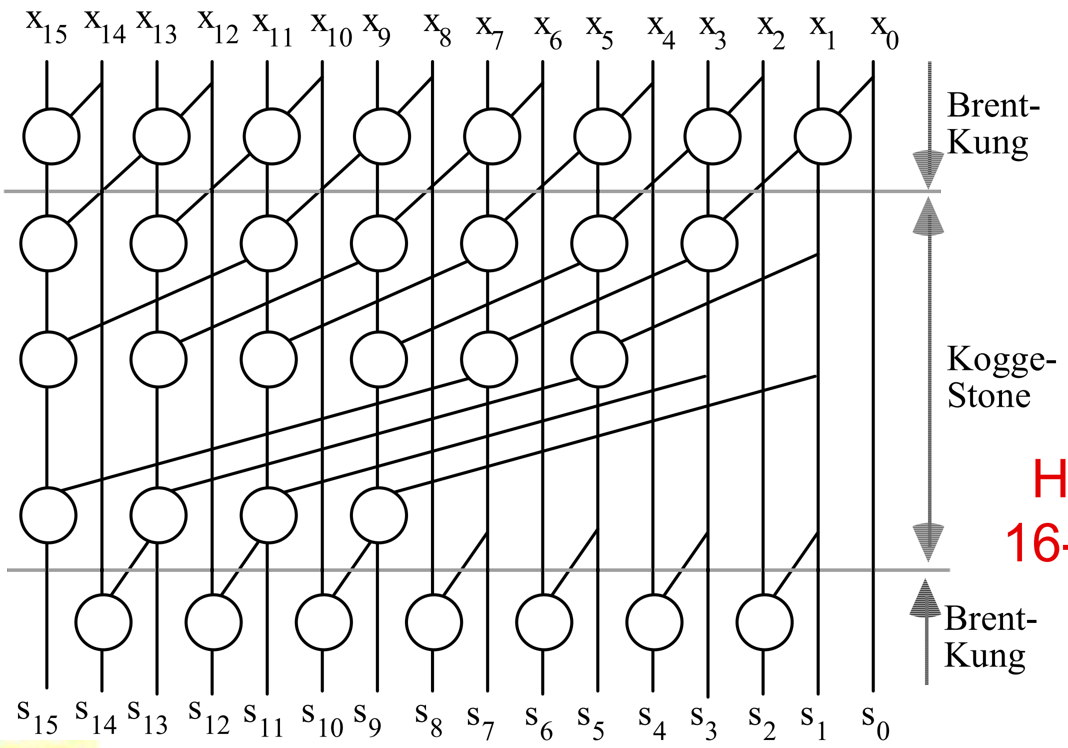


Fig. 8.10 A hybrid Brent–Kung / Kogge–Stone parallel prefix graph for *n* = 16.



Han/Carlson
16-input type-1
network

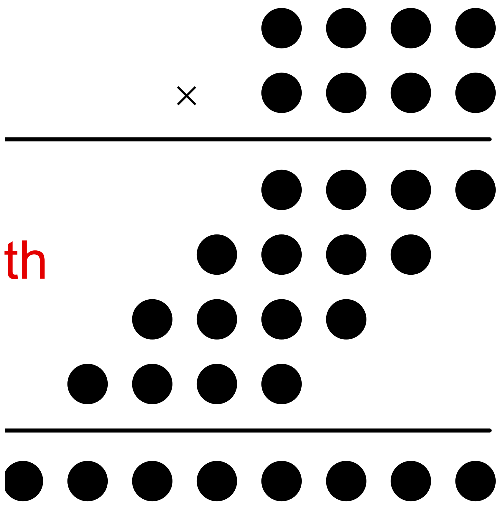
8B.5 Multiplication and Squaring Circuits

Notation for our discussion of multiplication algorithms:

a	Multiplicand	$a_{k-1}a_{k-2} \ldots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \ldots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \; . \; . \; . \; p_3p_2p_1p_0$

Initially, we assume unsigned operands

Sequential: O(k) circuit complexity O(k) time with carry-save additions



a	Multiplicand	
x	Multiplier	
$x_0 a 2^0$	Partial products bit-matrix	
$x_1 a 2^1$		
$x_2 a 2^2$		
$x_3 a 2^3$		
p	Product	

Parallel: O(k^2) circuit complexity O(log k) time

Fig. 9.1 (in *Computer Arithmetic*) Multiplication of 4-bit binary numbers.

Divide-and-Conquer (Recursive) Multipliers

Building wide multiplier from narrower ones

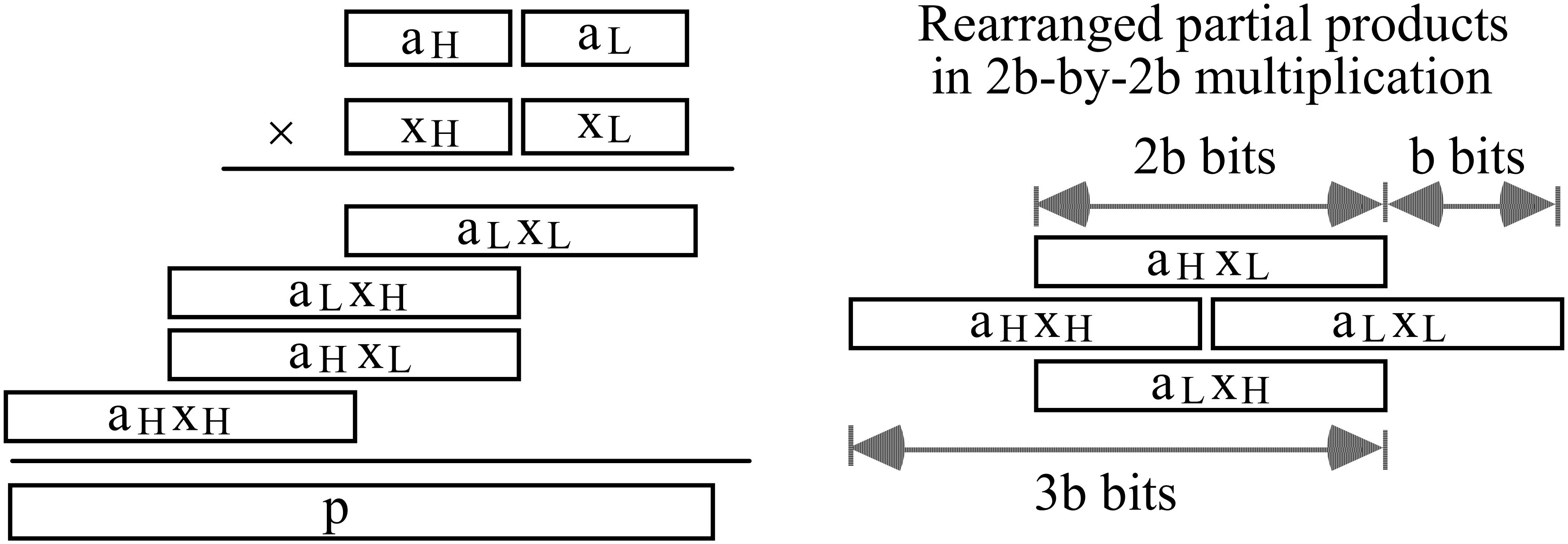


Fig. 12.1 (in *Computer Arithmetic*) Divide-and-conquer (recursive) strategy for synthesizing a $2b \times 2b$ multiplier from $b \times b$ multipliers.

$C(k) = 4C(k/2) + O(k) = O(k^2)$

$T(k) = T(k/2) + O(\log k) = O(\log^2 k)$

(Anatoly) Karatsuba Multiplication

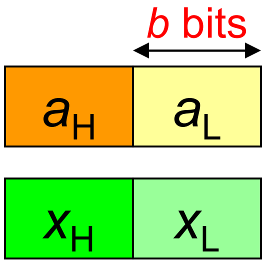
$2b \times 2b$ multiplication requires four $b \times b$ multiplications:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) = 2^{2b} a_H x_H + 2^b (a_H x_L + a_L x_H) + a_L x_L$$

Karatsuba noted that one of the four multiplications can be removed at the expense of introducing a few additions:

$$(2^b a_H + a_L) \times (2^b x_H + x_L) =$$

$$2^{2b} \underbrace{a_H x_H}_{\text{Mult 1}} + 2^b [\underbrace{(a_H + a_L) \times (x_H + x_L)}_{\text{Mult 3}} - a_H x_H - a_L x_L] + \underbrace{a_L x_L}_{\text{Mult 2}}$$

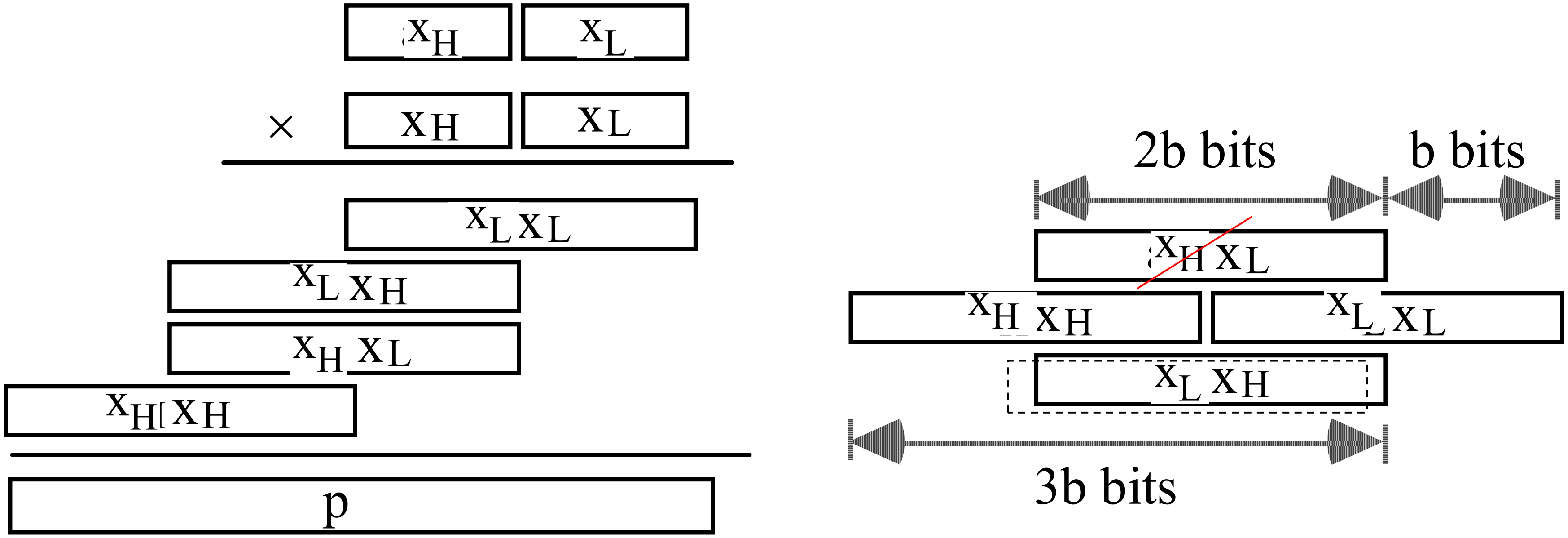


Benefit is quite significant for extremely wide operands

$$C(k) = 3C(k/2) + O(k) = O(k^{1.585})$$
$$T(k) = T(k/2) + O(\log k) = O(\log^2 k)$$

Divide-and-Conquer Squarers

Building wide squarers from narrower ones



Divide-and-conquer (recursive) strategy for synthesizing a *2b* × *2b* squarer from *b* × *b* squarers and multiplier.

VLSI Complexity Issues and Bounds

Any VLSI circuit computing the product of two k-bit integers must satisfy the following constraints:

AT grows at least as fast as $k^{3/2}$
AT^2 is at least proportional to k^2

Array multipliers: $O(k^2)$ gate count and area, $O(k)$ time

$AT = O(k^3)$ $AT^2 = O(k^4)$

Simple recursive multipliers: $O(k^2)$ gate count, $O(\log^2 k)$ time

$AT = O(k^2 \log^2 k)$? $AT^2 = O(k^2 \log^4 k)$?

Karatsuba multipliers: $O(k^{1.585})$ gate count, $O(\log^2 k)$ time

$AT = O(k^{1.585} \log^2 k)$? $AT^2 = O(k^{1.585} \log^4 k)$???

Discrepancy due to the fact that interconnect area is not taken into account in our previous analyses

Theoretically Best Multipliers

Arnold Schonhage and Volker Strassen (via FFT); best until 2007

O(log *k*) time
O(*k* log *k* log log *k*) complexity

In 2007, Martin Furer managed to replace the log log *k* term with an asymptotically smaller term (for astronomically large numbers)

It is an open problem whether there exist logarithmic-delay multipliers with linear cost
(it is widely believed that there are not)

In the absence of a linear cost multiplication circuit, multiplication must be viewed as a more difficult problem than addition

In 2019, David Harvey and Joris van der Hoeven developed an O(*n* log *n*) multiplication algorithm, which is believed to be the best possible theoretically (but not practical at present)

8B.6 Division and Square-Rooting Circuits

Division via Newton's method: $O(\log k)$ multiplications

Using Schonhage and Strassen's FFT-based multiplication, leads to:

$O(\log^2 k)$ time
$O(k \log k \log \log k)$ complexity

With the multiplication algorithm of Harvey and van der Hoeven:
$O(\log^2 k)$ time
$O(k \log k)$ complexity

Complexity theory results: It is possible to design dividers

with	$O(\log k)$ latency	and	$O(k^4)$ cost
with	$O(\log k \log \log k)$ latency	and	$O(k^2)$ cost

These theoretical constructions have not led to practical designs

Theoretically Best Dividers

Best known bounds; cannot be achieved at the same time (yet)

O(log k) time
O(k log k) complexity

In 1966, S. A. Cook established these simultaneous bounds:

O($\log^2 k$) time
O(k log k log log k) complexity

In 1983, J. H. Reif reduced the time complexity to the current best

O(log k (log log k)^2) time

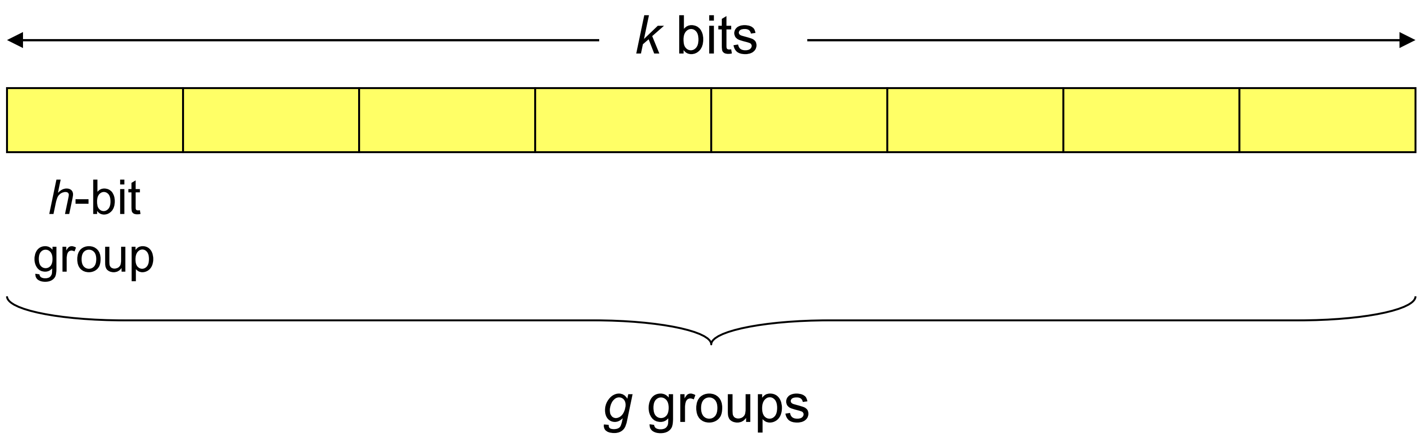
In 1984, Beame/Cook/Hoover established these simultaneous bounds:

O(log k) time
O(k^4) complexity

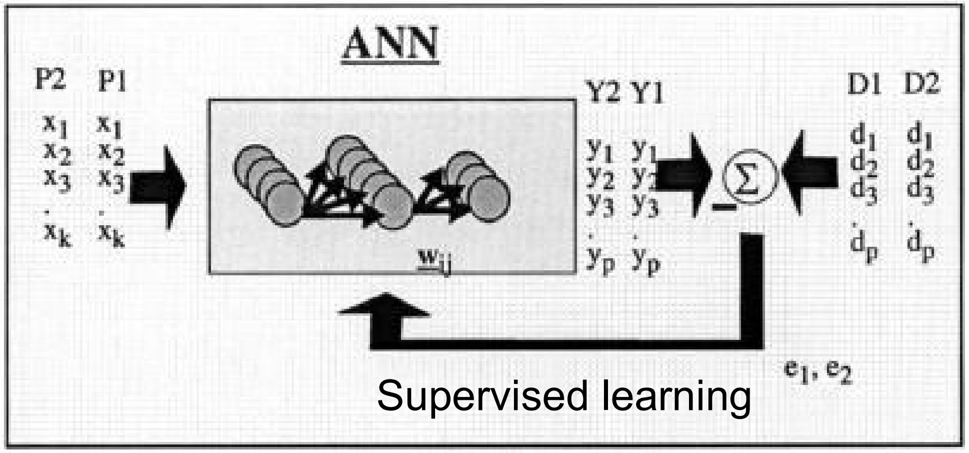
Given our current state of knowledge, division must be viewed as a more difficult problem than multiplication

Implications for Ultrawide High-Radix Arithmetic

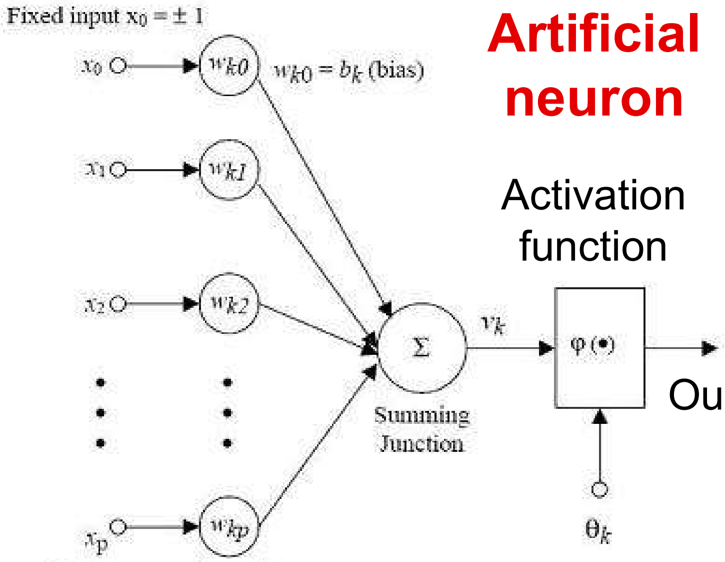
Arithmetic results with *k*-bit binary operands hold with no change when the *k* bits are processed as *g* radix-2^h digits ($gh = k$)



Another Circuit Model: Artificial Neural Nets



Supervised learning



Artificial neuron

Activation function

Output

Inputs Weights

Threshold

Characterized by connection topology and learning method

Feedforward network
Three layers: input, hidden, output
No feedback

Recurrent network
Simple version due to Elman
Feedback from hidden nodes to special nodes at the input layer

Hopfield network
All connections are bidirectional

Diagrams from
http://www.learnartificialneuralnetworks.com/

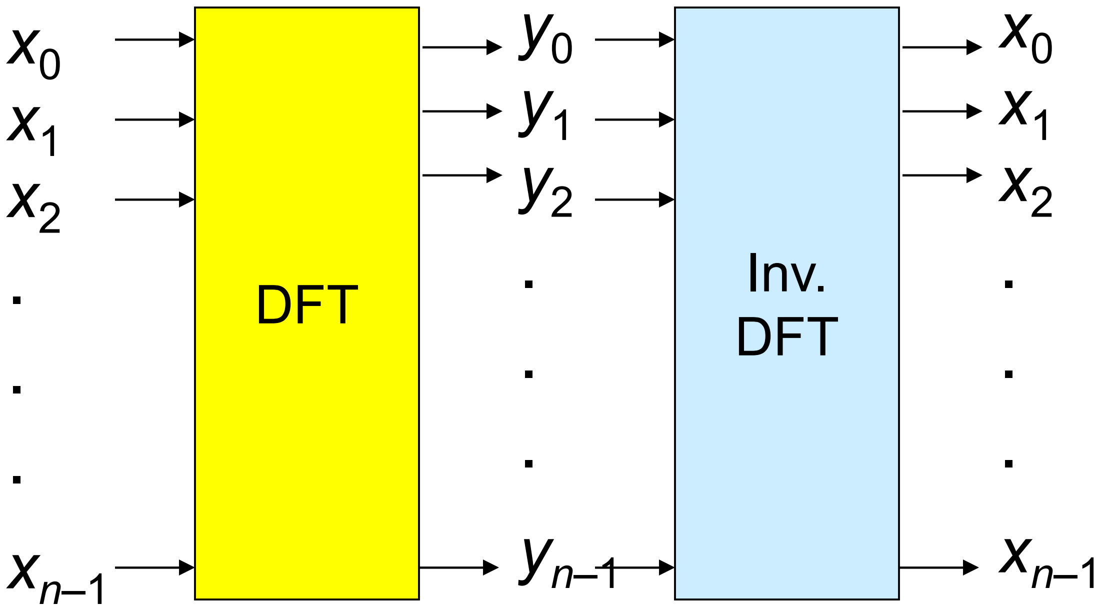
8C Fourier Transform Circuits

Fourier transform is quite important, and it also serves as a template for other types of arithmetic-intensive computations

- FFT; properties that allow efficient implementation
- General methods of mapping flow graphs to hardware

Topics in This Chapter
8C.1 The Discrete Fourier Transform
8C.2 Fast Fourier Transform (FFT)
8C.3 The Butterfly FFT Network
8C.4 Mapping of Flow Graphs to Hardware
8C.5 The Shuffle-Exchange Network
8C.6 Other Mappings of the FFT Flow Graph

8C.1 The Discrete Fourier Transform



n–point DFT

x in time domain
y in frequency domain

Some operations are easier in frequency domain; hence the need for transform

Other important transforms for discrete signals:

z-transform (generalized form of Fourier transform)

Discrete cosine transform (used in JPEG image compression)

Haar transform (a wavelet transform, which like DFT, has a fast version)

Defining the DFT and Inverse DFT

DFT yields output sequence y_i based on input sequence x_i ($0 \le i < n$)

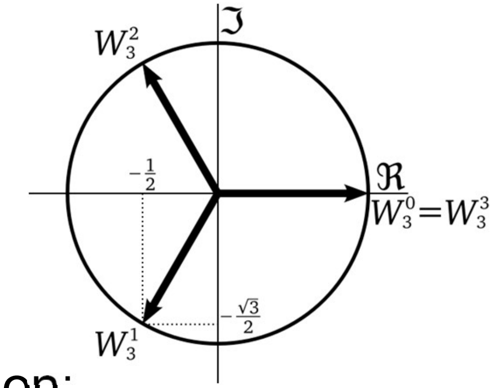
$y_i = \sum_{j=0 \text{ to } n-1} \omega_n^{ij} x_j$ O(n^2)-time naïve algorithm

where ω_n is the nth primitive root of unity; $\omega_n^n = 1$, $\omega_n^j \neq 1$ ($1 \le j < n$)

Examples: $\omega_4 = i$

$\omega_3 = (-1 + i\sqrt{3})/2$

$\omega_8 = \sqrt{2}(1 + i)/2$



The inverse DFT is almost exactly the same computation:

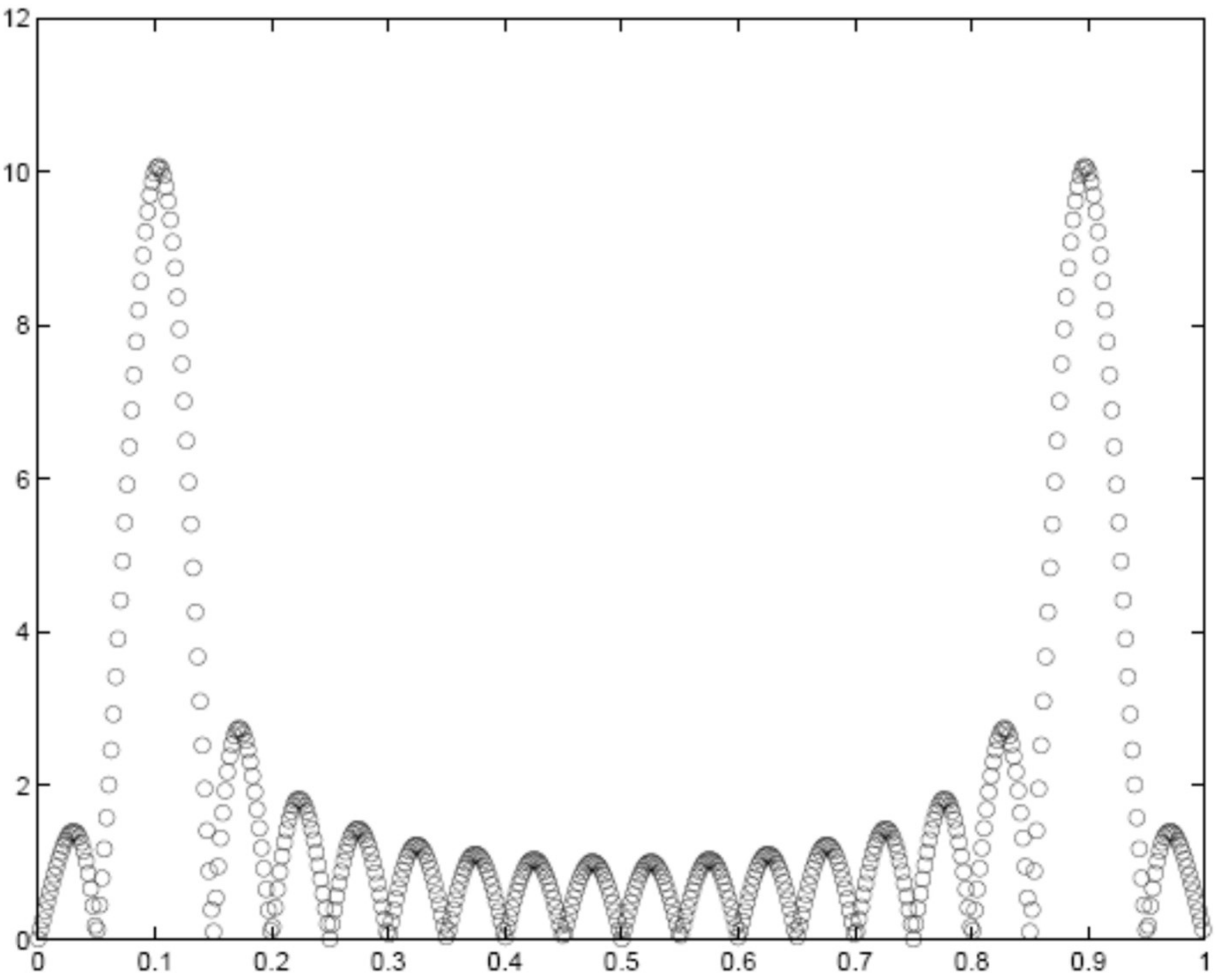
$x_i = (1/n) \sum_{j=0 \text{ to } n-1} \omega_n^{-ij} y_j$

Input seq. x_i ($0 \le i < n$) is said to be in time domain

Output seq. y_i ($0 \le i < n$) is the input's frequency-domain characterization

DFT of a Cosine Waveform

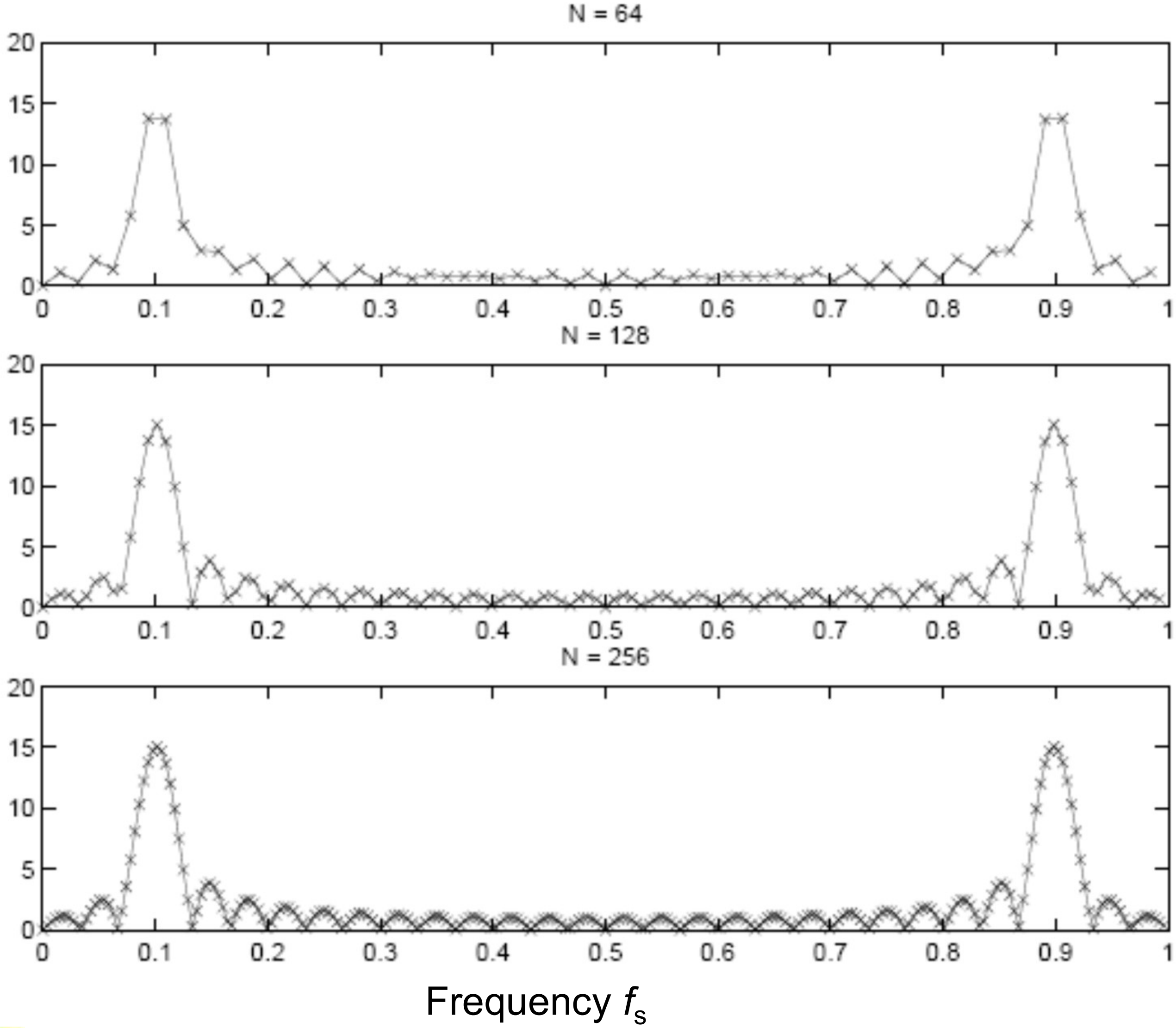
DFT of a cosine with a frequency 1/10 the sampling frequency f_s



Frequency f_s

DFT of a Cosine with Varying Resolutions

DFT of a cosine with a frequency 1/10 the sampling frequency f_s



DFT as Vector-Matrix Multiplication

DFT and inverse DFT computable via matrix-by-vector multiplication

$$y_i = \sum_{j=0}^{n-1} \omega_n^{ij} x_j$$

$$\begin{pmatrix} Y \end{pmatrix} = \begin{pmatrix} \quad W \quad \end{pmatrix} \times \begin{pmatrix} X \end{pmatrix}$$

DFT matrix

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

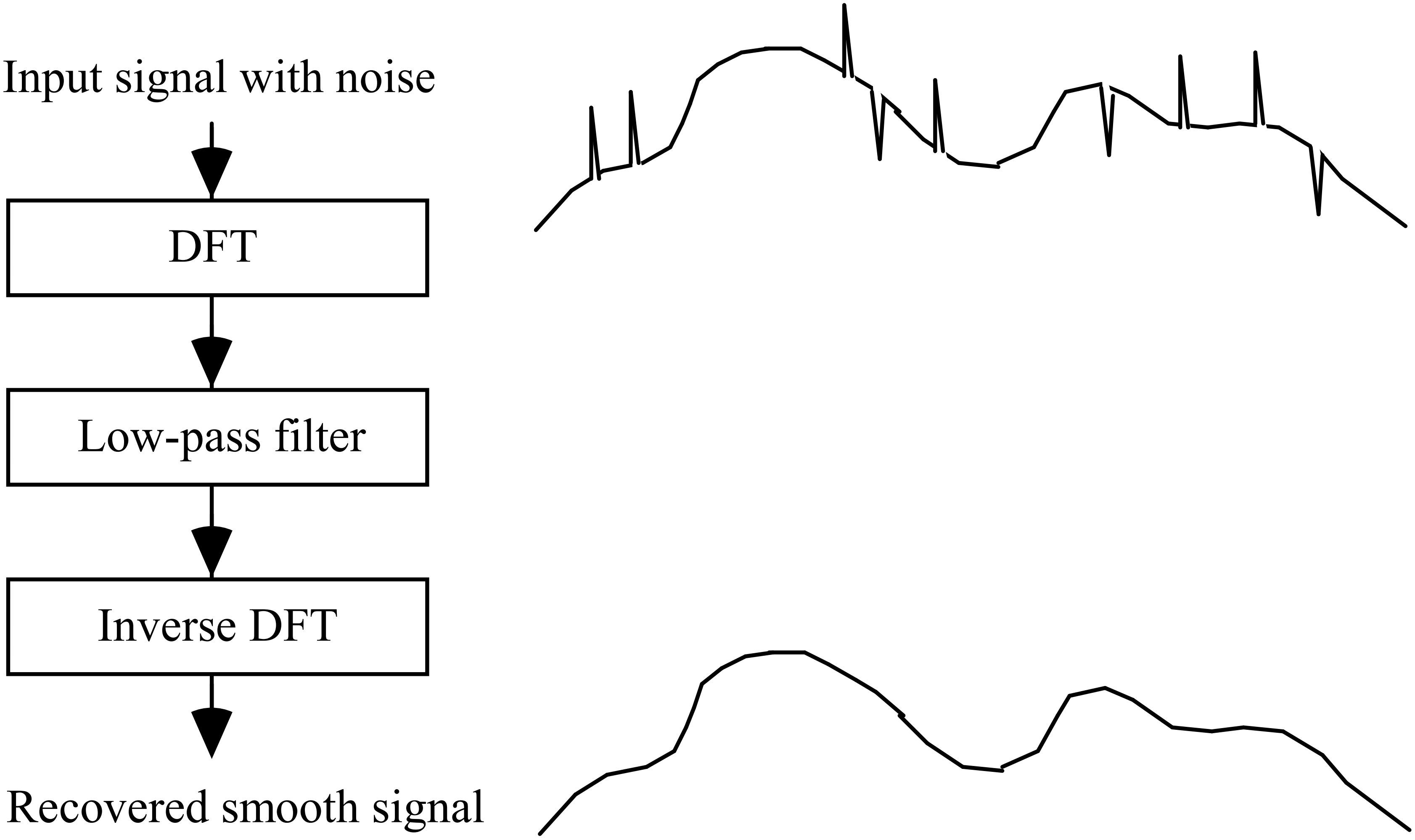
DFT Basics and Visualizations

Fourier transform, Fourier series, and frequency spectrum (16-min. video)
https://www.youtube.com/watch?v=r18Gi8lSkfM

Discrete Fourier transform: Introduction (11-minute video)
https://www.youtube.com/watch?v=mkGsMWi_j4Q

A visual introduction to Fourier transform (21-minute video)
https://www.youtube.com/watch?v=spUNpyF58BY

Application of DFT to Smoothing or Filtering



Input signal with noise

DFT

Low-pass filter

Inverse DFT

Recovered smooth signal

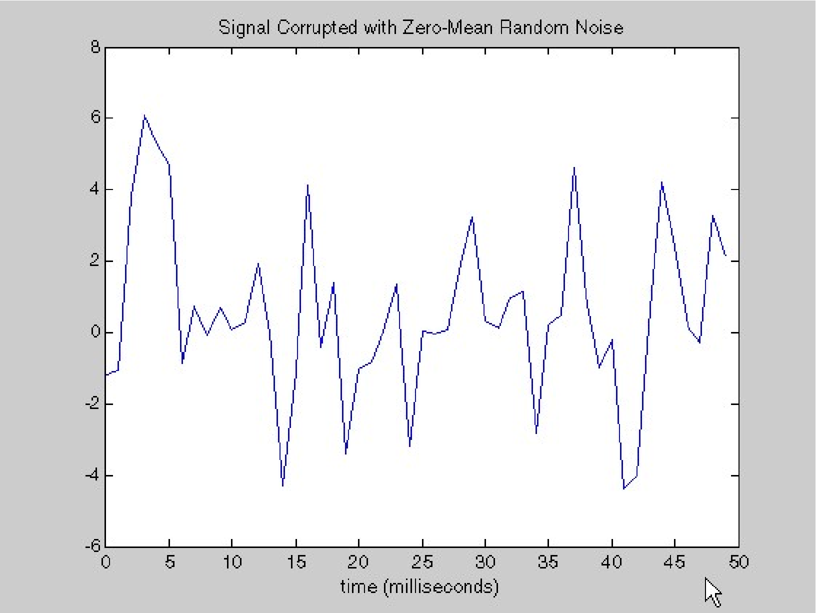
DFT Application Example

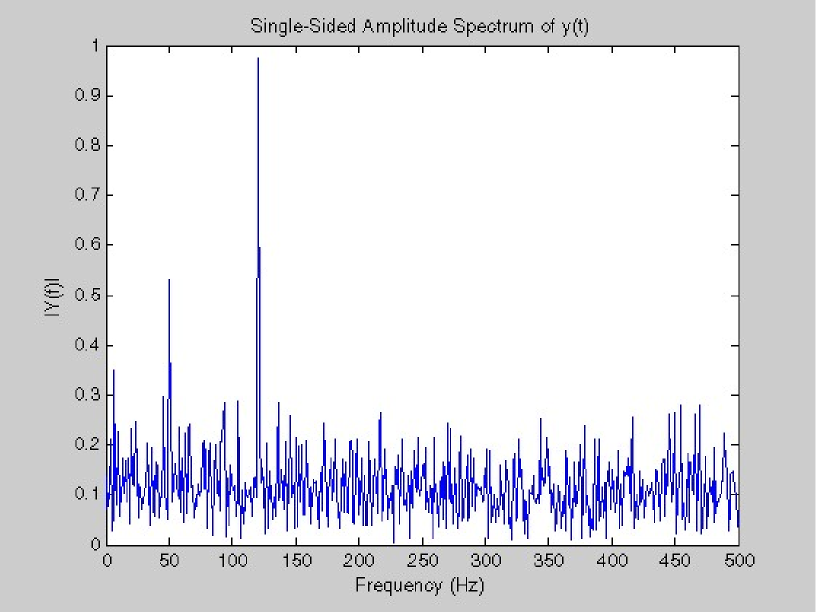
Signal corrupted by 0-mean random noise

FFT shows strong frequency components of 50 and 120

The uncorrupted signal was:

$x = 0.7 \sin(2\pi\, 50t) + \sin(2\pi\, 120t)$



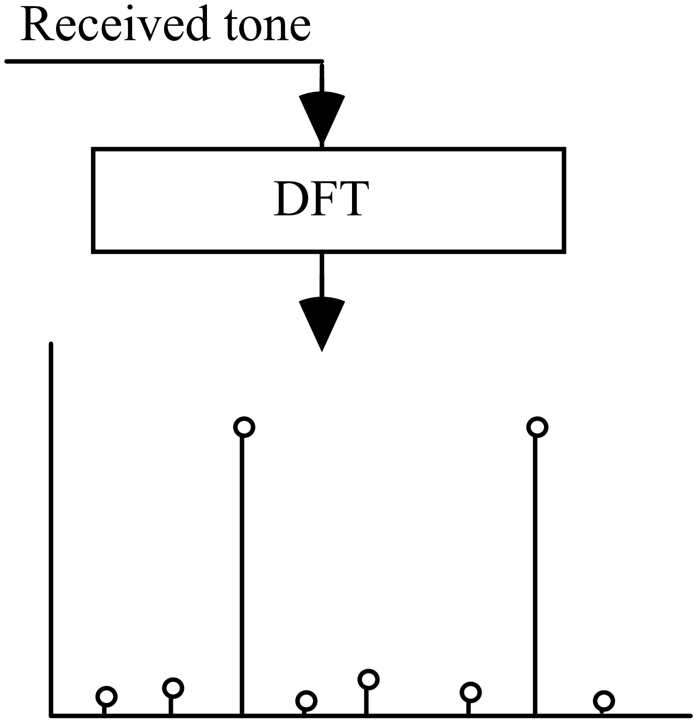


Source of images:
http://www.mathworks.com/help/techdoc/ref/fft.html

Application of DFT to Spectral Analysis

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Tone frequency assignments
for touch-tone dialing



Frequency spectrum of received tone

8C.2 Fast Fourier Transform

DFT yields output sequence y_i based on input sequence x_i $(0 \le i < n)$

$$y_i = \sum_{j=0 \text{ to } n-1} \omega_n^{ij} x_j$$

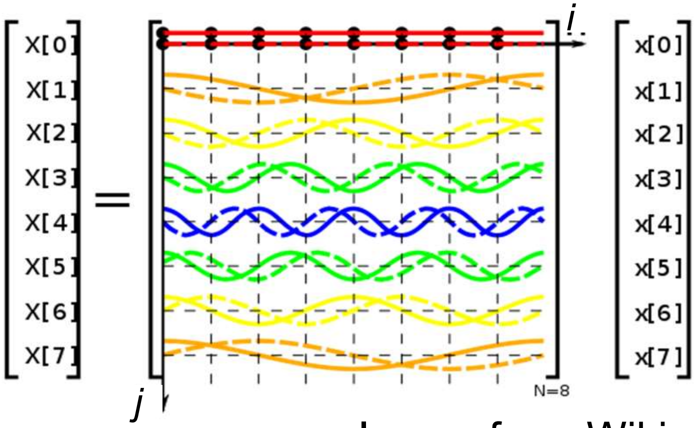


Image from Wikipedia

Fast Fourier Transform (FFT):

The Cooley-Tukey algorithm

O(n log n)-time DFT algorithm that derives y from half-length sequences u and v that are DFTs of even- and odd-indexed inputs, respectively

$$y_i = u_i + \omega_n^i v_i \qquad (0 \le i < n/2)$$

$$y_{i+n/2} = u_i + \omega_n^{i+n/2} v_i = u_i - \omega_n^i v_i$$

Butterfly operation

$T(n) = 2T(n/2) + n = n \log_2 n$ sequentially

$T(n) = T(n/2) + 1 = \log_2 n$ in parallel

More General Factoring-Based Algorithm

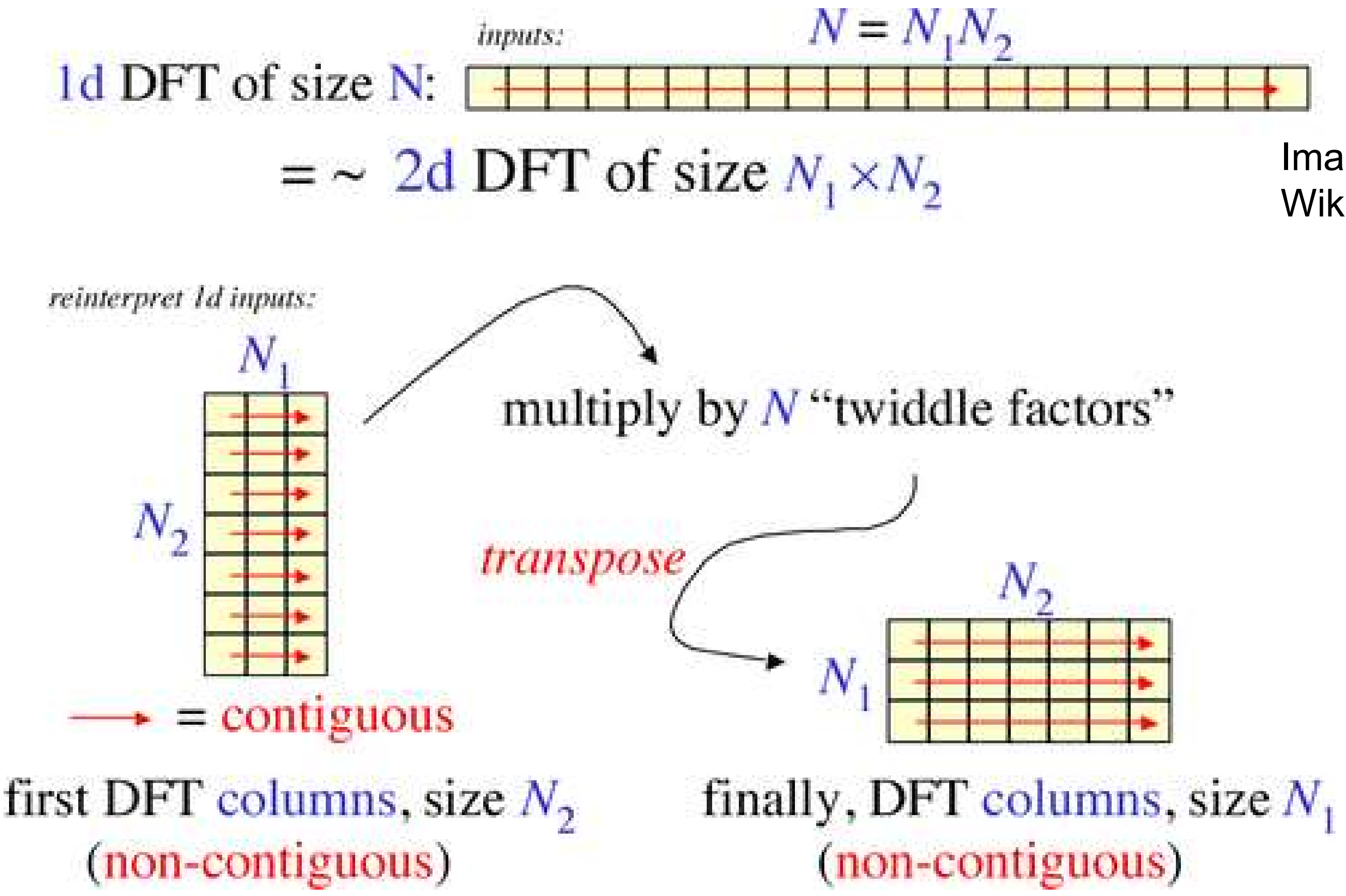

inputs:
$N = N_1 N_2$
1d DFT of size N:
$= \sim$ 2d DFT of size $N_1 \times N_2$
reinterpret 1d inputs:
N_1
N_2
multiply by N "twiddle factors"
transpose
N_2
N_1
= contiguous
first DFT columns, size N_2
(non-contiguous)
finally, DFT columns, size N_1
(non-contiguous)

Image from Wikipedia

8C.3 The Butterfly FFT Network

u: DFT of even-indexed inputs
v: DFT of odd-indexed inputs

$$y_i \;=\; u_i + \omega_n^{\,i}\, v_i \qquad (0 \le i < n/2)$$
$$y_{i+n/2} \;=\; u_i + \omega_n^{\,i+n/2}\, v_i$$

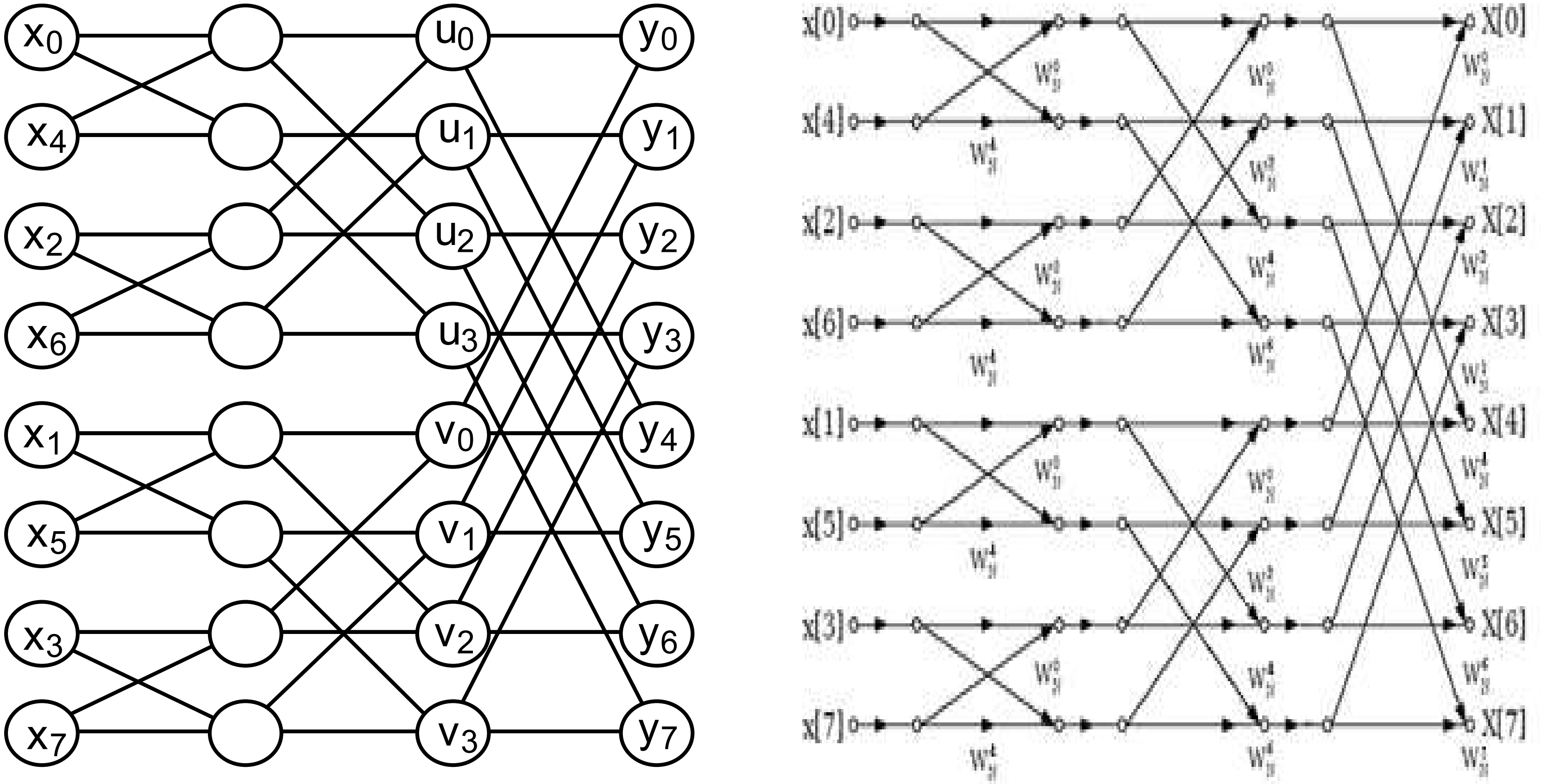
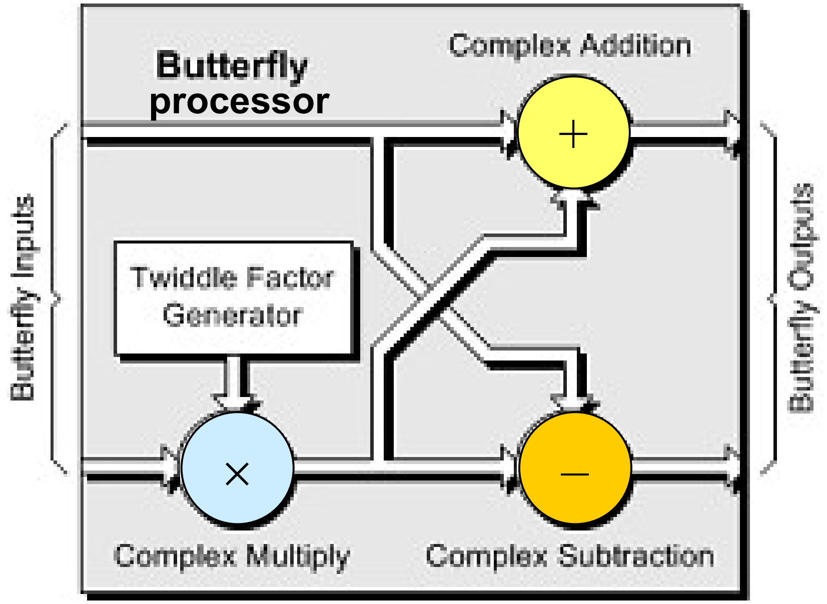


Fig. 8.11 Butterfly network for an 8-point FFT.

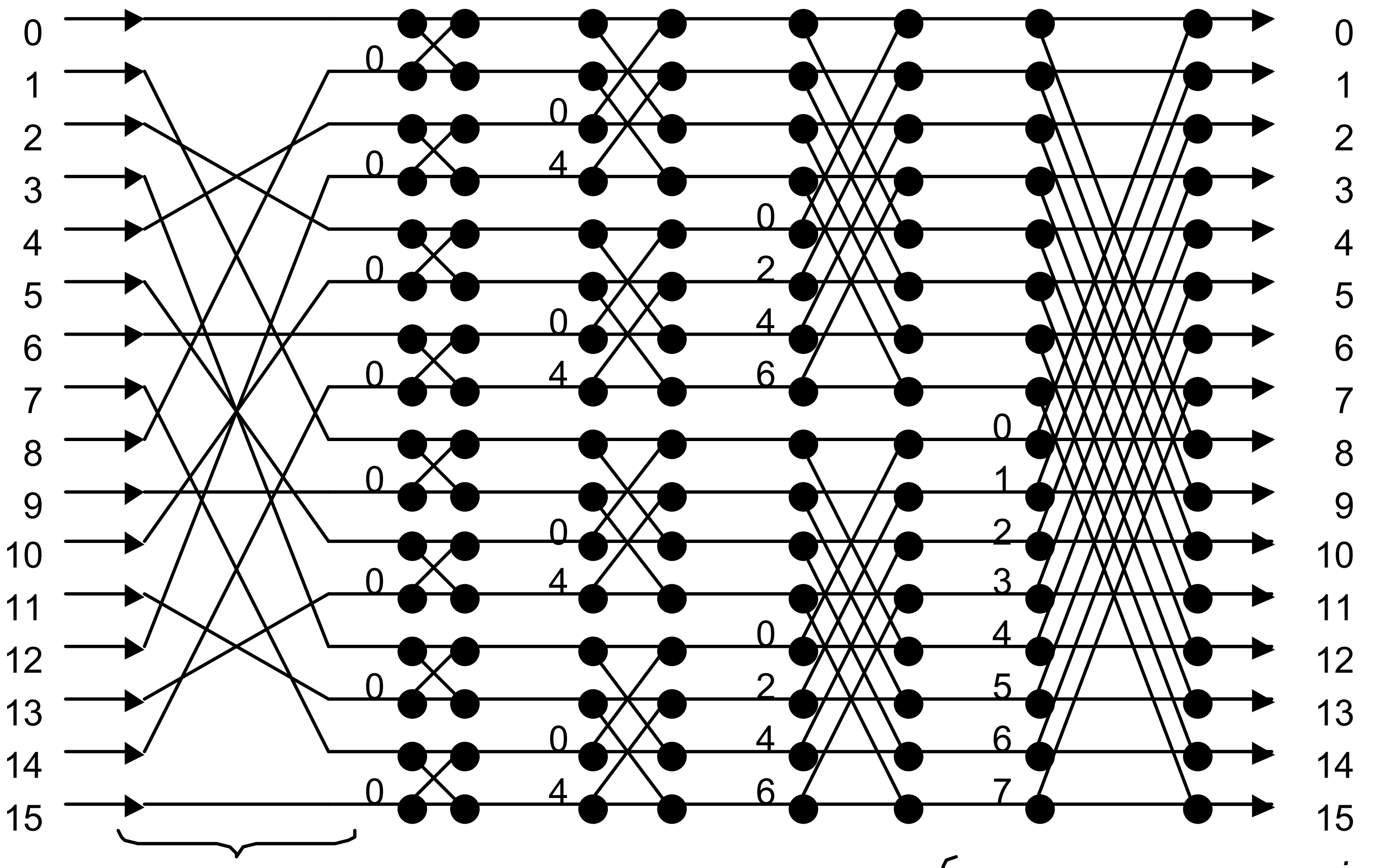
Butterfly Processor

Performs a pair of multiply-add operations, where the multiplication is by a constant



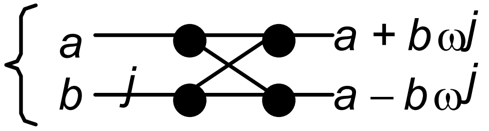
Design can be optimized by merging the adder and subtractor, as they receive the same inputs

Computation Scheme for 16-Point FFT



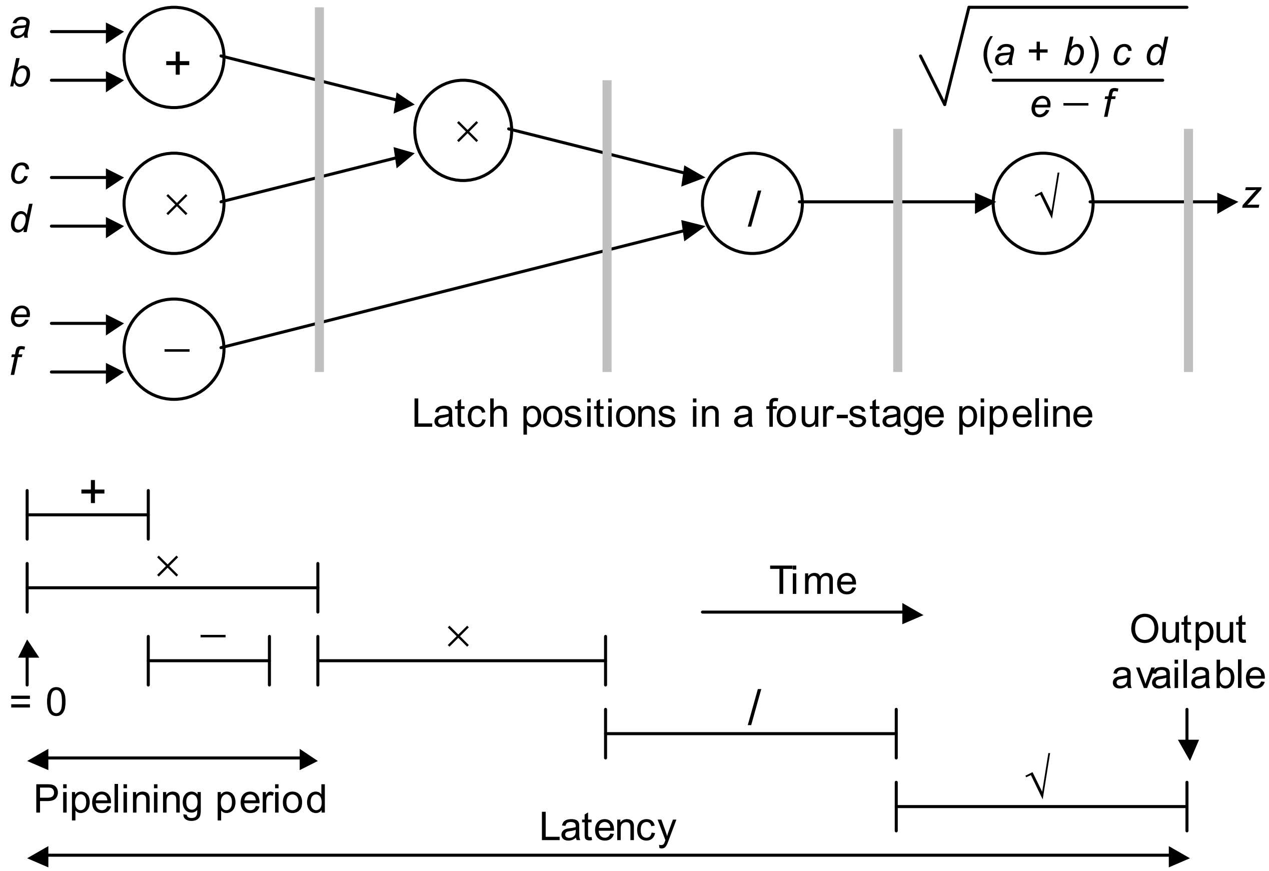
Bit-reversal permutation

Butterfly operation



8C.4 Mapping of Flow Graphs to Hardware

Given a computation flow graph, it can be mapped to hardware

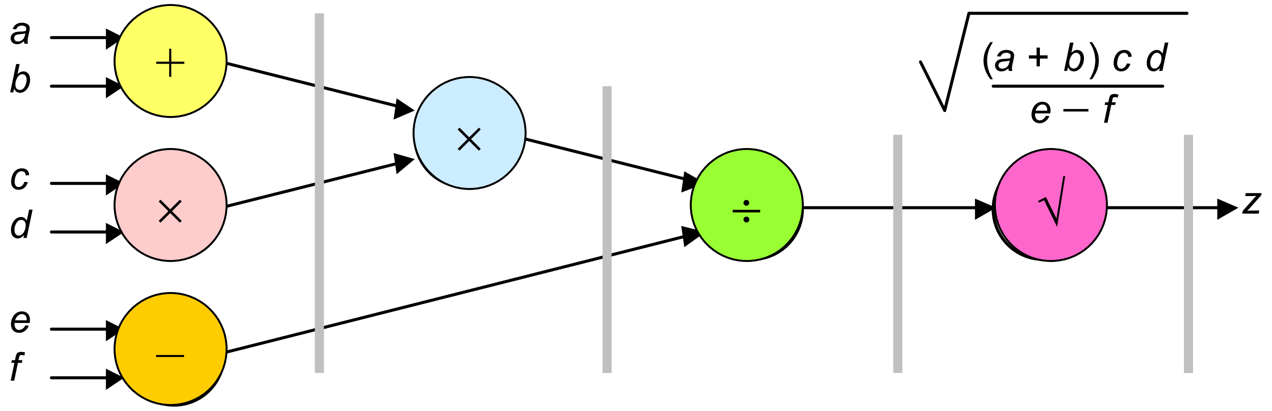

a
b
c
d
e
f
+
×
−
×
/
√
z
$\sqrt{\frac{(a+b)\,c\,d}{e-f}}$
Latch positions in a four-stage pipeline
+
×
−
×
/
√
Time
t = 0
Output available
Pipelining period
Latency

Direct one-to-one mapping (possibly with pipelining)

Fig. 25.6 Parhami's textbook on computer arithmetic.

Ad-hoc Scheduling on a Given Set of Resources

Given a computation flow graph, it can be mapped to hardware



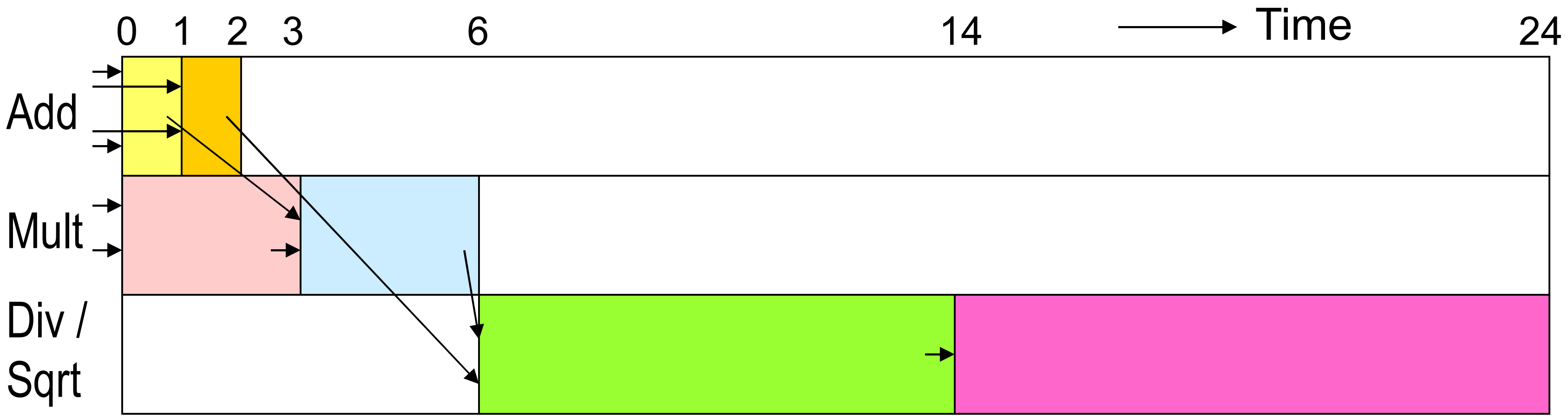
Assume:

$t_{\text{add}} = 1$

$t_{\text{mult}} = 3$

$t_{\text{div}} = 8$

$t_{\text{sqrt}} = 10$

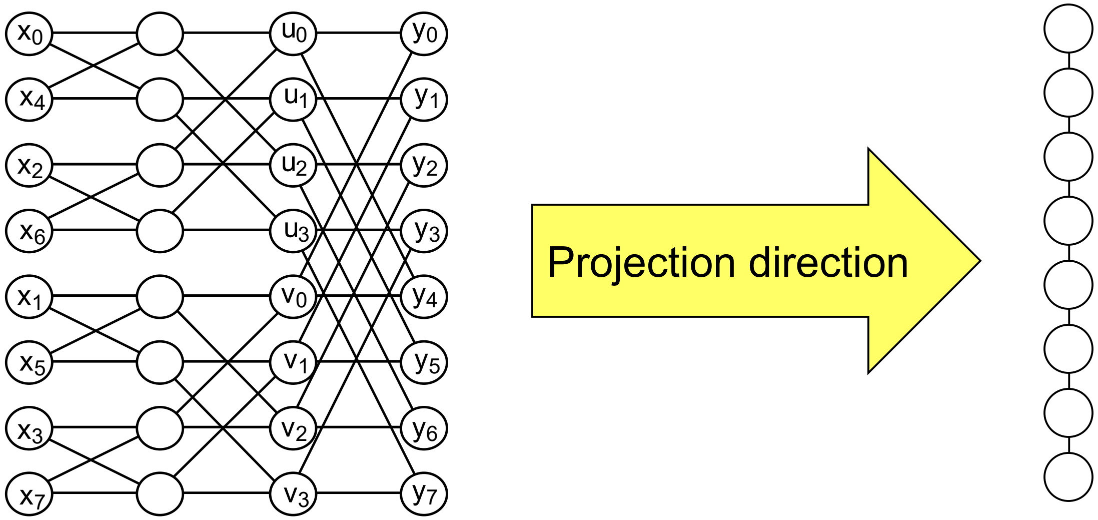


Mapping through Projection

Given a flow graph, it can be projected in various directions to obtain corresponding hardware realizations

Multiple nodes of a flow graph may map onto a single hardware node

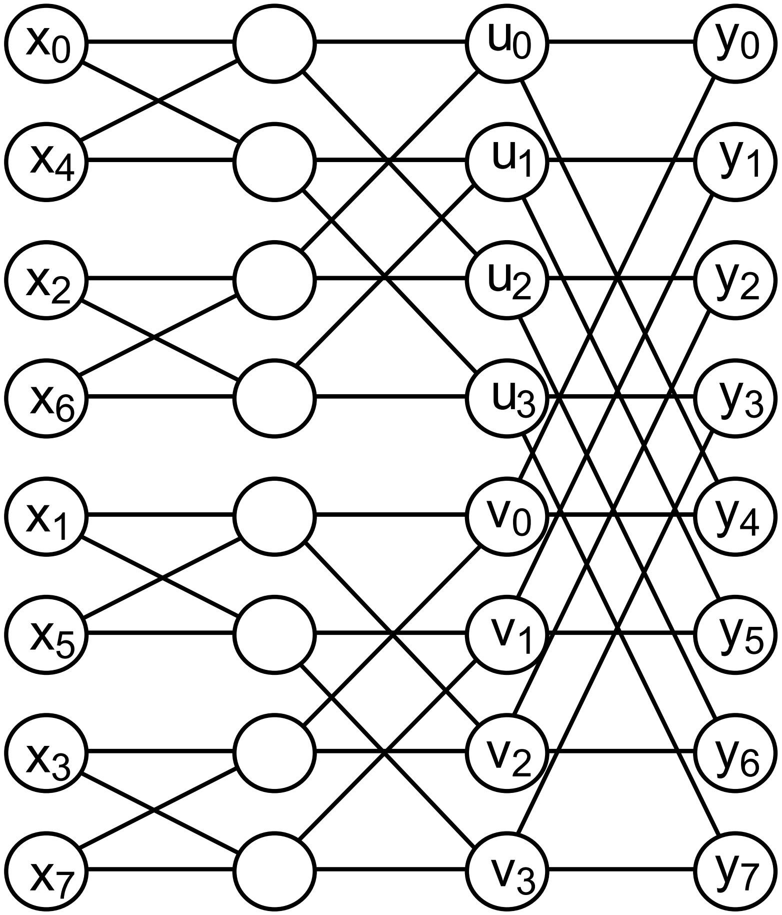
That one hardware node then performs the computations associated with the flow graph nodes one by one, according to some timing arrangement (schedule)

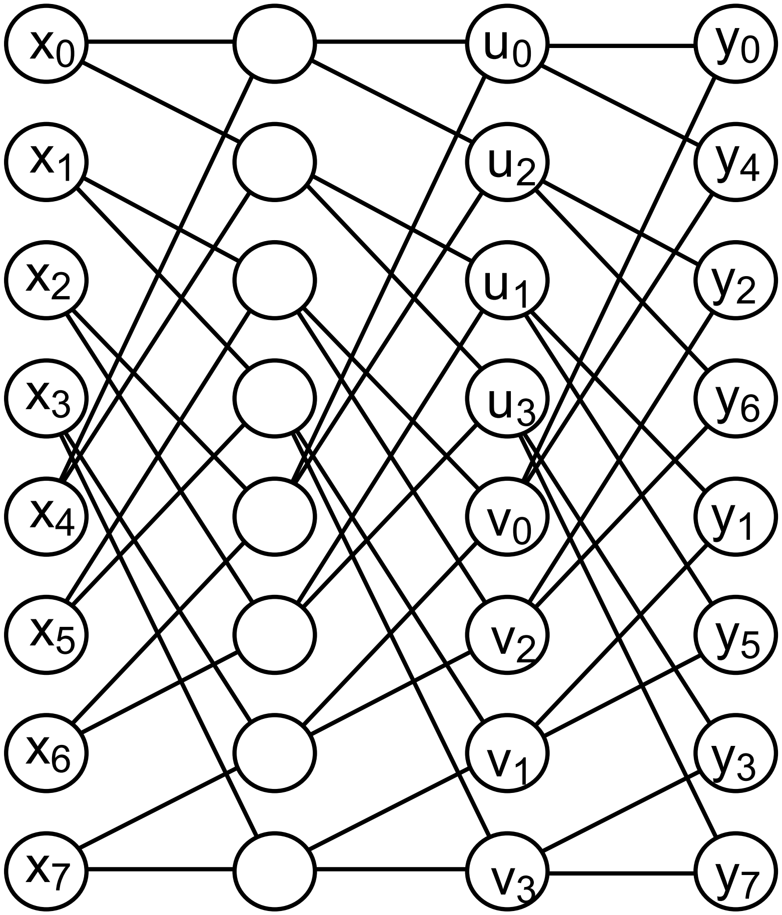


Projection direction

Linear array, with each cell acting for one butterfly network row

8C.5 The Shuffle-Exchange Network





Variants of the Butterfly Architecture

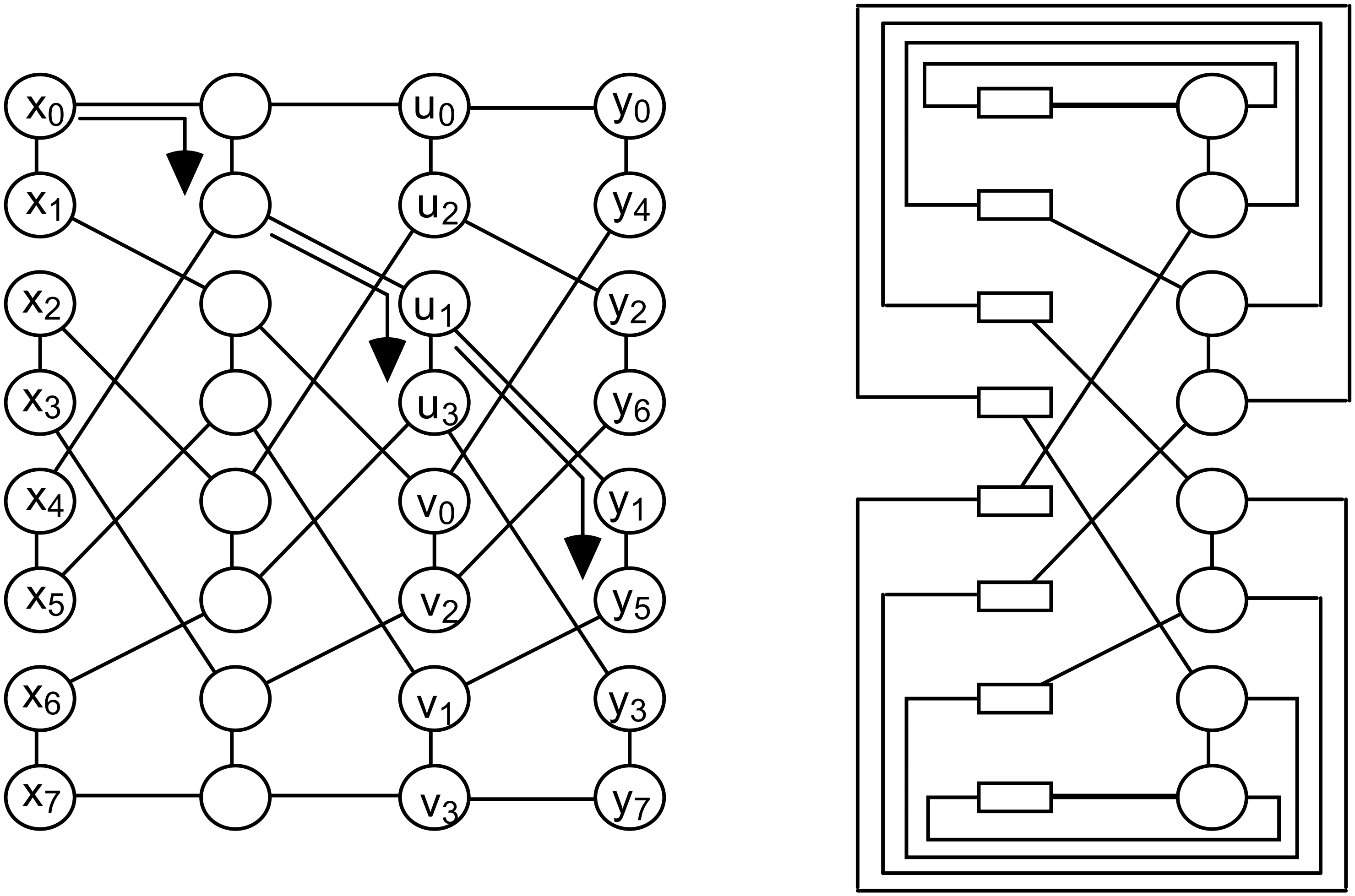


Fig. 8.12 FFT network variant and its shared-hardware realization.

8C.6 Other Mappings of the FFT Flow Graph

This section is incomplete at this time

More Economical FFT Hardware

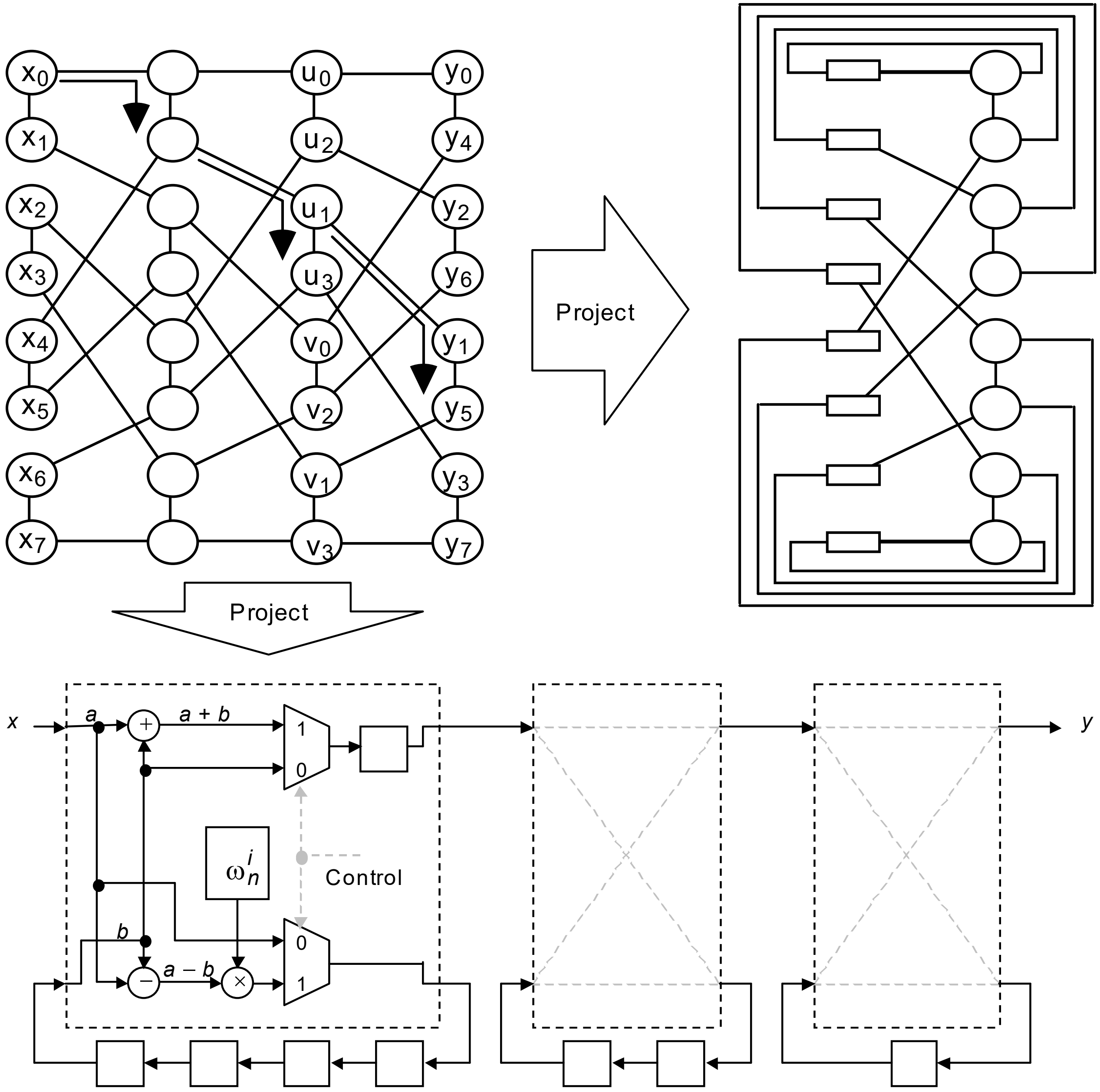
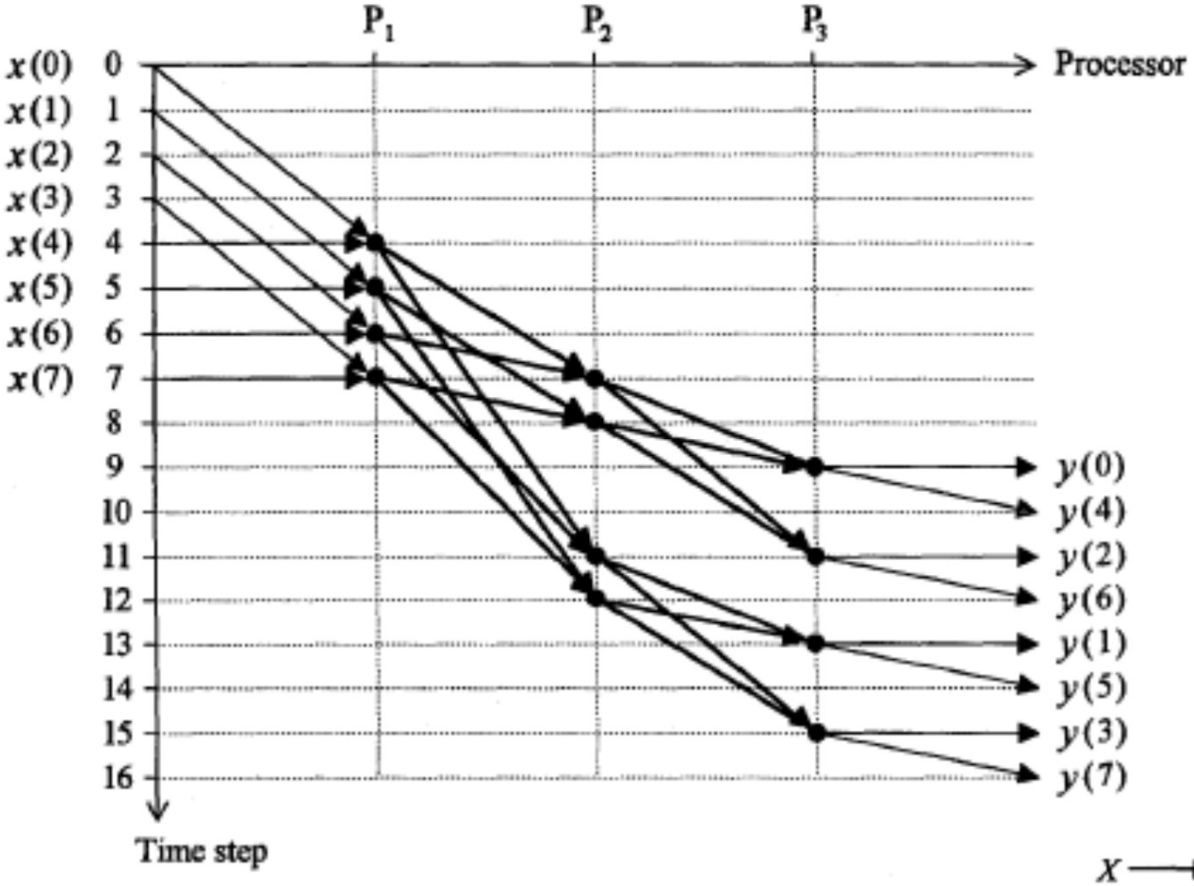
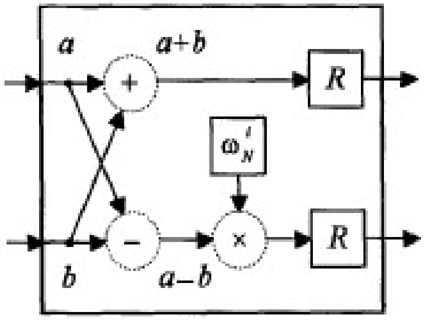


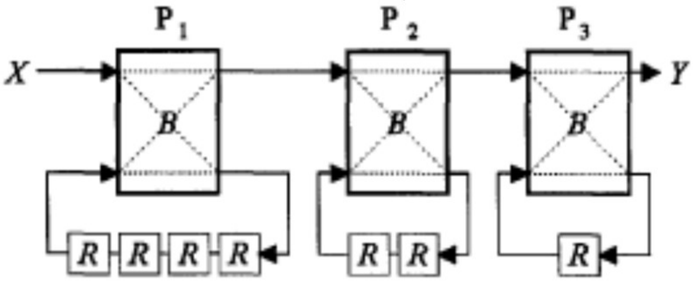
Fig. 8.13 Linear array of $\log_2 n$ cells for *n*-point FFT computation.

Space-Time Diagram for the Feedback FFT Array





Feedback butterfly processor


P₁
P₂
P₃
X
Y
B
B
B
R R R R
R R
R