VLSI Digital Signal Processing Systems

Keshab K. Parhi

VLSI Digital Signal Processing Systems

- Textbook:
  - K.K. Parhi, VLSI Digital Signal Processing Systems: Design and Implementation, John Wiley, 1999

- Buy Textbook:
  - http://www.bn.com
  - http://www.amazon.com
  - http://www.bestbookbuys.com

Chapter 1. Introduction to DSP Systems

- Introduction (Read Sec. 1.1, 1.3)
- Non-Terminating Programs Require Real-Time Operations
- Applications dictate different speed constraints (e.g., voice, audio, cable modem, settop box, Gigabit ethernet, 3-D Graphics)
- Need to design Families of Architectures for specified algorithm complexity and speed constraints
- Representations of DSP Algorithms (Sec. 1.4)

Typical DSP Programs

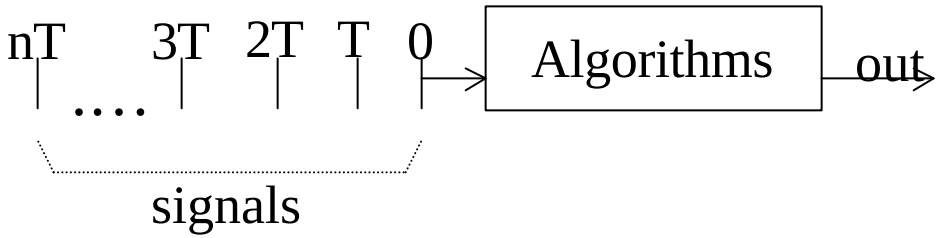
- Usually highly real-time, design hardware and/or software to meet the application speed constraint

- Non-terminating
  - Example:

for $n = 1$ *to* ∞

$$y(n) = a \cdot x(n) + b \cdot x(n-1) + c \cdot x(n-2)$$

end



Area-Speed-Power Tradeoffs

- 3-Dimensional Optimization (Area, Speed, Power)
- Achieve Required Speed, Area-Power Tradeoffs
- Power Consumption

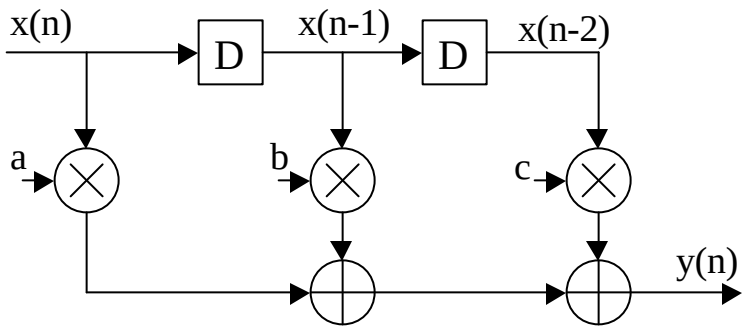
$$P = C \cdot V^2 \cdot f$$

- Latency reduction Techniques => Increase in speed or power reduction through lower supply voltage operation
- Since the capacitance of the multiplier is usually dominant, reduction of the number of multiplications is important (this is possible through strength reduction)

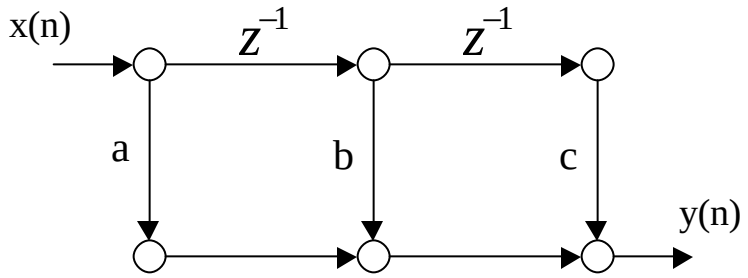
Representation Methods of DSP systems

Example: y(n)=a*x(n)+b*x(n-1)+c*x(n-2)

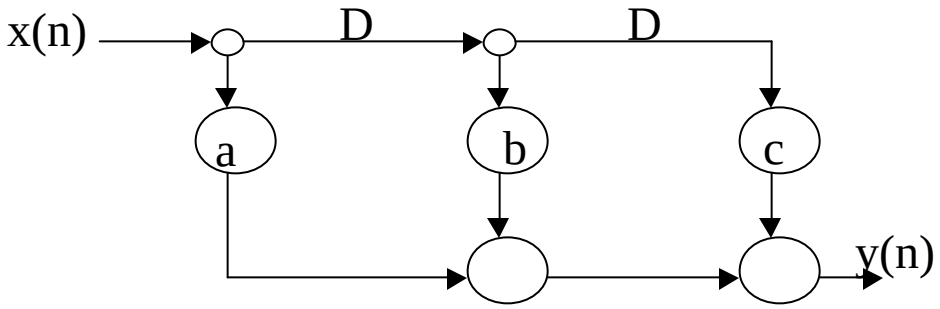
- Graphical Representation Method 1: Block Diagram
  - Consists of functional blocks connected with directed edges, which represent data flow from its input block to its output block



- Graphical Representation Method 2: Signal-Flow Graph
  - SFG: a collection of nodes and directed edges
  - Nodes: represent computations and/or task, sum all incoming signals
  - Directed edge (j, k): denotes a linear transformation from the input signal at node j to the output signal at node k
  - Linear SFGs can be transformed into different forms without changing the system functions. For example, ***Flow graph reversal*** or ***transposition*** is one of these transformations (Note: only applicable to single-input-single-output systems)
  - Usually used for linear time-invariant DSP systems representation



- Graphical Representation Method 3: Data-Flow Graph
  - DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.
  - DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.
  - Each edge describes a precedence constraint between two nodes in DFG:
    - Intra-iteration precedence constraint: if the edge has zero delays
    - Inter-iteration precedence constraint: if the edge has one or more delays
    - DFGs and Block Diagrams can be used to describe both linear single-rate and nonlinear **multi-rate** DSP systems
    - Fine-Grain DFG

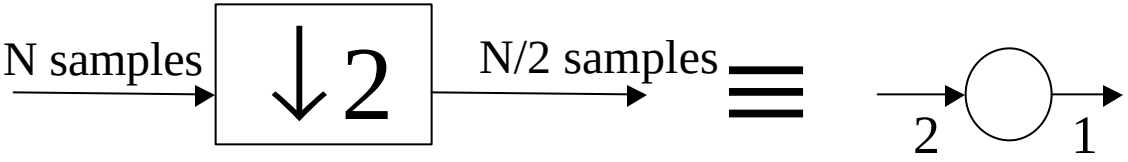


Examples of DFG

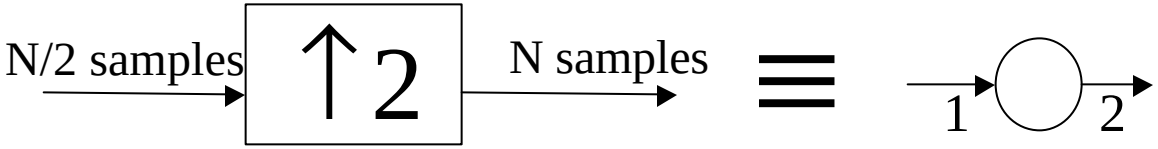
– Nodes are complex blocks (in Coarse-Grain DFGs)

– Nodes can describe expanders/decimators in Multi-Rate DFGs

Decimator



Expander

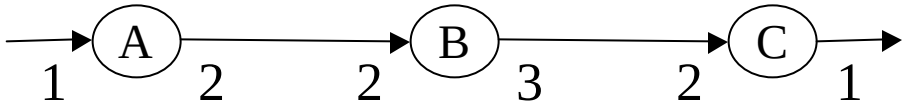


Chapter 2: Iteration Bound

- Introduction
- Loop Bound
  - Important Definitions and Examples
- Iteration Bound
  - Important Definitions and Examples
  - Techniques to Compute Iteration Bound

Introduction

- Iteration: execution of all computations (or functions) in an algorithm once
  - Example 1:



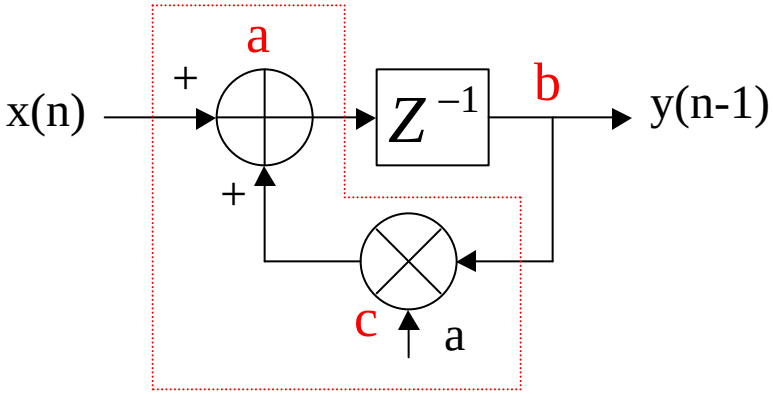
    - For 1 iteration, computations are:

A	B	C
2 times	2 times	3 times

- Iteration period: the time required for execution of one iteration of algorithm (same as sample period)
  - Example:

$$y(n) = a \cdot y(n-1) + x(n)$$

$$i.e. \qquad H(z) = \frac{1}{1 - a \cdot z^{-1}}$$



Introduction (cont'd)

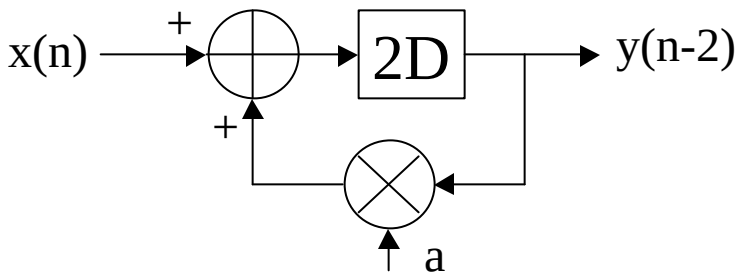
- Assume the execution times of multiplier and adder are T_m & T_a, then the iteration period for this example is $T_m + T_a$ (assume 10ns, see the red-color box). so for the signal, the sample period (T_s) must satisfy:

$$T_s \geq T_m + T_a$$

- Definitions:
  - Iteration rate: the number of iterations executed per second
  - Sample rate: the number of samples processed in the DSP system per second (also called throughput)

Iteration Bound

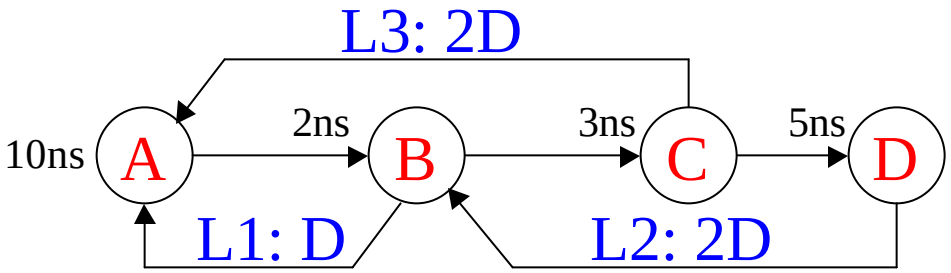
- Definitions:
  - Loop: a directed path that begins and ends at the same node
  - Loop bound of the j-th loop: defined as Tj/Wj, where Tj is the loop computation time & Wj is the number of delays in the loop
  - **Example 1:** a→ b→ c→ a is a loop (see the same example in Note 2, PP2), its loop bound: $T_{loopbound} = T_m + T_a = 10ns$
  - **Example 2:** y(n) = a*y(n-2) + x(n), we have:



$$T_{loopbound} = \frac{T_m + T_a}{2} = 5ns$$

Iteration Bound (cont'd)

– **Example 3:** compute the loop_bounds of the following loops:



$$T_{L1} = (10+2)/1 = 12ns$$
$$T_{L2} = (2+3+5)/2 = 5ns$$
$$T_{L3} = (10+2+3)/2 = 7.5ns$$

- Definitions (Important):
  - Critical Loop: the loop with the maximum loop bound
  - Iteration bound of a DSP program: the loop bound of the critical loop, it is defined as

$$T_\infty = \max_{j \in L} \left\{ \frac{T_j}{W_j} \right\}$$

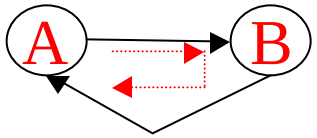
where L is the set of loops in the DSP system,
T_j is the computation time of the loop j and
W_j is the number of delays in the loop j

  - **Example 4:** compute the iteration bound of the example 3:

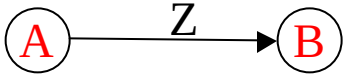
$$T_\infty = \max_{l \in L} \{12, \quad 5, \quad 7.5\}$$

Iteration bound (cont'd)

- If no delay element in the loop, then $T_\infty = T_L / 0 = \infty$
  - Delay-free loops are non-computable, see the example:



- Non-causal systems cannot be implemented

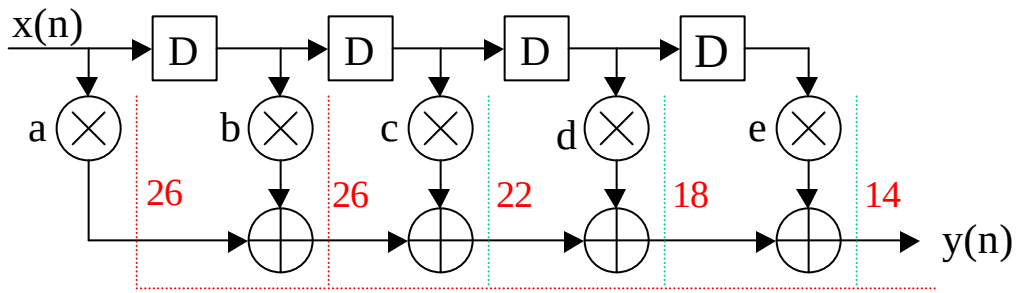


$$\begin{cases} B = A \cdot Z & non-causal \\ A = B \cdot Z^{-1} & causal \end{cases}$$

- Speed of the DSP system: depends on the "critical path comp. time"
  - Paths: do not contain delay elements (4 possible path locations)
    - (1) input node →delay element
    - (2) delay element's output → output node
    - (3) input node → output node
    - (4) delay element → delay element
  - Critical path of a DFG: the path with the longest computation time among all paths that contain zero delays
  - Clock period is lower bounded by the critical path computation time

Iteration Bound (cont'd)

- **Example:** Assume Tm = 10ns, Ta = 4ns, then the length of the critical path is 26ns (see the red lines in the following figure)

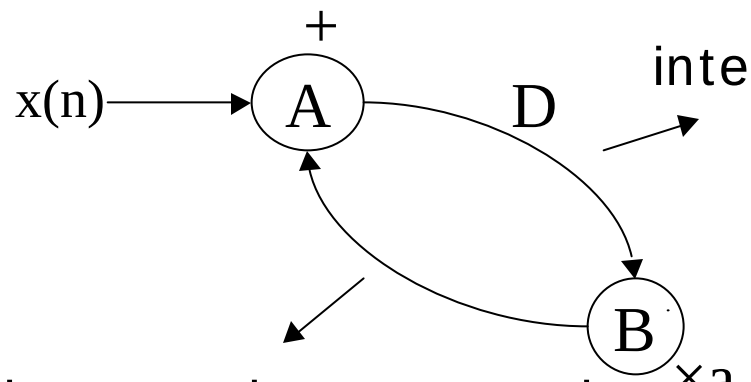


- Critical path: the lower bound on clock period
- To achieve high-speed, the length of the critical path can be reduced by *pipelining and parallel processing (Chapter 3).*

Precedence Constraints

- Each edge of DFG defines a precedence constraint
- Precedence Constraints:
  - Intra-iteration $\Rightarrow$ edges with no delay elements
  - Inter-iteration $\Rightarrow$ edges with non-zero delay elements
- Acyclic Precedence Graph(APG) : Graph obtained by deleting all edges with delay elements.

$y(n)=ay(n-1) + x(n)$



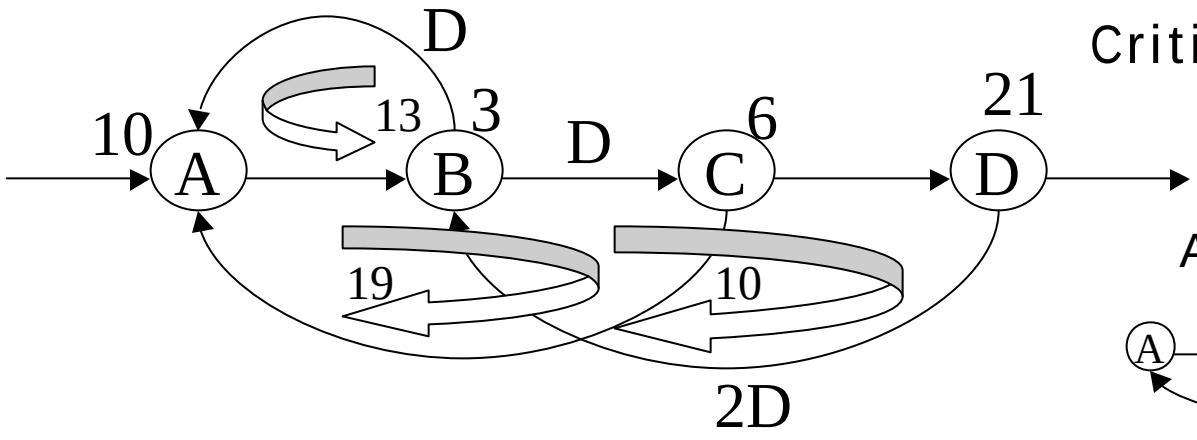
inter-iteration precedence constraint

$A_1 \rightarrow B_2$

$A_2 \rightarrow B_3$

intra-iteration precedence constraint

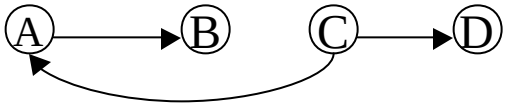
$B_1 \rightarrow A_1 \Rightarrow B_2 \rightarrow A_2 \Rightarrow B_3 \rightarrow A_3 \Rightarrow \ldots\ldots$



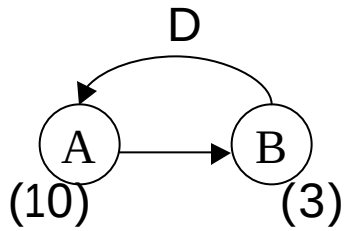
Critical Path = 27ut

$T_{clk} >= 27ut$

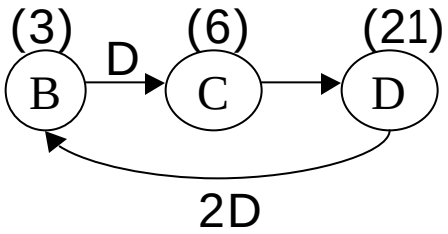
APG of this graph is



- Achieving Loop Bound



T_{loop}= 13ut $A_1 \rightarrow B_1 \Rightarrow A_2 \rightarrow B_2 \Rightarrow A_3 \ldots$



$B_1 \Rightarrow C_2 \rightarrow D_2 \Rightarrow B_4 \Rightarrow C_5 \rightarrow D_5 \Rightarrow B_7$
$B_2 \Rightarrow C_3 \rightarrow D_3 \Rightarrow B_5 \Rightarrow C_6 \rightarrow D_6 \Rightarrow B_8$
$C_1 \rightarrow D_1 \Rightarrow B_3 \Rightarrow C_4 \rightarrow D_4 \Rightarrow B_6$

Loop contains three delay elements

loop bound = 30 / 3 =10ut = (loop computation time) / (#of delay elements)

- Algorithms to compute iteration bound
  - Longest Path Matrix (LPM)
  - Minimum Cycle Mean (MCM)

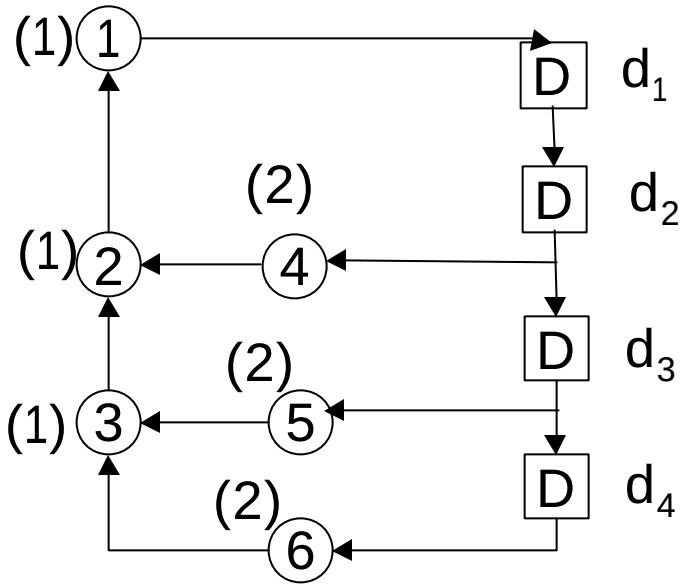
- Longest Path Matrix Algorithm
  - Let 'd' be the number of delays in the DFG.
  - A series of matrices $L^{(m)}$, m = 1, 2, … , d, are constructed such that $l_{i,j}^{(m)}$ is the longest computation time of all paths from delay element d_i to d_j that passes through exactly (m-1) delays. If such a path does not exist $l_{i,j}^{(m)} = -1$.
  - The longest path between any two nodes can be computed using either Bellman-Ford algorithm or Floyd-Warshall algorithm (Appendix A).
  - Usually, $L^{(1)}$ is computed using the DFG. The higher order matrices are computed recursively as follows :

$$l_{i,j}^{(m+1)} = \max(-1,\ l_{i,k}^{(1)} + l_{k,j}^{(m)}) \quad \text{for } k \in K$$

    where K is the set of integers k in the interval [1,d] such that neither $l_{i,k}^{(1)} = -1$ nor $l_{k,j}^{(m)} = -1$ holds.
  - The iteration bound is given by,

$$T_\infty = \max\{l_{i,i}^{(m)}/m\}\ , \text{ for } i, m \in \{1, 2, \ldots, d\}$$

- Example :



$$L^{(1)} = \begin{bmatrix} -1 & 0 & -1 & -1 \\ 4 & -1 & 0 & -1 \\ 5 & -1 & -1 & 0 \\ 5 & -1 & -1 & -1 \end{bmatrix}$$

$$L^{(2)} = \begin{bmatrix} 4 & -1 & 0 & -1 \\ 5 & 4 & -1 & 0 \\ 5 & 5 & -1 & -1 \\ -1 & 5 & -1 & -1 \end{bmatrix}$$

$$L^{(3)} = \begin{bmatrix} 5 & 4 & -1 & 0 \\ 8 & 5 & 4 & -1 \\ 9 & 5 & 5 & -1 \\ 9 & -1 & 5 & -1 \end{bmatrix}$$

$$L^{(4)} = \begin{bmatrix} 8 & 5 & 4 & -1 \\ 9 & 8 & 5 & 4 \\ 10 & 9 & 5 & 5 \\ 10 & 9 & -1 & 5 \end{bmatrix}$$

$$T_\infty = \max\{4/2, 4/2, 5/3, 5/3, 5/3, 8/4, 8/4, 5/4, 5/4\} = 2.$$

- Minimum Cycle Mean :
  - The cycle mean m(c) of a cycle c is the average length of the edges in c, which can be found by simply taking the sum of the edge lengths and dividing by the number of edges in the cycle.
  - Minimum cycle mean is the min{m(c)} for all c.
  - The cycle means of a new graph G_d are used to compute the iteration bound. G_d is obtained from the original DFG for which iteration bound is being computed. This is done as follows:
    - # of nodes in G_d is equal to the # of delay elements in G.
    - The weight w(i,j) of the edge from node i to j in G_d is the longest path among all paths in G from delay d_i to d_j that do not pass through any delay elements.
    - The construction of G_d is thus the construction of matrix $L^{(1)}$ in LPM.
  - The cycle mean of G_d is obtained by the usual definition of cycle mean and this gives the maximum cycle bound of the cycles in G that contain the delays in c.
  - The maximum cycle mean of G_d is the max cycle bound of all cycles in G, which is the iteration bound.

- To compute the maximum cycle mean of G_d the MCM of G_d' is computed and multiplied with –1. G_d' is similar to G_d except that its weights negative of that of G_d.

Algorithm for MCM :

- Construct a series of d+1 vectors, $f^{(m)}$, m=0, 1, … , d, which are each of dimension $d \times 1$.
- An arbitrary reference node s is chosen and $f^{(0)}$is formed by setting $f^{(0)}(s)=0$ and remaining entries of $f^{(0)}$ to ∞.
- The remaining vectors $f^{(m)}$, m = 1, 2, … , d are recursively computed according to

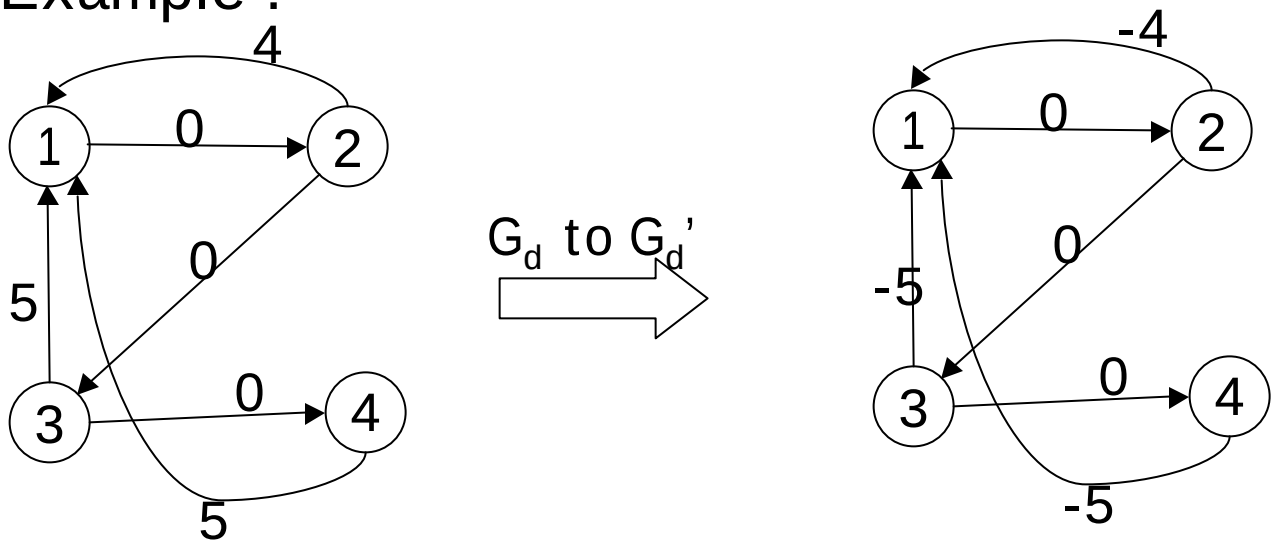
$$f^{(m)}(j) = \min(f^{(m-1)}(i) + w'(i,j)) \quad \text{for } i \in I$$

where, I is the set of nodes in G_d' such that there exists an edge from node i to node j.

- The iteration bound is given by :

$$T_\infty = -\min_{i \in \{1,2,\ldots,d\}} (\max_{m \in \{0,1,\ldots,d-1\}}((f^{(d)}(i) - f^{(m)}(i))/(d-m)))$$

- Example :



	m=0	m=1	m=2	m=3	$\max_{m \in \{0,1,\ldots,d-1\}}(f^{(d)}(i) - f^{(m)}(i))/(d-m))$
i=1	-2	-∞	-2	-3	-2
i=2	-∞	-5/3	-∞	-1	-1
i=3	-∞	-∞	-2	-∞	-2
i=4	∞-∞	∞-∞	∞-∞	∞	∞

$$T_\infty = -\min\{-2, -1, -2, \infty\} = 2$$

Chapter 3: Pipelining and Parallel Processing

Keshab K. Parhi

Outline

- Introduction
- Pipelining of FIR Digital Filters
- Parallel Processing
- Pipelining and Parallel Processing for Low Power
  - Pipelining for Lower Power
  - Parallel Processing for Lower Power
  - Combining Pipelining and Parallel Processing for Lower Power

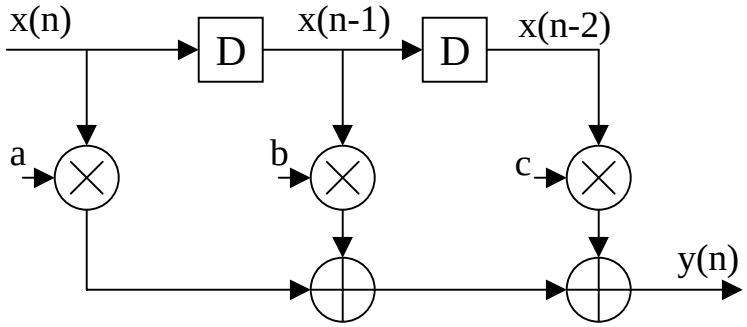
Introduction

- Pipelining
  - Comes from the idea of a water pipe: continue sending water without waiting the water in the pipe to be out

  - leads to a reduction in the critical path
  - Either increases the clock speed (or sampling speed) or reduces the power consumption at same speed in a DSP system
- Parallel Processing
  - Multiple outputs are computed in parallel in a clock period
  - The effective sampling speed is increased by the level of parallelism
  - Can also be used to reduce the power consumption

Introduction (cont'd)

- **Example 1**: Consider a 3-tap FIR filter: y(n)=ax(n)+bx(n-1)+cx(n-2)



$$T_M: \quad multiplica\,tion - time$$

$$T_A: \quad Addition - time$$

  - The critical path (or the minimum time required for processing a new sample) is limited by 1 multiply and 2 add times. Thus, the "sample period" (or the "sample frequency") is given by:

$$T_{sample} \geq T_M + 2T_A$$

$$f_{sample} \leq \frac{1}{T_M + 2T_A}$$

Introduction (cont'd)

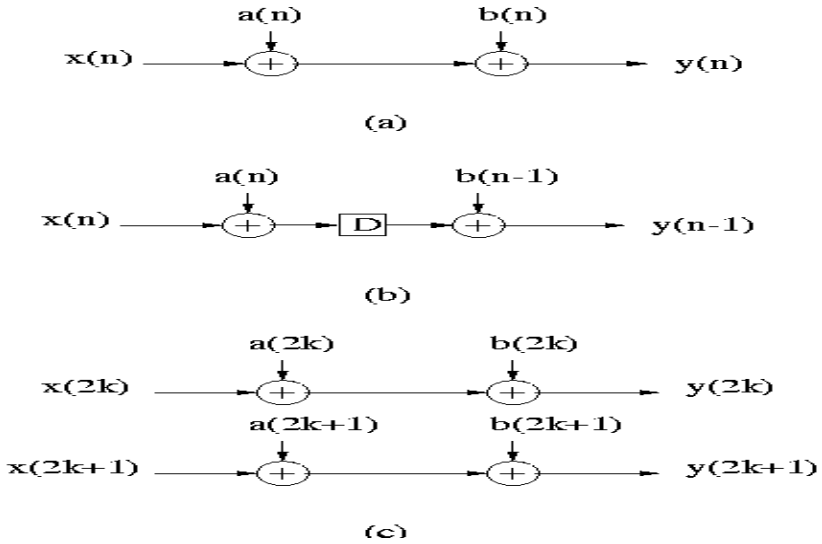
  - If some real-time application requires a faster input rate (sample rate), then this ***direct-form structure*** cannot be used! In this case, the critical path can be reduced by either ***pipelining*** or ***parallel processing***.

- Pipelining: reduce the effective critical path by introducing pipelining latches along the critical data path
- Parallel Processing: increases the sampling rate by replicating hardware so that several inputs can be processed in parallel and several outputs can be produced at the same time
- Examples of Pipelining and Parallel Processing
  - See the figures on the next page

Introduction (cont'd)

Example 2

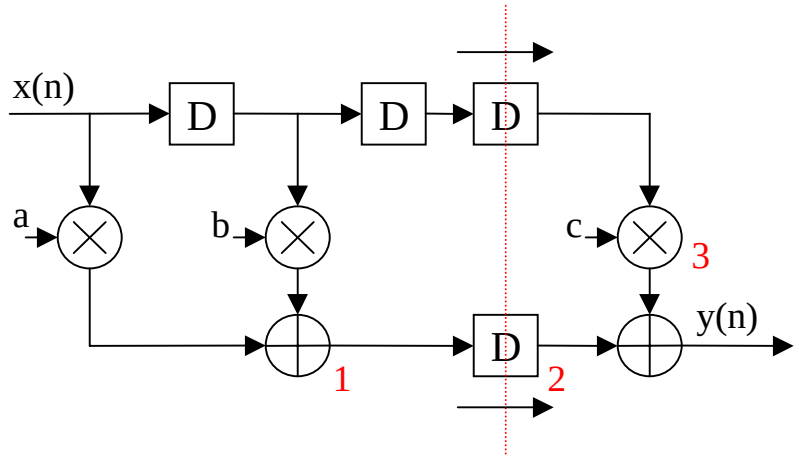
Figure (a): A data path
Figure (b): The 2-level pipelined structure of (a)
Figure (c): The 2-level parallel processing structure of (a)



Pipelining of FIR Digital Filters

- The pipelined implementation: By introducing 2 additional latches in Example 1, the critical path is reduced from T_M+2T_A to T_M+T_A.The schedule of events for this pipelined system is shown in the following table. You can see that, at any time, 2 consecutive outputs are computed in an interleaved manner.

Clock	Input	Node 1	Node 2	Node 3	Output
0	x(0)	ax(0)+bx(-1)	—	—	—
1	x(1)	ax(1)+bx(0)	ax(0)+bx(-1)	cx(-2)	y(0)
2	x(2)	ax(2)+bx(1)	ax(1)+bx(0)	cx(-1)	y(1)
3	x(3)	ax(3)+bx(2)	ax(2)+bx(1)	cx(0)	y(2)



Pipelining of FIR Digital Filters (cont'd)

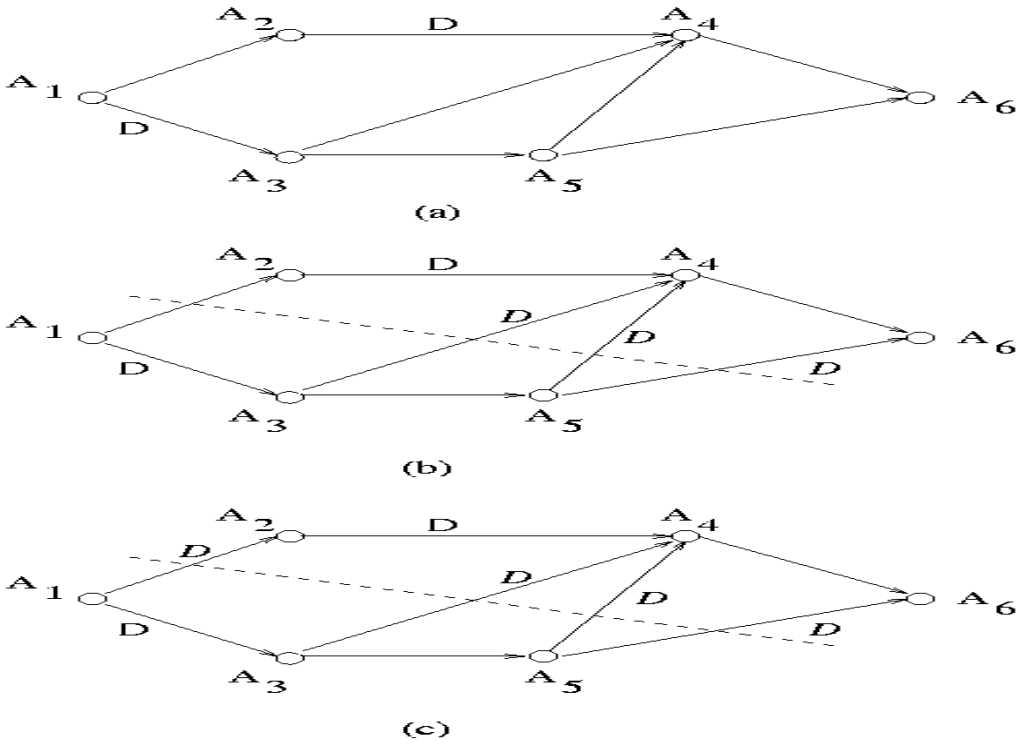
- In a pipelined system:
  - In an M-level pipelined system, the number of delay elements in any path from input to output is (M-1) greater than that in the same path in the original sequential circuit
  - Pipelining reduces the critical path, but leads to a penalty in terms of an increased latency
    - *Latency: the difference in the availability of the first output data in the pipelined system and the sequential system*
  - Two main drawbacks: increase in the number of latches and in system latency
- Important Notes:
  - The speed of a DSP architecture (or the clock period) is limited by the longest path between any 2 latches, or between an input and a latch, or between a latch and an output, or between the input and the output

Pipelining of FIR Digital Filters (cont'd)

  - This longest path or the "critical path" can be reduced by suitably placing the pipelining latches in the DSP architecture
  - The pipelining latches can only be placed across any ***feed-forward cutset*** of the graph
- Two important definitions
  - **Cutset:** a cutset is a set of edges of a graph such that if these edges are removed from the graph, the graph becomes disjoint
  - **Feed-forward cutset:** a cutset is called a feed-forward cutset if the data move in the forward direction on all the edge of the cutset
  - **Example 3:** (P.66, Example 3.2.1, see the figures on the next page)
    - (1) The critical path is $A_3 \rightarrow A_5 \rightarrow A_4 \rightarrow A_6$, its computation time: 4 u.t.
    - (2) Figure (b) is not a valid pipelining because it's not a feed-forward cutset
    - (3) Figure (c) shows 2-stage pipelining, a valid feed-forward cutset. Its critical path is 2 u.t.

Pipelining of FIR Digital Filters (cont'd)

Signal-flow graph representation of Cutset



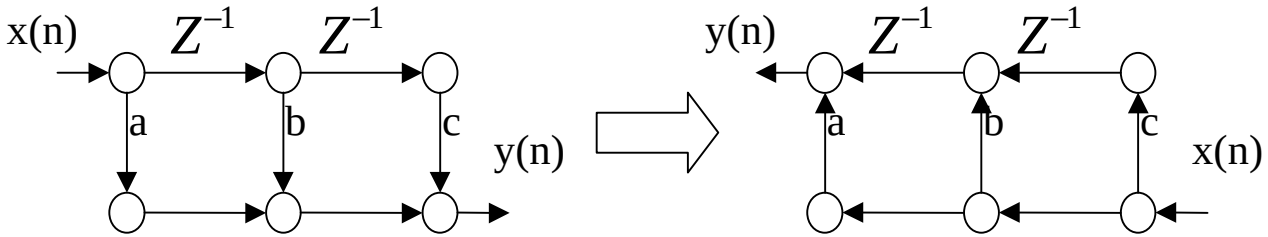
- Original SFG
- A cutset
- A feed-forward cutset

Assume:

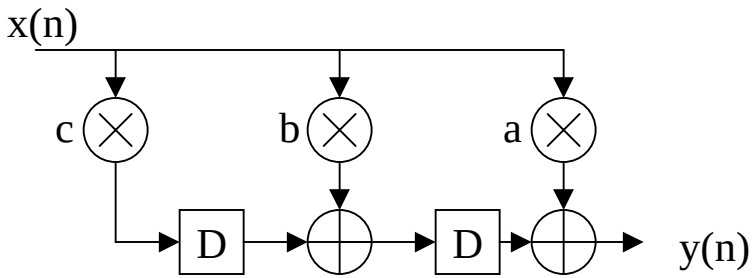
The computation time for each node is assumed to be 1 unit of time

Pipelining of FIR Digital Filters (cont'd)

- Transposed SFG and Data-broadcast structure of FIR filters
  - Transposed SFG of FIR filters



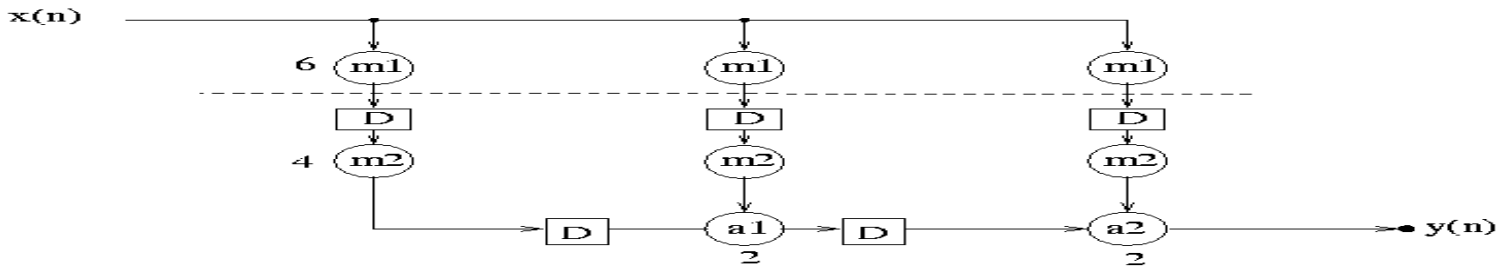
  - Data broadcast structure of FIR filters



Pipelining of FIR Digital Filters (cont'd)

- Fine-Grain Pipelining
  - Let T_M=10 units and T_A=2 units. If the multiplier is broken into 2 smaller units with processing times of 6 units and 4 units, respectively (by placing the latches on the horizontal cutset across the multiplier), then the desired clock period can be achieved as $(T_M+T_A)/2$
  - A fine-grain pipelined version of the 3-tap data-broadcast FIR filter is shown below.

Figure: fine-grain pipelining of FIR filter



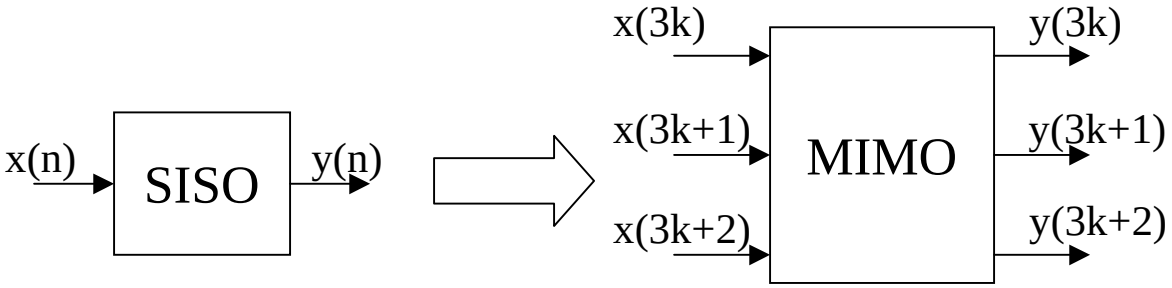
Parallel Processing

- Parallel processing and pipelining techniques are duals each other: if a computation can be pipelined, it can also be processed in parallel. Both of them exploit concurrency available in the computation in different ways.
- How to design a Parallel FIR system?
  - Consider a single-input single-output (SISO) FIR filter:
    - $y(n)=a*x(n)+b*x(n-1)+c*x(n-2)$
  - Convert the SISO system into an MIMO (multiple-input multiple-output) system in order to obtain a parallel processing structure
    - For example, to get a parallel system with 3 inputs per clock cycle (i.e., level of parallel processing L=3)

$$y(3k)=a*x(3k)+b*x(3k-1)+c*x(3k-2)$$
$$y(3k+1)=a*x(3k+1)+b*x(3k)+c*x(3k-1)$$
$$y(3k+2)=a*x(3k+2)+b*x(3k+1)+c*x(3k)$$

Parallel Processing (cont'd)

- Parallel processing system is also called ***block processing***, and the number of inputs processed in a clock cycle is referred to as the ***block size***



Sequential System 3-Parallel System

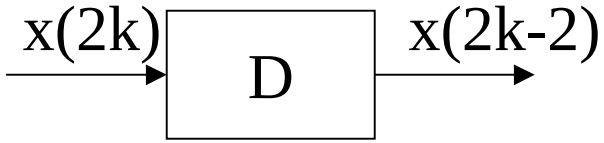
- In this parallel processing system, at the k-th clock cycle, 3 inputs x(3k), x(3k+1) and x(3K+2) are processed and 3 samples y(3k), y(3k+1) and y(3k+2) are generated at the output

- Note 1: In the MIMO structure, placing a latch at any line produces an effective delay of L clock cycles at the sample rate (L: the block size). So, each delay element is referred to as a ***block delay*** (also referred to as L-slow)

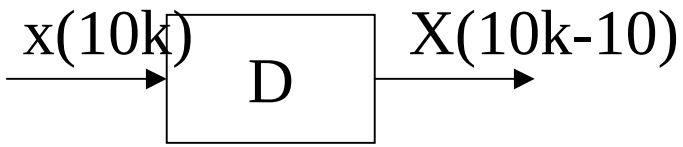
Parallel Processing (cont'd)

– For example:

When block size is 2, 1 delay element = 2 sampling delays

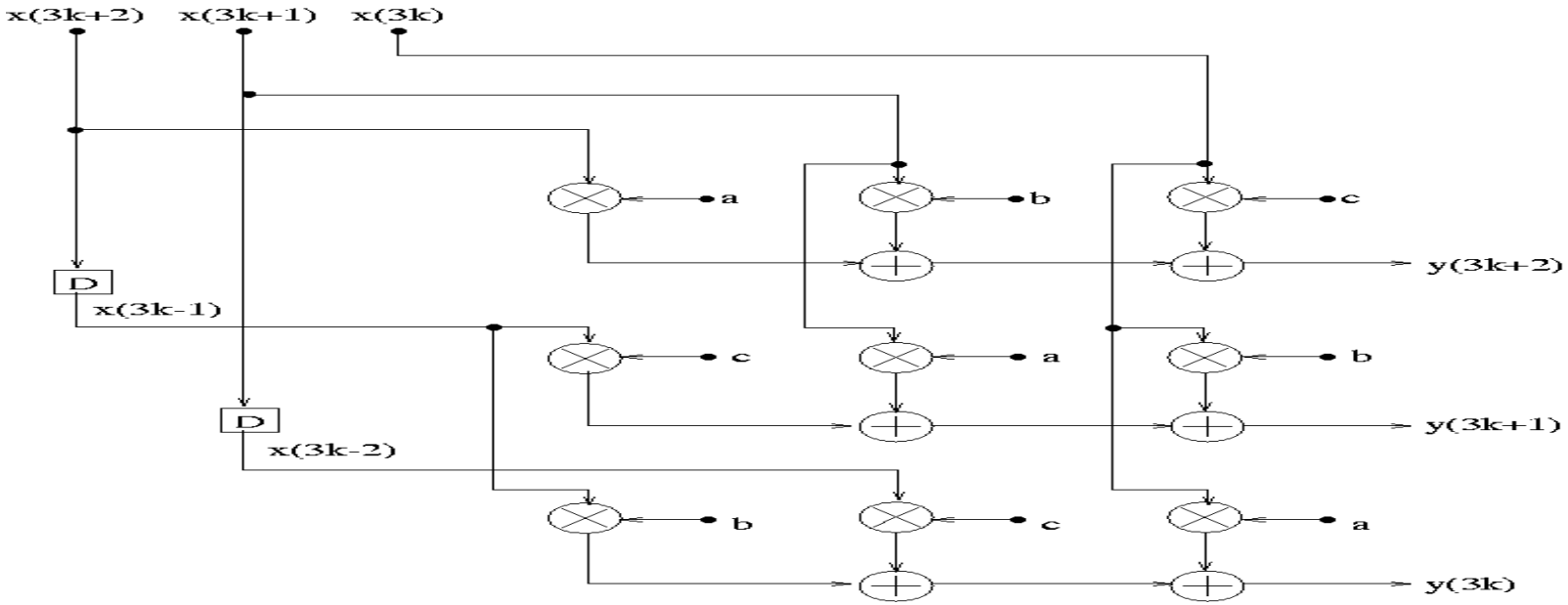


When block size is 10, 1 delay element = 10 sampling delays



Parallel Processing (cont'd)

Figure: Parallel processing architecture for a 3-tap FIR filter with block size 3



Parallel Processing (cont'd)

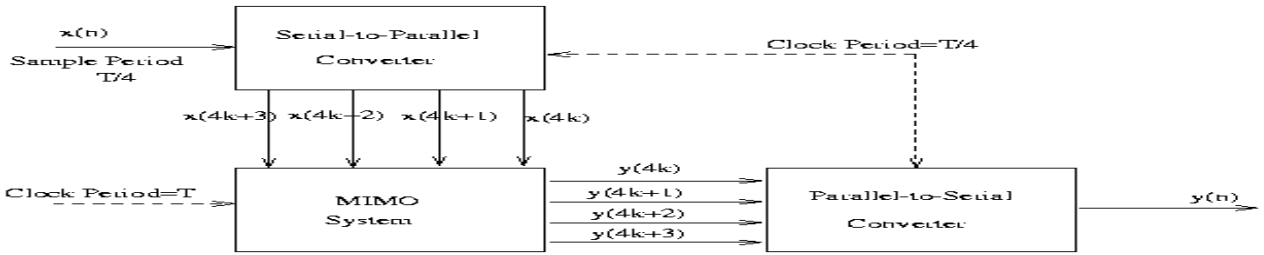
- Note 2: The critical path of the block (or parallel) processing system remains unchanged. But since 3 samples are processed in 1 (not 3) clock cycle, the iteration (or sample) period is given by the following equations:

$$T_{clock} \geq T_M + 2T_A$$

$$T_{iteration} = T_{sample} = \frac{T_{clock}}{L} \geq \frac{T_M + 2T_A}{3}$$

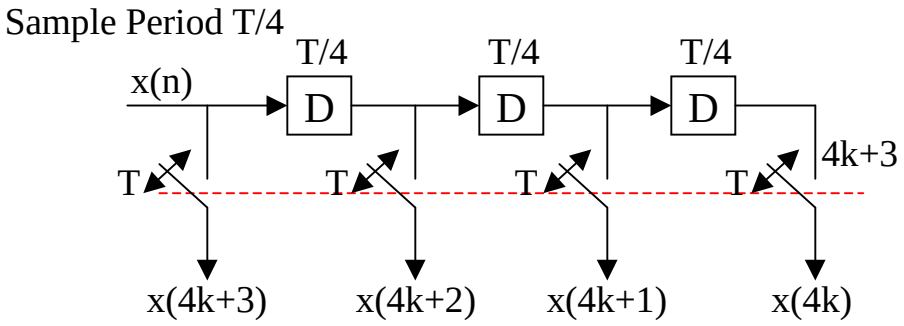
  - So, it is important to understand that in a parallel system $T_{sample} \neq T_{clock}$, whereas in a pipelined system $T_{sample} = T_{clock}$

- **Example:** A complete parallel processing system with block size 4 (including serial-to-parallel and parallel-to-serial converters) (also see P.72, Fig. 3.11)

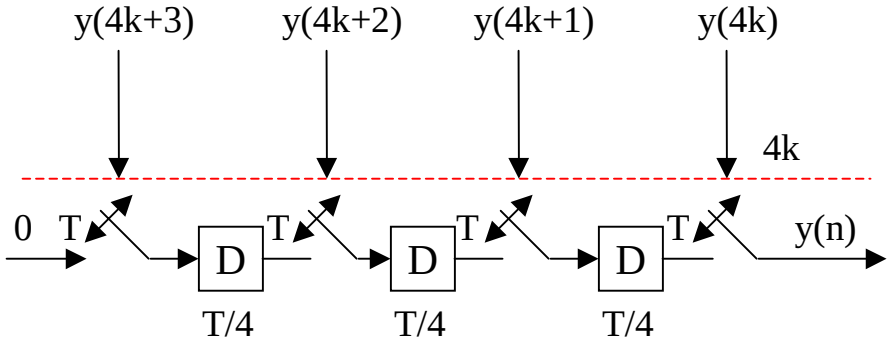


Parallel Processing (cont'd)

- A serial-to-parallel converter

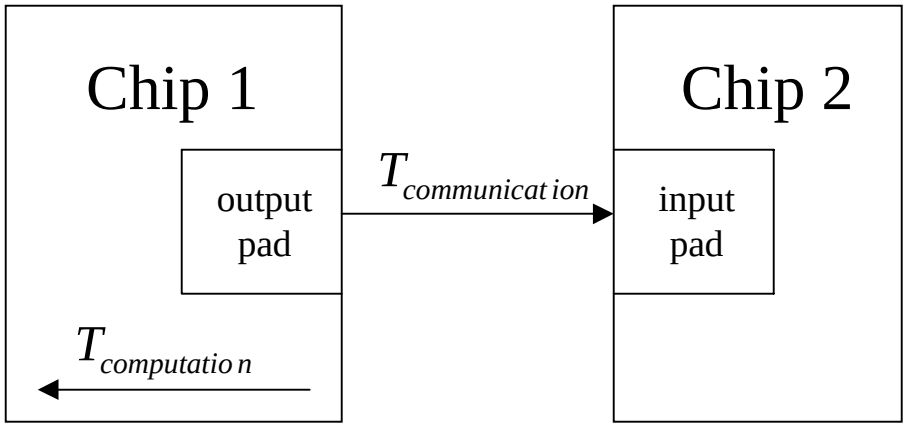


- A parallel-to-serial converter



Parallel Processing (cont'd)

- Why use parallel processing when pipelining can be used equally well?
  - Consider the following chip set, when the critical path is less than the I/O bound (output-pad delay plus input-pad delay and the wire delay between the two chips), we say this system is ***communication bounded***
  - So, we know that pipelining can be used only to the extent such that the critical path computation time is limited by the communication (or I/O) bound. Once this is reached, pipelining can no longer increase the speed



Parallel Processing (cont'd)

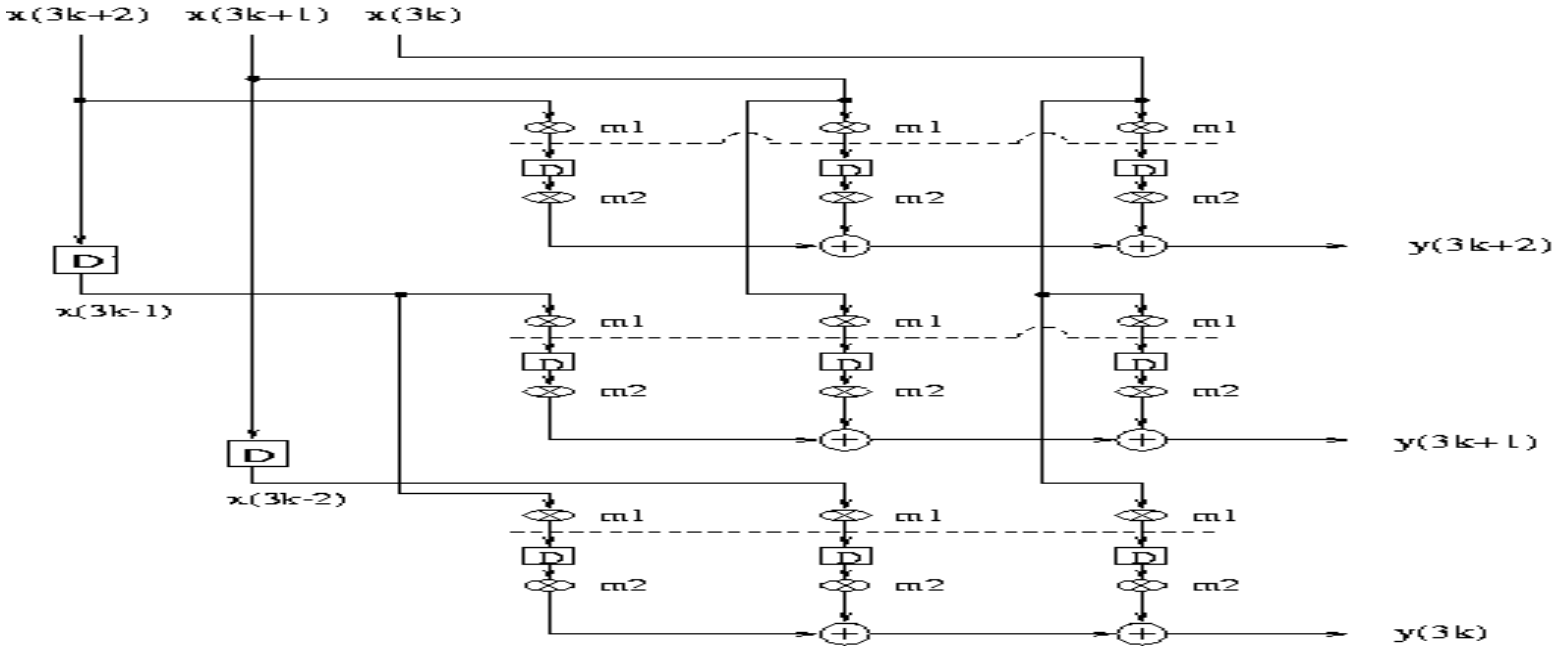
- So, in such cases, pipelining can be combined with parallel processing to further increase the speed of the DSP system
- By combining parallel processing (block size: L) and pipelining (pipelining stage: M), the sample period can be reduce to:

$$T_{iteration} = T_{sample} = \frac{T_{clock}}{L \cdot M}$$

- Example: (p.73, Fig.3.15) Pipelining plus parallel processing Example (see the next page)
- Parallel processing can also be used for reduction of power consumption while using slow clocks

Parallel Processing (cont'd)

Example:Combined fine-grain pipelining and parallel processing for 3-tap FIR filter



Pipelining and Parallel Processing for Low Power

- Two main advantages of using pipelining and parallel processing:
  - ***Higher speed*** and ***Lower power consumption***
- When sample speed does not need to be increased, these techniques can be used for lowering the power consumption
- Two important formulas:
  - Computing the propagation delay T_{pd} of CMOS circuit

$$T_{pd} = \frac{C_{charge} \cdot V_0}{k(V_0 - V_t)^2}$$

Ccharge: the capacitance to be charged or discharged in a single clock cycle

  - Computing the power consumption in CMOS circuit

$$P_{CMOS} = C_{total} \cdot V_0^2 \cdot f$$

Ctotal: the total capacitance of the CMOS circuit

Pipelining and Parallel Processing for Low Power (cont'd)

- **Pipelining for Lower Power**
  - The power consumption in the original sequential FIR filter

$$P_{seq} = C_{total} \cdot V_0^{\,2} \cdot f, \quad f = 1/T_{seq}$$

T_{seq}: the clock period of the original sequential FIR filter

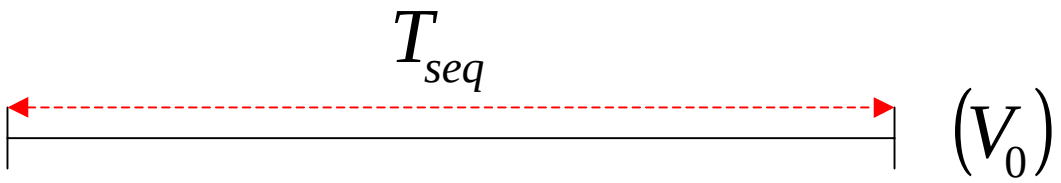
  - For a M-level pipelined system, its critical path is reduced to 1/M of its original length, and the capacitance to be charged/discharged in a single clock cycle is also reduced to 1/M of its original capacitance
  - If the same clock speed (clock frequency f) is maintained, only a fraction (1/M) of the original capacitance is charged/discharged in the same amount of time. This implies that the supply voltage can be reduced to βV_o ($0<\beta<1$). Hence, the power consumption of the pipelined filter is:

$$P_{pip} = C_{total} \cdot \beta^2 \cdot V_0^{\,2} \cdot f = \beta^2 \cdot P_{seq}$$

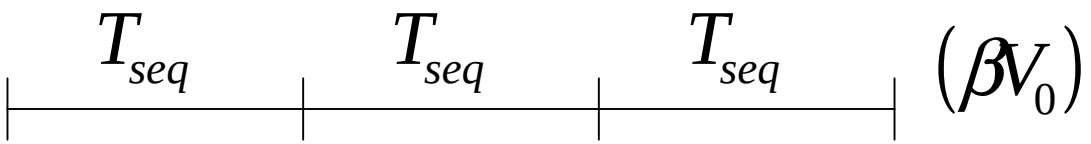
Pipelining and Parallel Processing for Low Power (cont'd)

- The power consumption of the pipelined system, compared with the original system, is reduced by a factor of β^2
- How to determine the power consumption reduction factor β?
  - Using the relationship between the propagation delay of the original filter and the pipelined filter

Sequential (critical path):



Pipelined: (critical path when M=3)



Pipelining and Parallel Processing for Low Power (cont'd)

- The propagation delays of the original sequential filter and the pipelined FIR filter are:

$$T_{seq} = \frac{C_{charge} \cdot V_0}{k(V_0 - V_t)^2}, \quad T_{pip} = \frac{(C_{charge} / M) \cdot \beta V_0}{k(\beta V_0 - V_t)^2}$$

- Since the same clock speed is maintained in both filters, we get the equation to solve β:

$$M(\beta V_0 - V_t)^2 = \beta(V_0 - V_t)^2$$

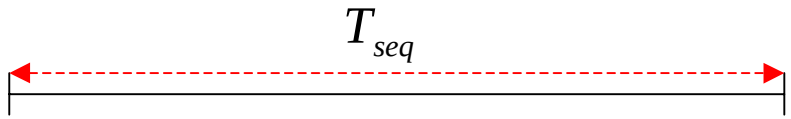
- Example: Please read textbook for Example 3.4.1 (pp.75)

Pipelining and Parallel Processing for Low Power (cont'd)

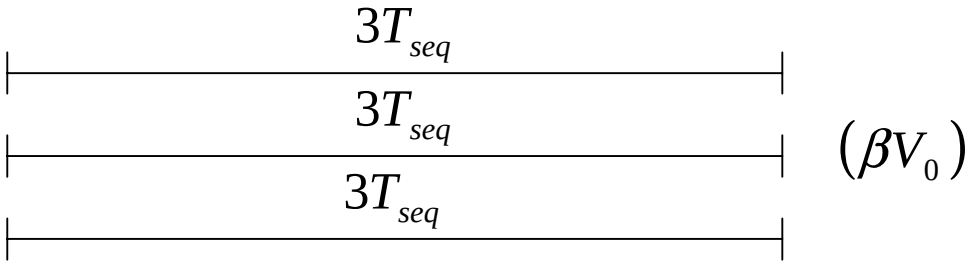
- **Parallel Processing for Low Power**
  - In an L-parallel system, the charging capacitance does not change, but the total capacitance is increased by L times
  - In order to maintain the same sample rate, the clock period of the L-parallel circuit is increased to LT_{seq} (where T_{seq} is the propagation delay of the original sequential circuit).
  - This means that the charging capacitance is charged/discharged L times longer (i.e., LT_{seq}). In other words, the supply voltage can be reduced to βV_o since there is more time to charge the same capacitance
  - How to get the power consumption reduction factor β?
    - The propagation delay consideration can again be used to compute β (Please see the next page)

Pipelining and Parallel Processing for Low Power (cont'd)

Sequential(critical path):



Parallel: (critical path when L=3)



- The propagation delay of the original system is still same, but the propagation delay of the L-parallel system is given by

$$L \cdot T_{seq} = \frac{C_{charge} \cdot \beta V_0}{k(\beta V_0 - V_t)^2}$$

Pipelining and Parallel Processing for Low Power (cont'd)

- Hence, we get the following equation to compute β:

$$L(\beta V_0 - V_t)^2 = \beta (V_0 - V_t)^2$$

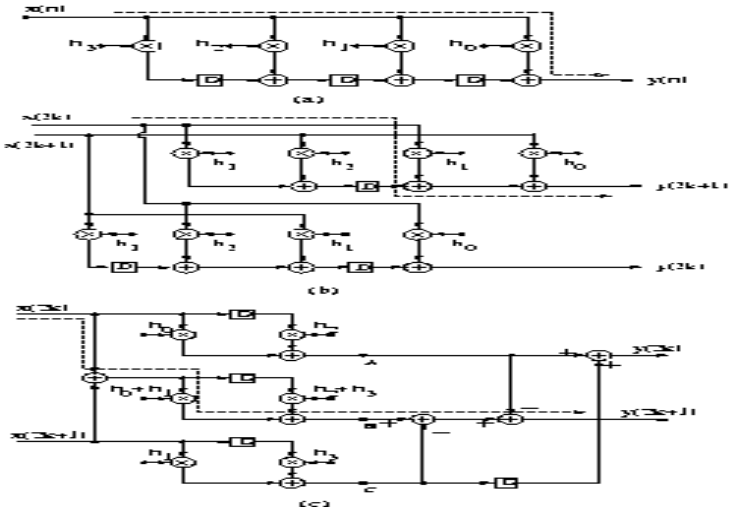
- Once β is computed, the power consumption of the L-parallel system can be calculated as

$$P_{para} = (LC_{total})(\beta V_0)^2 \frac{f}{L} = \beta^2 \cdot P_{seq}$$

– **Examples:** Please see the examples (Example 3.4.2 and Example 3.4.3) in the textbook (pp.77 and pp.80)

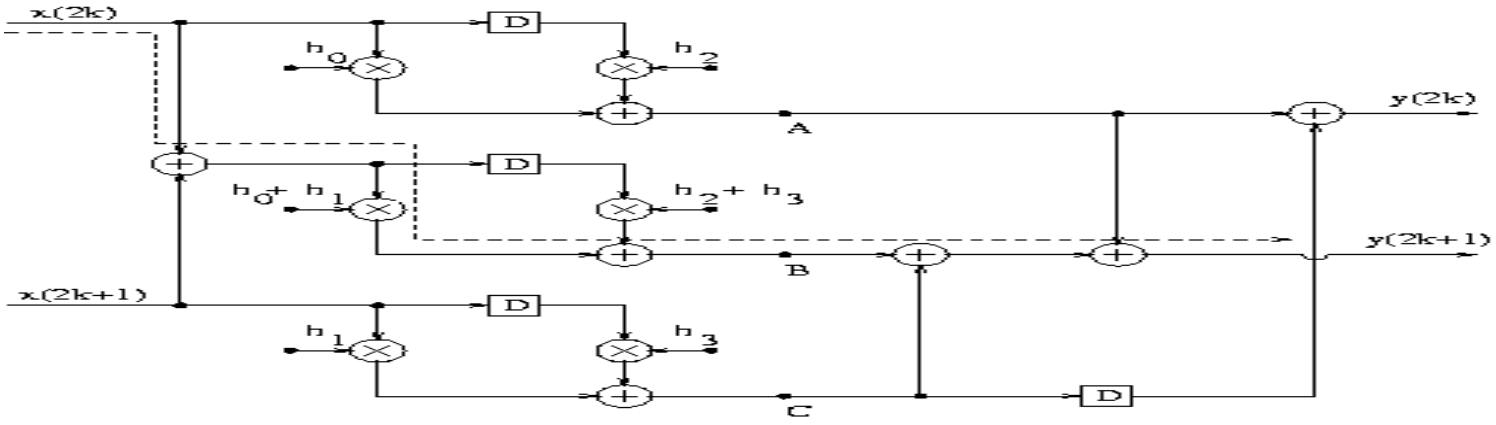
Figures for Example 3.4.2

- A 4-tap FIR filter
- A 2-parallel filter
- An area-efficient 2-parallel filter



Figures for Example 3.4.3 (pp.80)

An area-efficient 2-parallel filter and its critical path



Pipelining and Parallel Processing for Low Power (cont'd)

- **Combining Pipelining and Parallel Processing for Lower Power**
  - Pipelining and parallel processing can be combined for lower power consumption: pipelining reduces the capacitance to be charged/discharged in 1 clock period, while parallel processing increases the clock period for charging/discharging the original capacitance

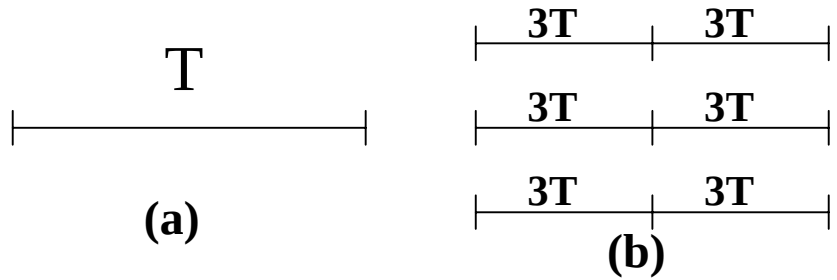


Figure: (a) Charge/discharge of entire capacitance in clock period T (b) Charge/discharge of capacitance in clock period 3T using a 3-parallel 2-pipelined FIR filter

Pipelining and Parallel Processing for Low Power (cont'd)

– The propagation delay of the L-parallel M-pipelined filter is obtained as:

$$LT_{pd} = \frac{\left(C_{ch\,\mathrm{arg}\,e} / M\right) \cdot \beta V_0}{k(\beta V_0 - V_t)^2} = \frac{L \cdot C_{ch\,\mathrm{arg}\,e} \cdot V_0}{k(V_0 - V_t)^2}$$

– Finally, we can obtain the following equation to compute β

$$LM \cdot (\beta V_0 - V_t)^2 = \beta (V_0 - V_t)^2$$

– Example: please see the example in the textbook (pp.82)

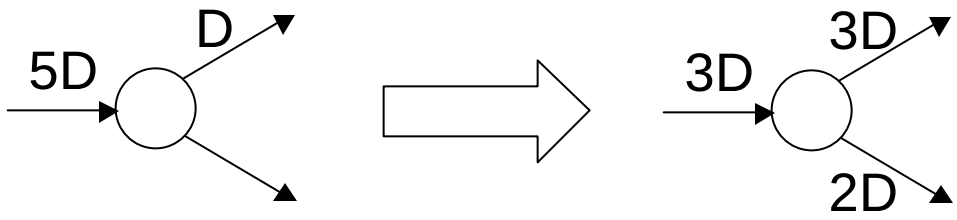
Chapter 4: Retiming

Keshab K. Parhi

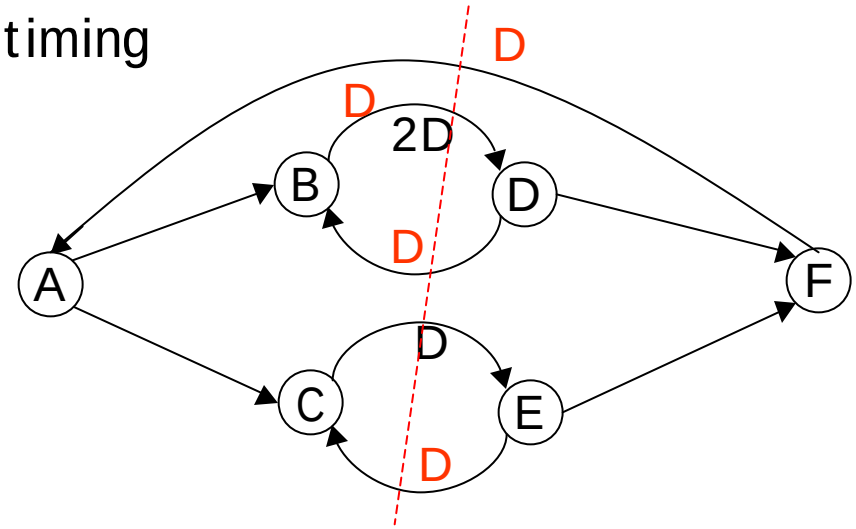
Retiming :

Moving around existing delays

- Does not alter the latency of the system
- Reduces the critical path of the system
- Node Retiming



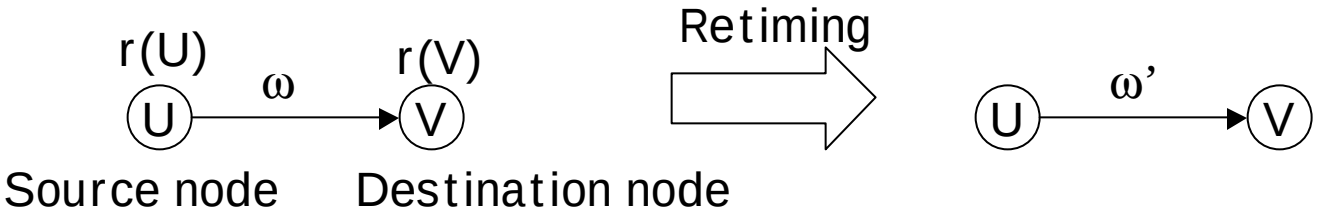
•Cutset Retiming



Retiming

- Generalization of Pipelining
- Pipelining is Equivalent to Introducing Many delays at the Input followed by Retiming

- Retiming Formulation



$$\omega' = \omega + r(V) - r(U)$$

- Properties of retiming
  - The weight of the retimed path $p = V_0 \rightarrow V_1 \rightarrow \ldots V_k$ is given by $\omega_r(p) = \omega(p) + r(V_k) - r(V_0)$
  - Retiming does not change the number of delays in a cycle.
  - Retiming does not alter the iteration bound in a DFG as the number of delays in a cycle does not change
  - Adding the constant value *j* to the retiming value of each node does not alter the number of delays in the edges of the retimed graph.
- Retiming is done to meet the following
  - Clock period minimization
  - Register minimization

- Retiming for clock period minimization
  - Feasibility constraint

$$\omega'(U,V) \geq 0 \quad \Rightarrow \quad \text{causality of the system}$$

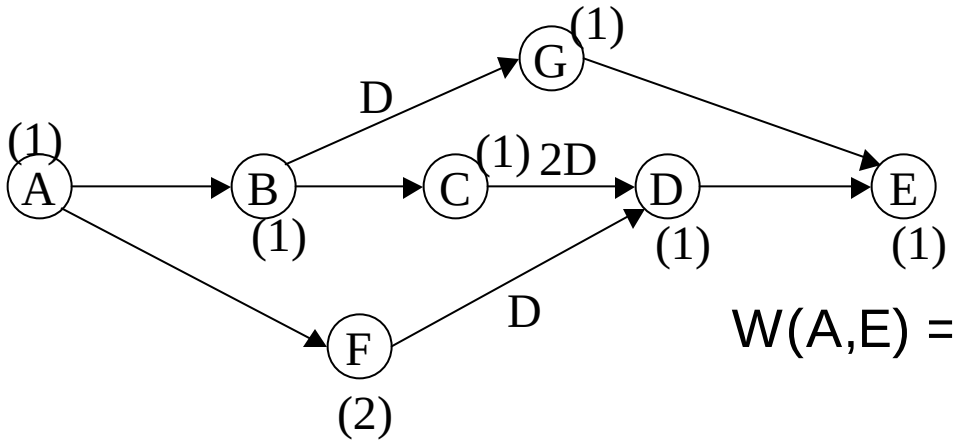
$$\Rightarrow \omega(U,V) \geq r(U) - r(V) \quad \text{(one inequality per edge)}$$

  - Critical Path constraint

$r(U) - r(V) \leq W(U,V) - 1$ for all vertices U and V in the graph such that $D(U,V) > c$ where c = target clock period. The two quantities W(U,V) and D(U,V) are given as:

$$W(U,V) = \min\{w(p) : U \rightarrow V\}$$

$$D(U,V) = \max\{t(p) : U \rightarrow V \text{ and } w(p) = W(U,V)\}$$

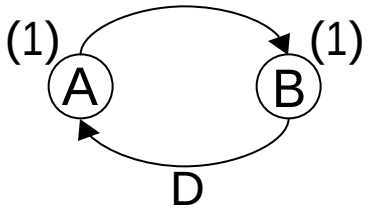


W(A,E) = 1 & D(A,E) = 5

- Algorithm to compute W(U,V) and D(U,V):
  - Let $M = t_{max}n$, where t_{max} is the maximum computation time of the nodes in G and n is the # of nodes in G.
  - Form a new graph G' which is the same as G except the edge weights are replaced by $w'(e) = Mw(e) - t(u)$ for all edges $U \rightarrow V$.
  - Solve for all pair shortest path problem on G' by using Floyd Warshall algorithm. Let S'_{UV} be the shortest path form $U \rightarrow V$.
  - If $U \neq V$, then $W(U,V) = \lceil S'_{UV}/M \rceil$ and $D(U,V) = MW(U,V) - S'_{UV} + t(V)$. If $U = V$, then $W(U,V) = 0$ and $D(U,V) = t(U)$.
- Using W(U,V) and D(U,V) the feasibility and critical path constraints are formulated to give certain inequalities. The inequalities are solved using *constraint graphs* and if a feasible solution is obtained then the circuit can be clocked with a period 'c'.

- Solving a system of inequalities : Given M inequalities in N variables where each inequality is of the form $r_i - r_j \leq k$ for integer values of k.
  - Draw a constraint graph
    - Draw the node i for each of the N variables r_i, I= 1, 2, …, N.
    - Draw the node N+1.
    - For each inequality $r_i - r_j \leq k$, draw the edge j→i of length k.
    - For each node i, i = 1, 2, …, n, draw the edge N+1 →i from the node N+1 to node I with length 0.
  - Solve using a shortest path algorithm.
    - The system of inequalities have a solution iff the constraint graph contains no negative cycles.
    - If a solution exists, one solution is where r_i is the minimum length path from the node N+1 to node i.

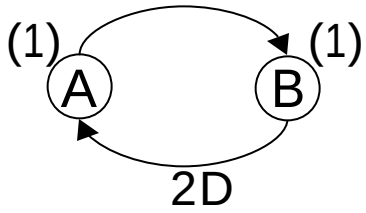
- K-slow transformation
  - Replace each D by kD



Clock	
0	$A0 \rightarrow B0$
1	$A1 \rightarrow B1$
2	$A2 \rightarrow B2$

T_{iter}= 2ut

After 2-slow transformation

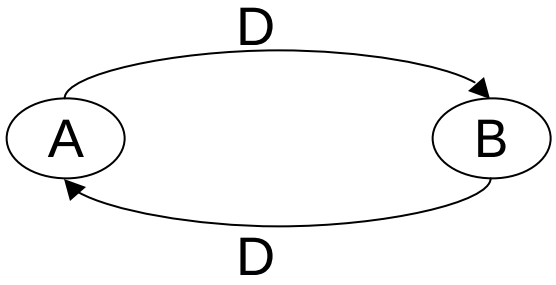


Clock	
0	$A0 \rightarrow B0$
1	
2	$A1 \rightarrow B1$
3	
4	$A2 \rightarrow B2$

T_{clk}= 2ut
T_{iter}= 2×2ut=4ut

*Input new samples every alternate cycles.
*null operations account for odd clock cycles.
*Hardware utilized only 50% time

- Retiming 2-slow graph

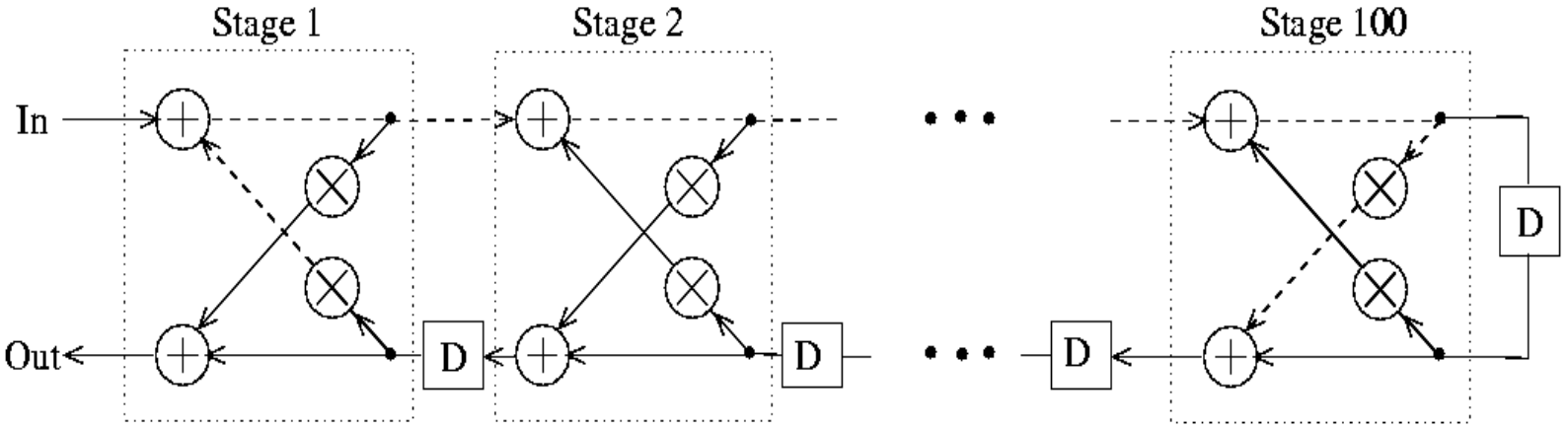


T_{clk} = 1ut
T_{iter} = 2×1=2ut

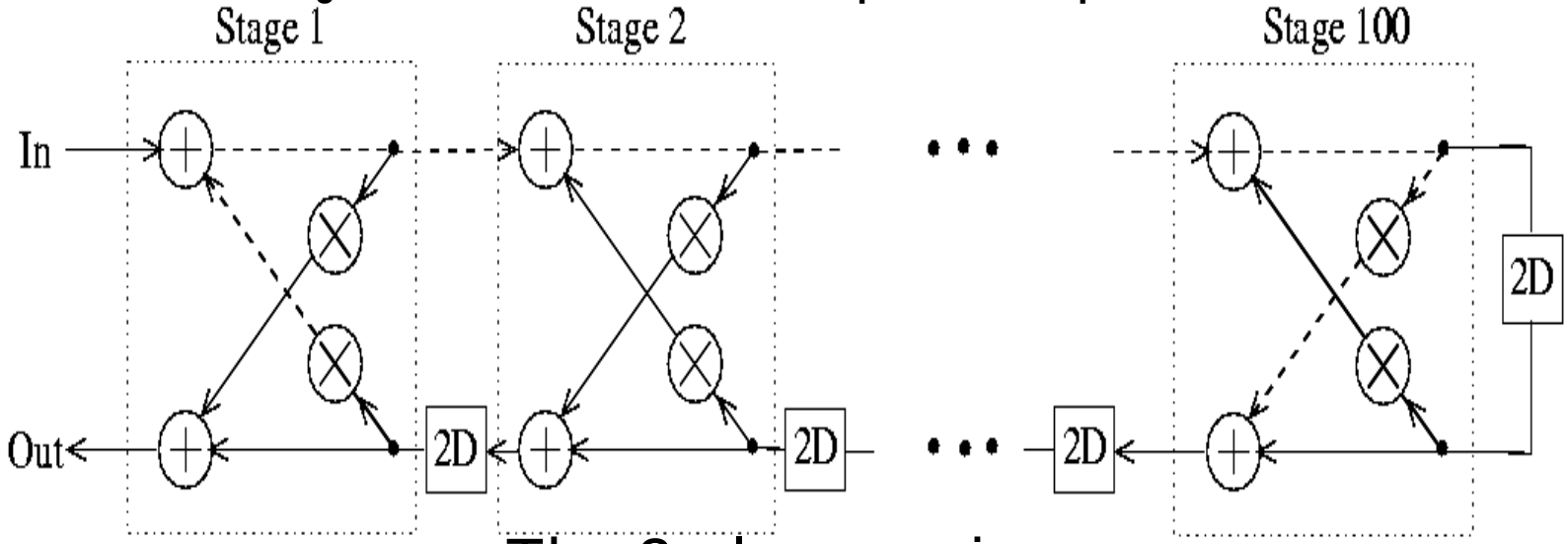
*Hardware Utilization = 50 %

*Hardware can be fully utilized if two independent operations are available.

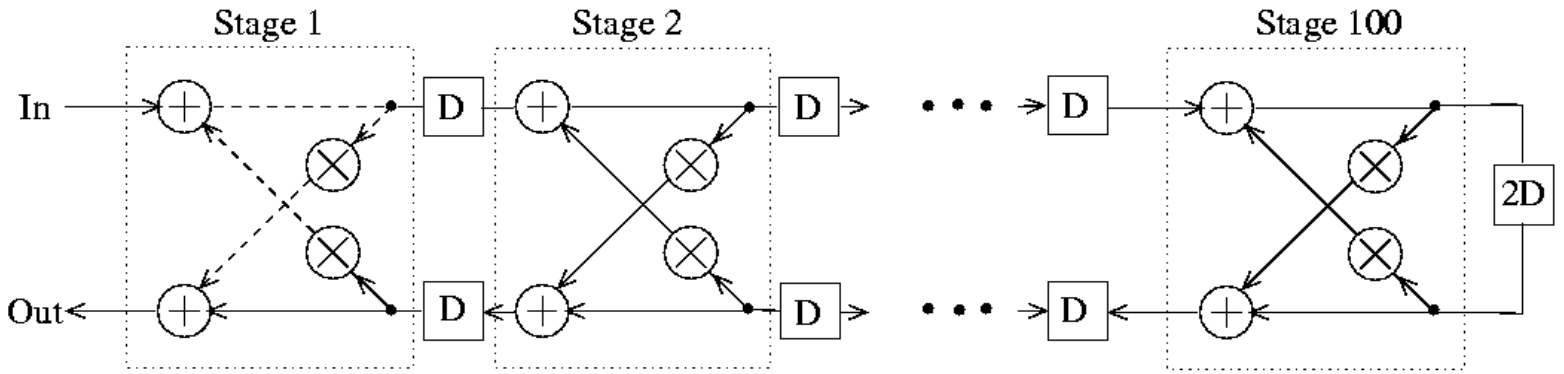
2-Slow Lattice Filter (Fig. 4.7)



A 100 stage Lattice Filter with critical path 2 multiplications and 101 additions



The 2-slow version



A retimed version of the 2 slow circuit with critical path of 2 multiplications and 2 additions

If T_m = 2 u.t. and T_a = 1 u.t., then
T_{clk} = 6 u.t., T_{iter} = 2X6 = 12 u.t.

In Original Lattice Filter, T_{iter} = 105 u.t.
Iteration Period Bound = 7 u.t.

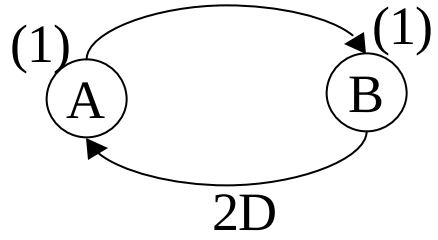
Other Applications of Retiming

- Retiming for Register Minimization (Section 4.4.3)
- Retiming for Folding (Chapter 6)
- Retiming for Power Reduction (Chap. 17)
- Retiming for Logic Synthesis (Beyond Scope of This Class)
- Multi-Rate/Multi-Dimensional Retiming (Denk/Parhi, Trans. VLSI, Dec. 98, Jun.99)

Chapter 5: Unfolding

Keshab K. Parhi

- Unfolding ≡ Parallel Processing



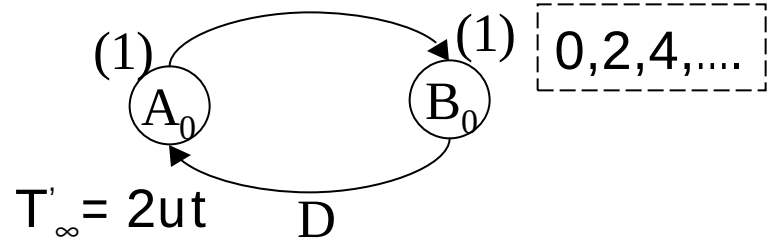
$A_0 \rightarrow B_0 \Rightarrow A_2 \rightarrow B_2 \Rightarrow A_4 \rightarrow B_4 \Rightarrow \ldots\ldots$

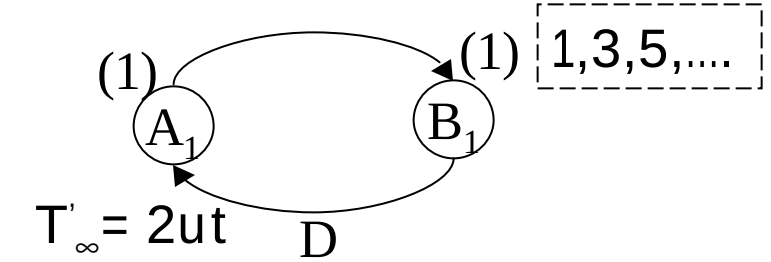
$A_1 \rightarrow B_1 \Rightarrow A_3 \rightarrow B_3 \Rightarrow A_5 \rightarrow B_5 \Rightarrow \ldots\ldots$

2 nodes & 2 edges

$T_\infty = (1+1)/2 = 1ut$

2-unfolded



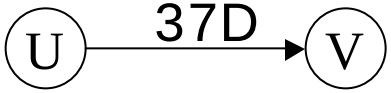


4 nodes & 4 edges

$T_\infty = 2/2 = 1ut$

- In a '*J*' unfolded system each delay is *J*-slow => if input to a delay element is the signal $x(kJ + m)$, the output is $x((k-1)J + m) = x(kJ + m - J)$.

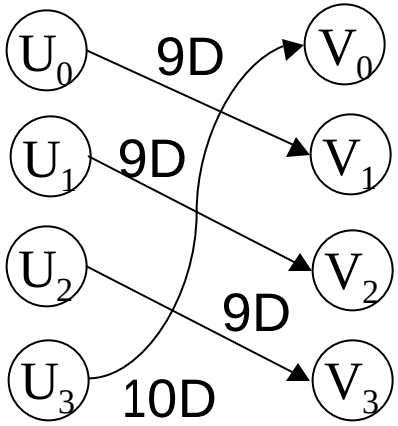
- Algorithm for unfolding:
  - For each node U in the original DFG, draw J node U_0 , U_1 , U_2 ,..., U_{J-1} .
  - For each edge $U \rightarrow V$ with w delays in the original DFG, draw the J edges $U_i \rightarrow V_{(i+w)\%J}$ with $\lfloor (i+w)/J \rfloor$ delays for i = 0, 1, ..., J-1.



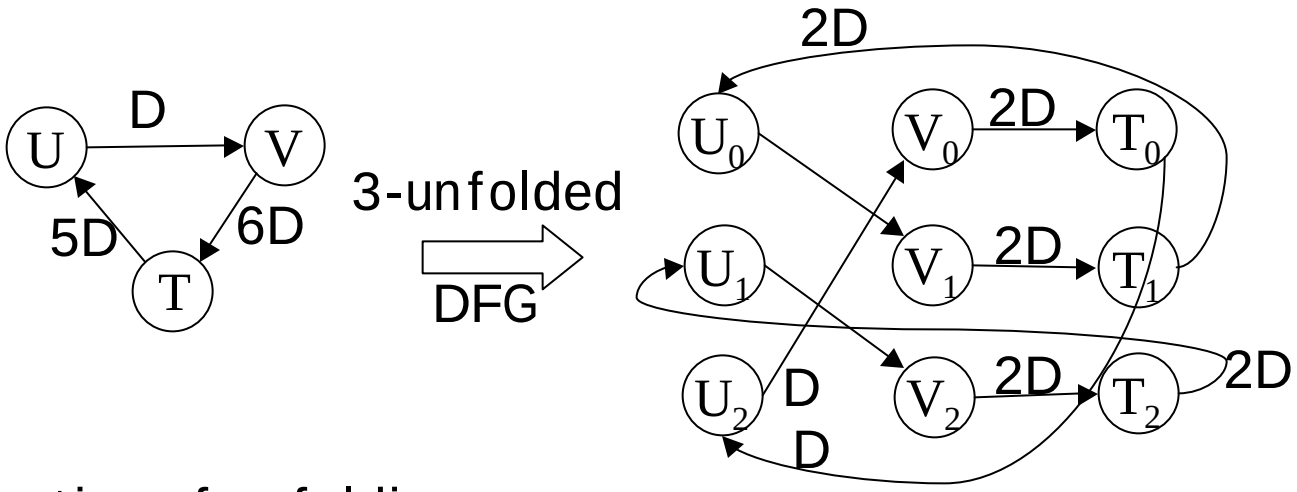
$w = 37$

$\Rightarrow \lfloor (i+w)/4 \rfloor = 9,\ i = 0,1,2$

$= 10,\ i = 3$



- Unfolding of an edge with w delays in the original DFG produces J-w edges with no delays and w edges with 1delay in J unfolded DFG for $w < J$.
- Unfolding preserves precedence constraints of a DSP program.



Properties of unfolding :

- *Unfolding preserves the number of delays in a DFG.*

  This can be stated as follows:

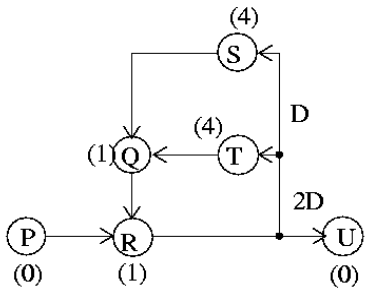
  $$\lfloor w/J \rfloor + \lfloor (w+1)/J \rfloor + \ldots + \lfloor (w+J-1)/J \rfloor = w$$

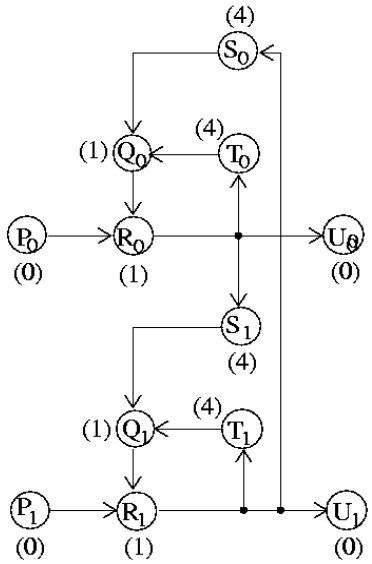
- *J-unfolding of a loop l with w_l delays in the original DFG leads to gcd(w_l, J) loops in the unfolded DFG, and each of these gcd(w_l, J) loops contains w_l/ gcd(w_l, J) delays and J/ gcd(w_l, J) copies of each node that appears in l.*
- *Unfolding a DFG with iteration bound T_∞ results in a J-unfolded DFG with iteration bound JT_∞ .*

- Applications of Unfolding
  - Sample Period Reduction
  - Parallel Processing
- Sample Period Reduction
  - Case 1 : A node in the DFG having computation time greater than T_∞.
  - Case 2 : Iteration bound is not an integer.
  - Case 3 : Longest node computation is larger than the iteration bound T_∞, and T_∞ is not an integer.

Case 1 :

- The original DFG cannot have sample period equal to the iteration bound because a node computation time is more than iteration bound

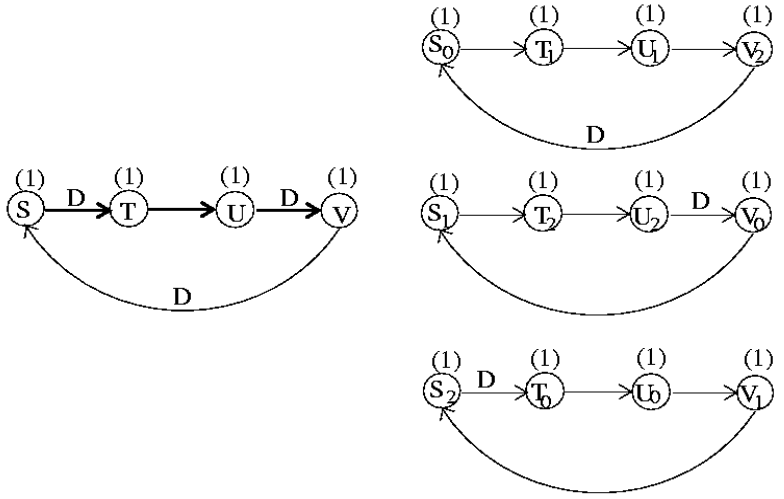




- If the computation time of a node 'U', t_u, is greater than the iteration bound T_∞, then $\lceil t_u / T_\infty \rceil$ - unfolding should be used.
- In the example, $t_u = 4$, and $T_\infty = 3$, so $\lceil 4/3 \rceil$ - unfolding i.e., 2-unfolding is used.

- **Case 2** :

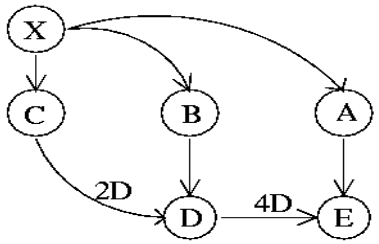
  - The original DFG cannot have sample period equal to the iteration bound because the iteration bound is not an integer.

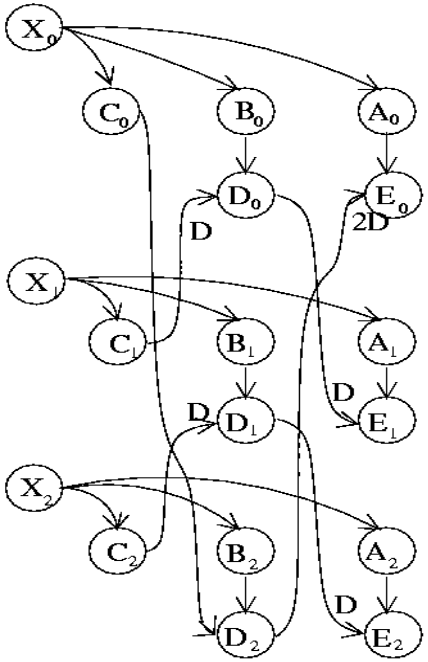


  - If a critical loop bound is of the form t_l/w_l where t_l and w_l are mutually co-prime, then w_l-unfolding should be used.
  - In the example $t_l = 60$ and $w_l = 45$, then t_l/w_l should be written as 4/3 and 3-unfolding should be used.

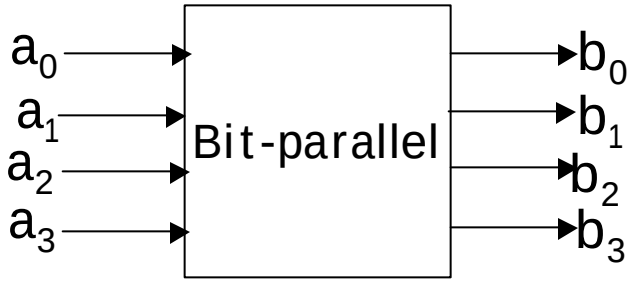
- Case 3 : In this case the minimum unfolding factor that allows the iteration period to equal the iteration bound is the min value of J such that JT_∞ is an integer and is greater than the longest node computation time.

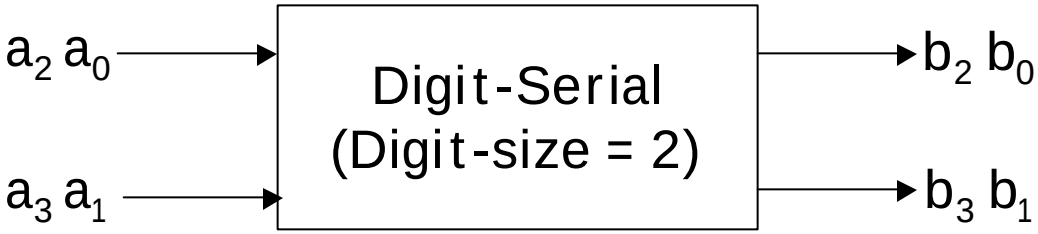
- Parallel Processing :
  - Word- Level Parallel Processing
  - Bit Level Parallel processing
    - ❖Bit-serial processing
    - ❖Bit-parallel processing
    - ❖Digit-serial processing

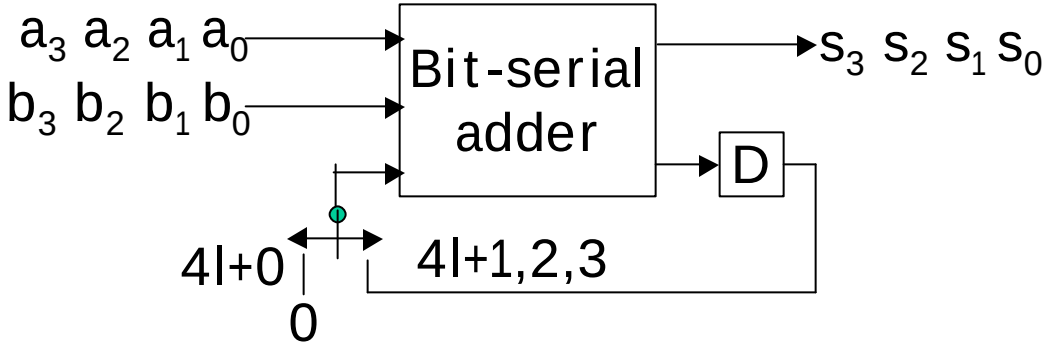




- Bit-Level Parallel Processing





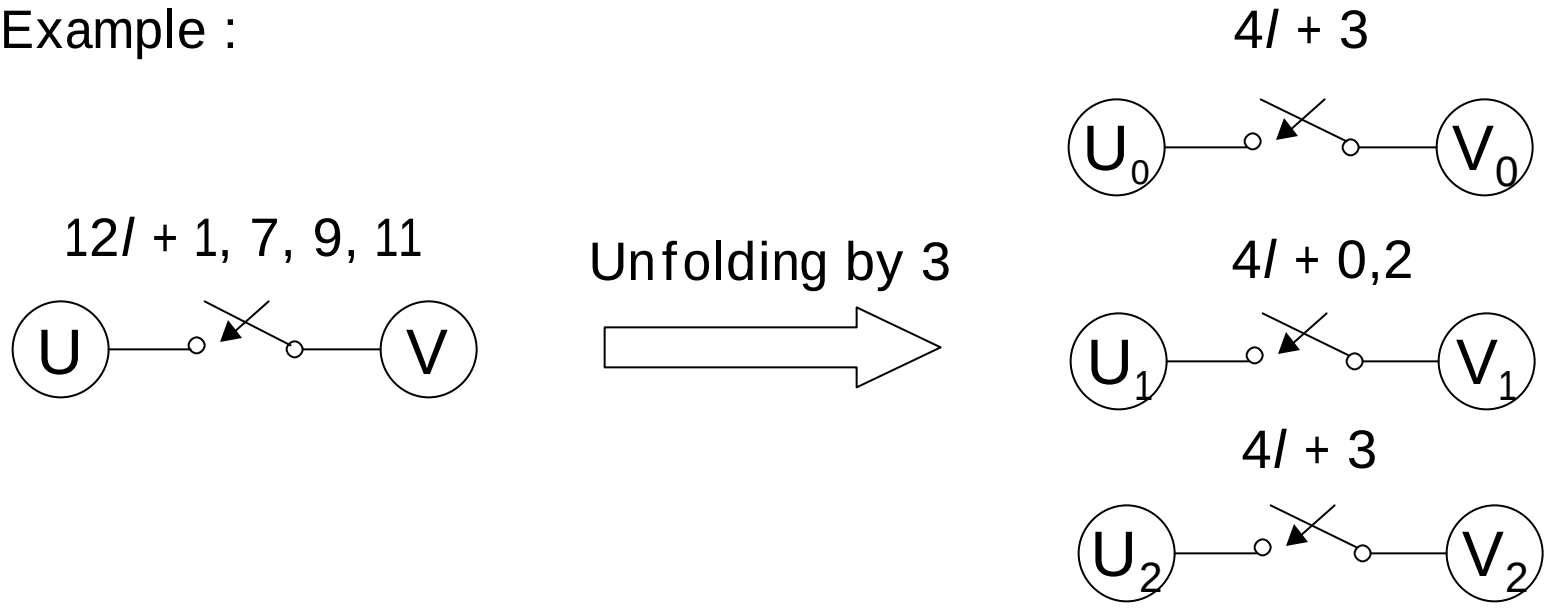


- The following assumptions are made when unfolding an edge $U \rightarrow V$:
  - The wordlength W is a multiple of the unfolding factor J, i.e. $W = W'J$.
  - All edges into and out of the switch have no delays.
- With the above two assumptions an edge $U \rightarrow V$ can be unfolded as follows :
  - Write the switching instance as

$$Wl + u = J(W'l + \lfloor u/J \rfloor) + (u\%J)$$

  - Draw an edge with no delays in the unfolded graph from the node $U_{u\%J}$ to the node $V_{u\%J}$, which is switched at time instance $(W'l + \lfloor u/J \rfloor)$.

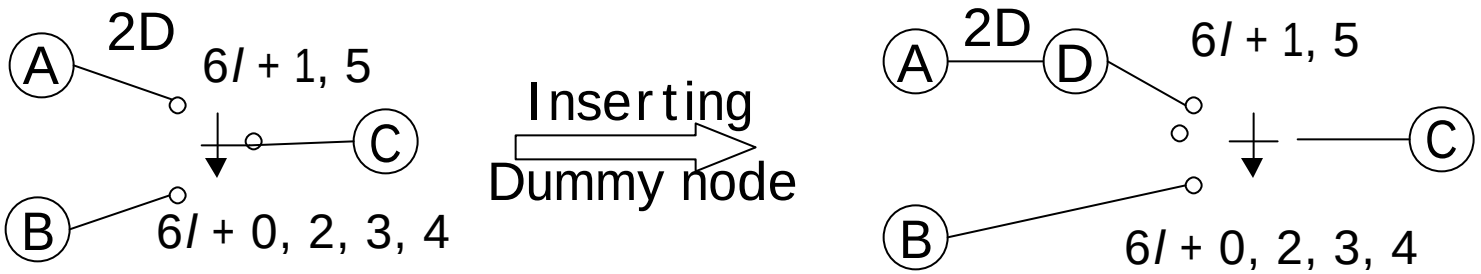
Example :

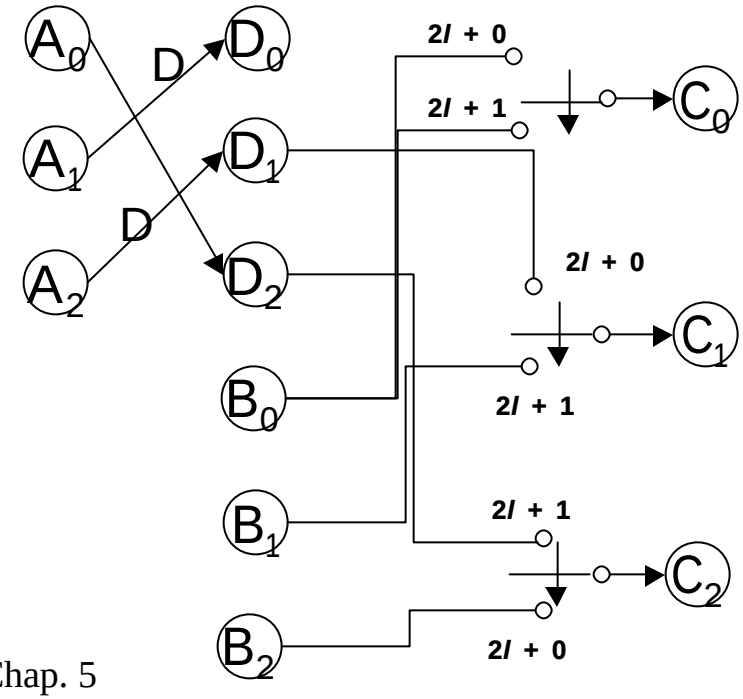


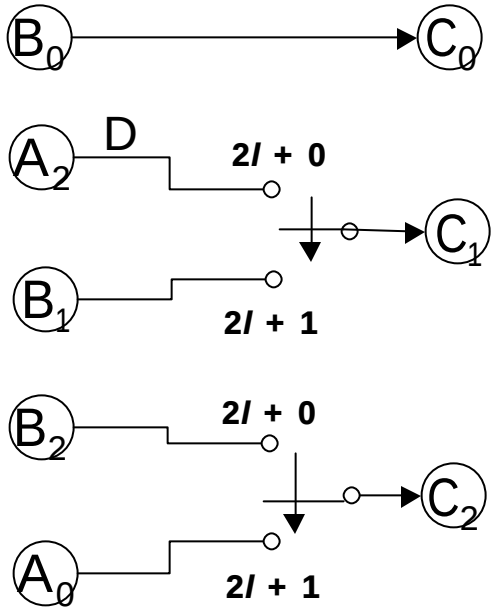
To unfold the DFG by J=3, the switching instances are as follows

$$12I + 1 = 3(4I + 0) + 1$$
$$12I + 7 = 3(4I + 2) + 1$$
$$12I + 9 = 3(4I + 3) + 0$$
$$12I + 11 = 3(4I + 3) + 2$$

- Unfolding a DFG containing an edge having a switch and a positive number of delays is done by introducing a dummy node.





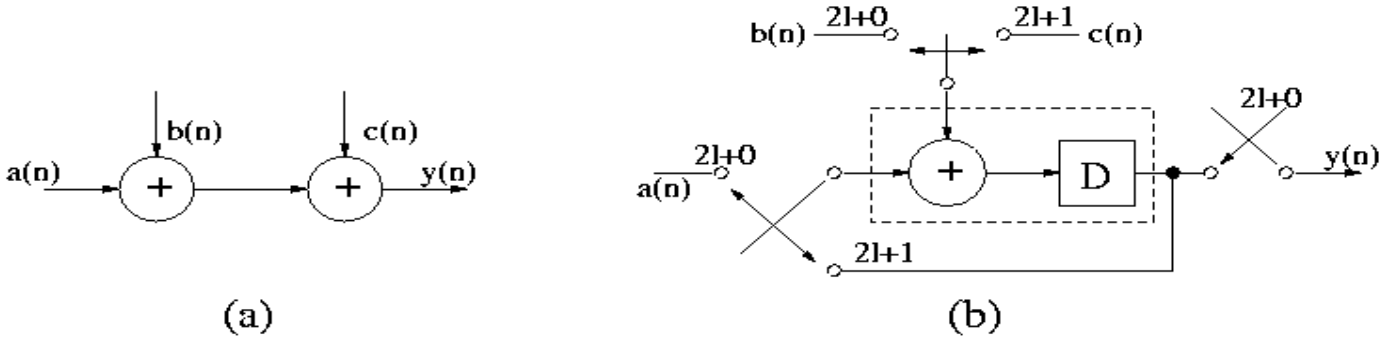


- If the word-length, W, is not a multiple of the unfolding factor, J, then expand the switching instances with periodicity lcm(W,J)
- Example: Consider W=4, J=3. Then lcm(4,3) = 12. For this case, 4l = 12l + {0,4,8), 4l+1 = 12l + {1,5,9}, 4l+2 = 12l + {2,6,10}, 4l+3 = 12l + {3,7,11}. All new switching instances are now multiples of J=3.

Chapter 6: Folding

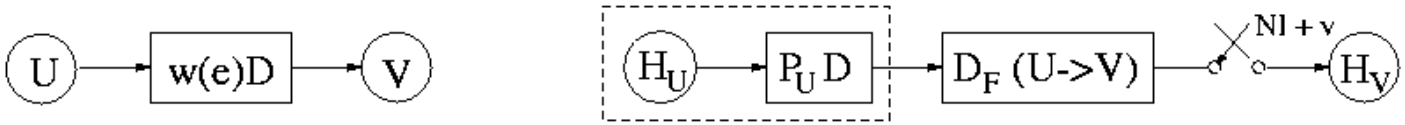
Keshab K. Parhi

- Folding is a technique to reduce the silicon area by time-multiplexing many algorithm operations into single functional units (such as adders and multipliers)



- Fig(a) shows a DSP program : y(n) = a(n) + b(n) + c(n) .
- Fig(b) shows a folded architecture where 2 additions are folded or time-multiplexed to a single pipelined adder
  One output sample is produced every 2 clock cycles $\Rightarrow$ input should be valid for 2 clock cycles.
- In general, the data on the input of a folded realization is assumed to be valid for N cycles before changing, where N is the number of algorithm operations executed on a single functional unit in hardware.

Folding Transformation :



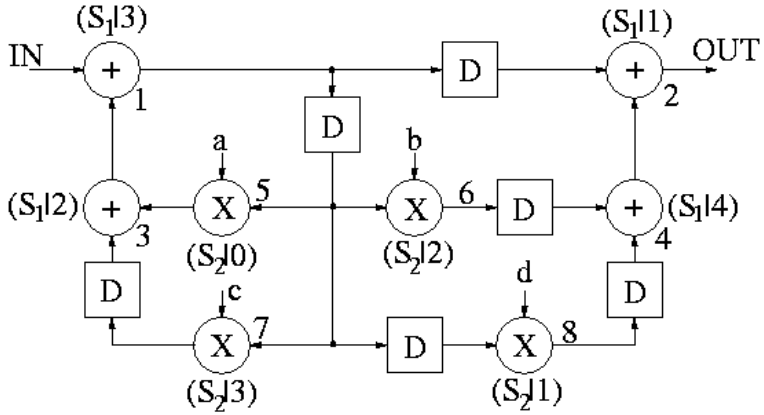
(a) (b)

- N*l* + u and N*l* + v are respectively the time units at which l-th iteration of the nodes U and V are scheduled.
- u and v are called folding orders (time partition at which the node is scheduled to be executed) and satisfy $0 \le u,v \le N-1$.
- N is the folding factor i.e., the number of operations folded to a single functional unit.
- H_u and H_v are functional units that execute u and v respectively.
- H_u is pipelined by P_u stages and its output is available at N*l* + u + P_u.
- Edge U$\rightarrow$V has w(e) delays $\Rightarrow$ the *l*-th iteration of U is used by (*l* + w(e)) th iteration of node V, which is executed at N(*l* + w(e)) + v. So, the result should be stored for :

$$D_F(U \rightarrow V) = [N(l + w(e)) + v] - [Nl + P_u + u]$$

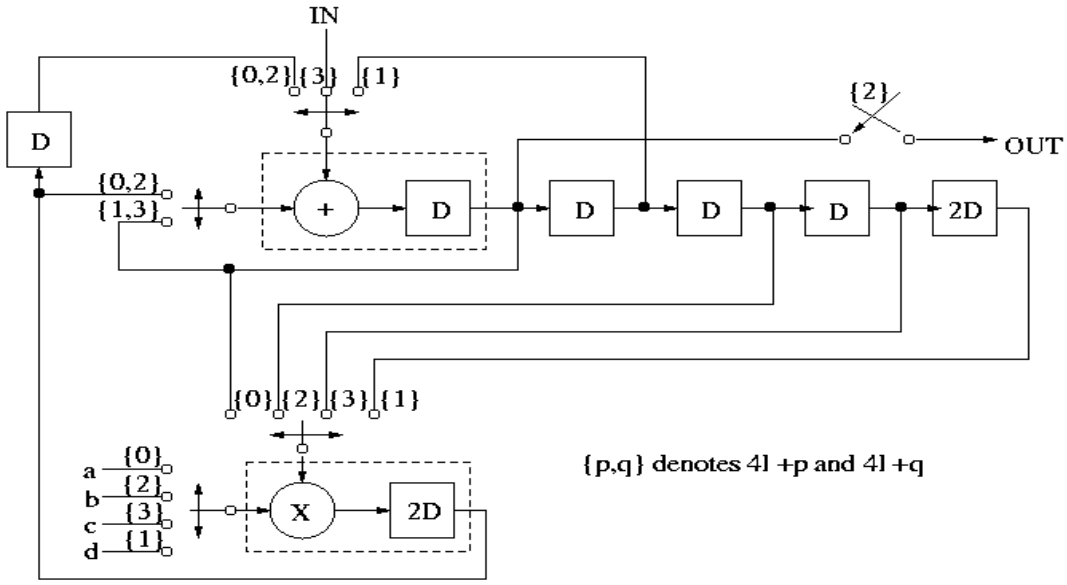
$$\Rightarrow D_F(U \rightarrow V) = Nw(e) - P_u + v - u \quad \textbf{(independent of } l \textbf{)}$$

- Folding Set : An ordered set of N operations executed by the same functional unit. The operations are ordered from 0 to N-1. Some of the operations may be null. For example, Folding set $S_1=\{A_1,0,A_2\}$ is for folding order N=3. A_1 has a folding order of 0 and A_2 of 2 and are respectively denoted by $(S_1|0)$ and $(S_2|2)$.
- Example: Folding a retimed biquad filter by N = 4.



Addition time = 1u.t., Multiplication time = 2u.t., 1 stage pipelined adder and 2 stage pipelined multiplier(i.e., $P_A=1$ and $P_M=2$)

The folding sets are $S_1 = \{4, 2, 3, 1\}$ and $S_2 = \{5, 8, 6, 7\}$



Folding equations for each of the 11 edges are as follows:

$D_F(1\rightarrow2) = 4(1) - 1 + 1 - 3 = 1$

$D_F(1\rightarrow5) = 4(1) - 1 + 0 - 3 = 0$

$D_F(1\rightarrow6) = 4(1) - 1 + 2 - 3 = 2$

$D_F(1\rightarrow7) = 4(1) - 1 + 3 - 3 = 3$

$D_F(1\rightarrow8) = 4(2) - 1 + 1 - 3 = 5$

$D_F(3\rightarrow1) = 4(0) - 1 + 3 - 2 = 0$

$D_F(4\rightarrow2) = 4(0) - 1 + 1 - 0 = 0$

$D_F(5\rightarrow3) = 4(0) - 2 + 2 - 0 = 0$

$D_F(6\rightarrow4) = 4(1) - 2 + 0 - 2 = 0$

$D_F(7\rightarrow3) = 4(1) - 2 + 2 - 3 = 1$

$D_F(8\rightarrow4) = 4(1) - 2 + 0 - 1 = 1$

- Retiming for Folding :
  - For a folded system to be realizable $D_F(U \rightarrow V) \geq 0$ for all edges.
  - If $D'_F(U \rightarrow V)$ is the folded delays in the edge $U \rightarrow V$ for the retimed graph then $D'_F(U \rightarrow V) \geq 0$.

  So,

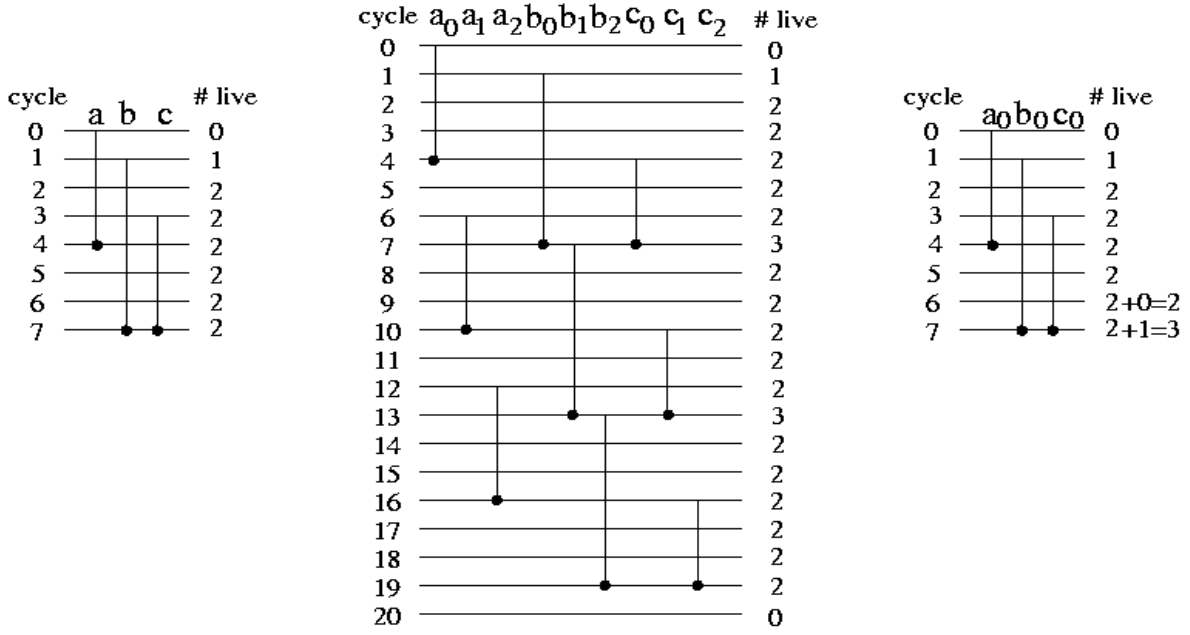
  $$Nw_r(e) - P_U + v - u \geq 0 \quad \text{... where } w_r(e) = w(e) + r(V) - r(U)$$

  $$\Rightarrow N(w(e) + r(V) - r(U)) - P_U + v - u \geq 0$$

  $$\Rightarrow r(U) - r(V) \leq D_F(U \rightarrow V) / N$$

  $$\Rightarrow r(U) - r(V) \leq \lfloor D_F(U \rightarrow V) / N \rfloor$$ (since retiming values are integers)

- Register Minimization Technique : Lifetime analysis is used for register minimization techniques in a DSP hardware.
- A ‘data sample or variable’ is live from the time it is produced through the time it is consumed. After that it is dead.
- Linear lifetime chart : Represents the lifetime of the variables in a linear fashion.
- Example

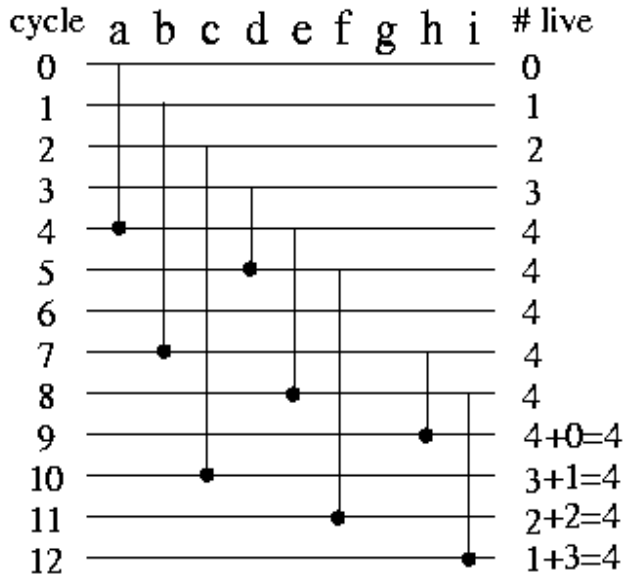


Note : Linear lifetime chart uses the convention that the variable is not live during the clock cycle when it is produced but live during the clock cycle when it is consumed.

- Due to the periodic nature of DSP programs the lifetime chart can be drawn for only one iteration to give an indication of the # of registers that are needed. This is done as follows :
  - Let N be the iteration period
  - Let the # of live variables at time partitions $n \geq N$ be the # of live variables due to 0-th iteration at cycles n-kN for $k \geq 0$. In the example, # of live variables at cycle $7 \geq N$ (=6) is the sum of the # of live variables due to the 0-th iteration at cycles 7 and $(7 - 1 \times 6) = 1$, which is 2 + 1 = 3.
- Matrix transpose example :

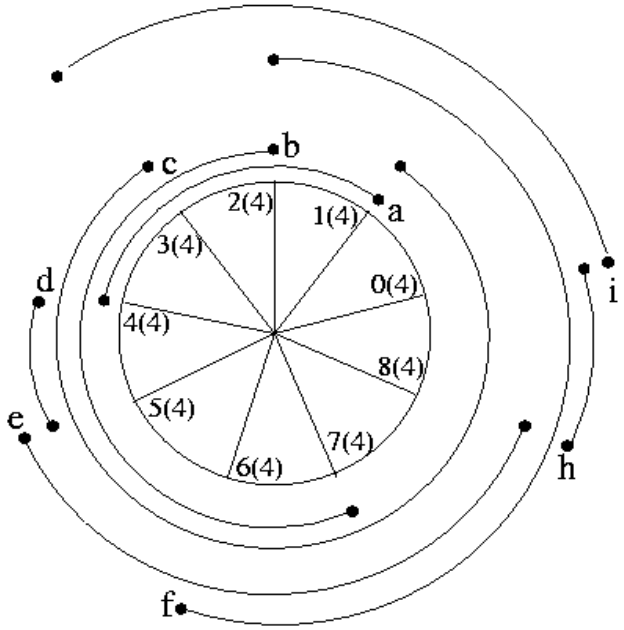
Sample	T_{in}	T_{zlout}	T_{diff}	T_{out}	Life
a	0	0	0	4	0→4
b	1	3	2	7	1→7
c	2	6	4	10	2→10
d	3	1	-2	5	3→5
e	4	4	0	8	4→8
f	5	7	2	11	5→11
g	6	2	-4	6	6→6
h	7	5	-2	9	7→9
i	8	8	0	12	8→12

- To make the system causal a latency of 4 is added to the difference so that T_{out} is the actual output time.

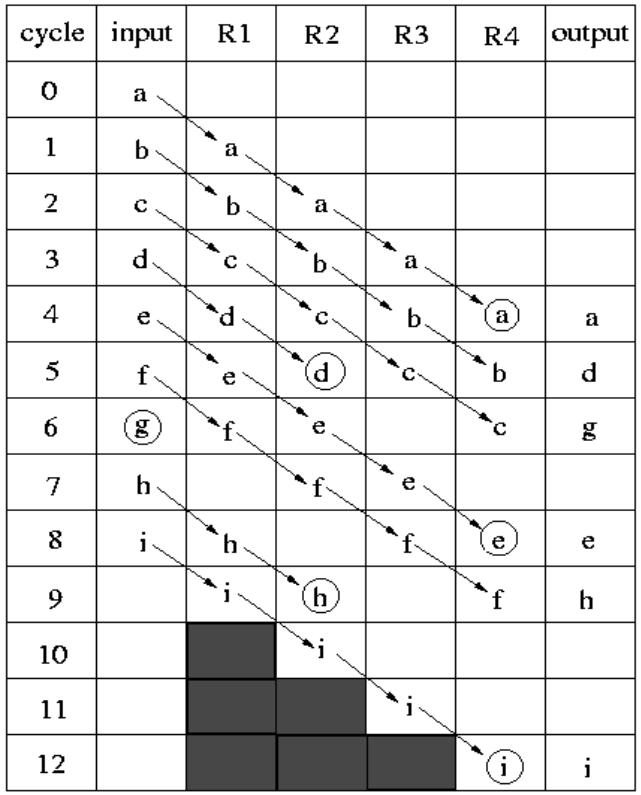


- Circular lifetime chart : Useful to represent the periodic nature of the DSP programs.
- In a circular lifetime chart of periodicity N, the point marked i ($0 \leq i \leq N - 1$) represents the time partition i and all time instances {(N*l* + i)} where *l* is any non-negative integer.
- For example : If N = 8, then time partition i = 3 represents time instances {3, 11, 19, ...}.

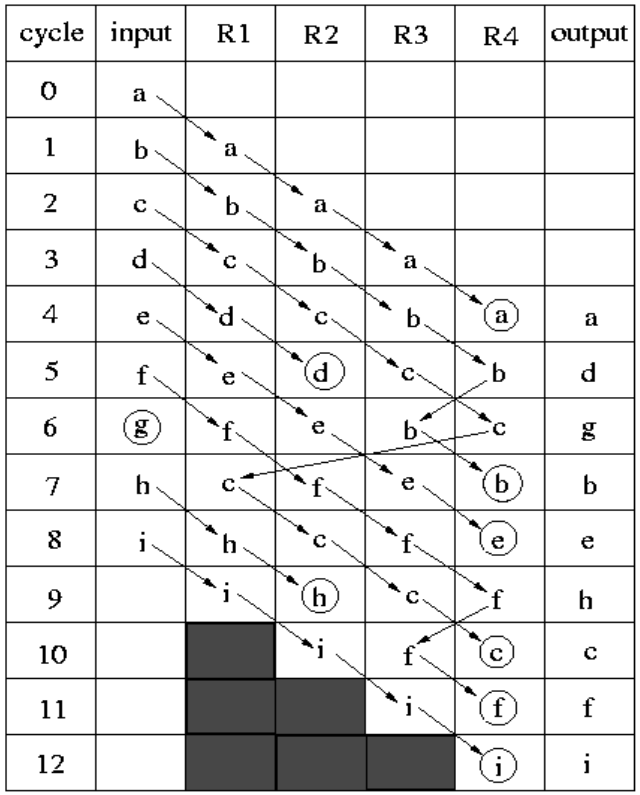
- Note : Variable produced during time unit j and consumed during time unit k is shown to be alive from 'j + 1' to 'k'.
- The numbers in the bracket in the adjacent figure correspond to the # of live variables at each time partition.



Forward Backward Register Allocation Technique :



cycle	input	R1	R2	R3	R4	output
0	a					
1	b	a				
2	c	b	a			
3	d	c	b	a		
4	e	d	c	b	(a)	a
5	f	e	(d)	c	b	d
6	(g)	f	e		c	g
7	h		f	e		
8	i	h		f	(e)	e
9		i	(h)		f	h
10			i			
11				i		
12					(i)	i



cycle	input	R1	R2	R3	R4	output
0	a					
1	b	a				
2	c	b	a			
3	d	c	b	a		
4	e	d	c	b	(a)	a
5	f	e	(d)	c	b	d
6	(g)	f	e	b	c	g
7	h	c	f	e	(b)	b
8	i	h	c	f	(e)	e
9		i	(h)	c	f	h
10			i	f	(c)	c
11				i	(f)	f
12					(i)	i

Note : Hashing is done to avoid conflict during backward allocation.

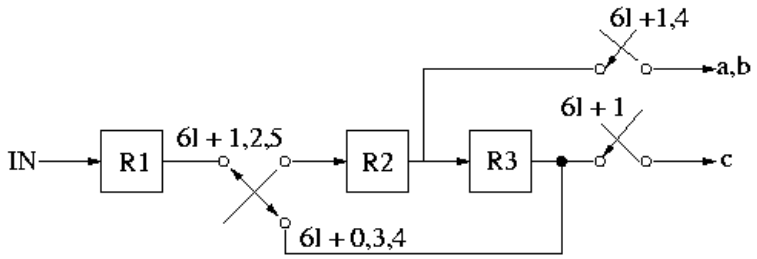
Steps for Forward-Backward Register allocation :

- Determine the minimum number of registers using lifetime analysis.
- Input each variable at the time step corresponding to the beginning of its lifetime. If multiple variables are input in a given cycle, these are allocated to multiple registers with preference given to the variable with the longest lifetime.
- Each variable is allocated in a forward manner until it is dead or it reaches the last register. In forward allocation, if the register i holds the variable in the current cycle, then register i + 1 holds the same variable in the next cycle. If (i + 1)-th register is not free then use the first available forward register.
- Being periodic the allocation repeats in each iteration. So hash out the register R_j for the cycle l + N if it holds a variable during cycle l.
- For variables that reach the last register and are still alive, they are allocated in a backward manner on a first come first serve basis.
- Repeat steps 4 and 5 until the allocation is complete.

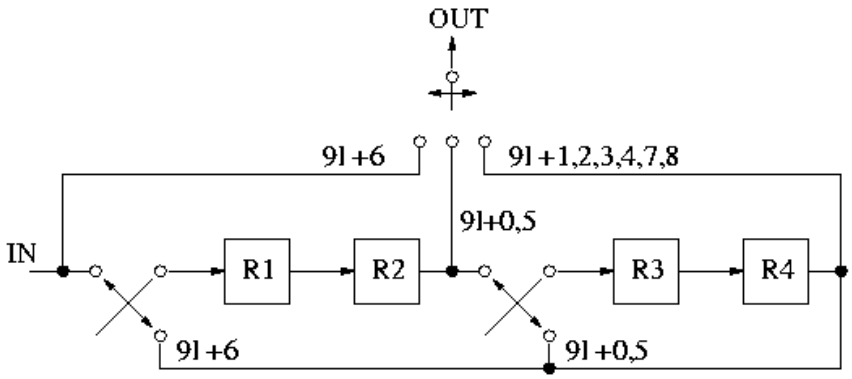
- Example : Forward backward Register Allocation

cycle	input	R1	R2	R3	output
0	a				
1	b	a			
2		b	a		
3			b	a	
4	c			b	
5		c			
6			c		
7				(c)	c

cycle	input	R1	R2	R3	output
0	a				
1	b	a			
2		b	a		
3			b	a	
4	c		(a)	b	a
5		c	b		
6			c	b	
7			(b)	(c)	b, c



- Folded architecture for matrix tranposer :



- Register minimization in folded architectures :
  - Perform retiming for folding
  - Write the folding equations
  - Use the folding equations to construct a lifetime table
  - Draw the lifetime chart and determine the required number of registers
  - Perform forward-backward register allocation
  - Draw the folded architecture that uses the minimum number of registers.

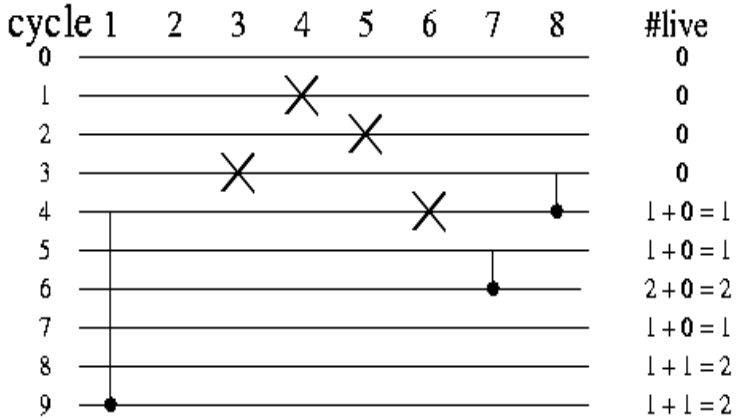
•Example : Biquad Filter

- Steps 1 & 2 have already been done.
- Step 3:The lifetime table is then constructed. The 2nd row is empty as $D_F(2 \rightarrow U)$ is not present.

Note : As retiming for folding ensures causality, we need not add any latency.

Node	$T_{in} \rightarrow T_{out}$
1	4→9
2	--
3	3→3
4	1→1
5	2→2
6	4→4
7	5→6
8	3→4

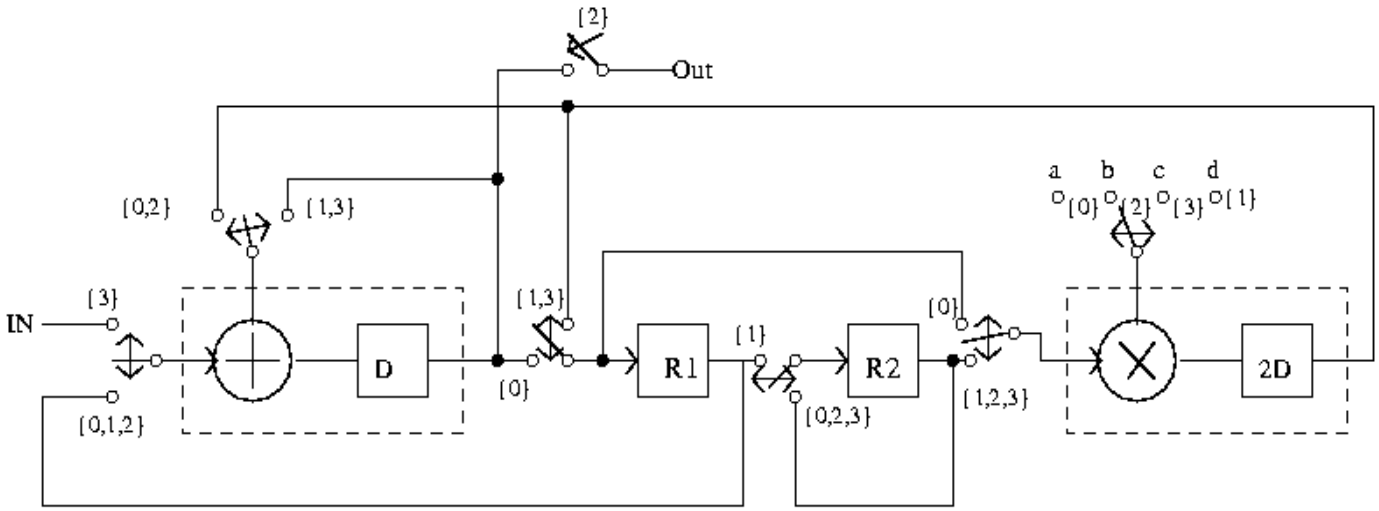
- Step 4 : Lifetime chart is constructed and registers determined.



- Step 5 : Forward-backward register allocation

cycle	input	R1	R2	output
0				
1				
2				
3	n_8			
4	n_1	n_8		n_8
5	n_7	n_1		
6		n	n_1	n_7
7			n_1	
8			n_1	
9			n	n_1

➢Folded architecture is drawn with minimum # of registers.



Chapter 7: Systolic Architecture Design

Keshab K. Parhi

- Systolic architectures are designed by using linear mapping techniques on regular dependence graphs (DG).
- Regular Dependence Graph : The presence of an edge in a certain direction at any node in the DG represents presence of an edge in the same direction at all nodes in the DG.
- DG corresponds to space representation → no time instance is assigned to any computation ⇒ t=0.
- Systolic architectures have a space-time representation where each node is mapped to a certain processing element(PE) and is scheduled at a particular time instance.
- Systolic design methodology maps an N-dimensional DG to a lower dimensional systolic architecture.
- Mapping of N-dimensional DG to (N-1) dimensional systolic array is considered.

- Definitions :
  - Projection vector (also called iteration vector), $d = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$

    Two nodes that are displaced by d or multiples of d are executed by the same processor.
  - Processor space vector, $p^T = (p_1 \quad p_2)$

    Any node with index I^T=(i,j) would be executed by processor;

$$p^T I = (p_1 \quad p_2) \begin{pmatrix} i \\ j \end{pmatrix}$$

  - Scheduling vector, $s^T = (s_1\ s_2)$. Any node with index I would would be executed at time, $s^T I$.
  - Hardware Utilization Efficiency, HUE = $1/|S^T d|$. This is because two tasks executed by the same processor are spaced $|S^T d|$ time units apart.
  - Processor space vector and projection vector must be orthogonal to each other $\Rightarrow p^T d = 0$.

- If A and B are mapped to the same processor, then they cannot be executed at the same time, i.e., $S^T I_A \neq S^T I_B$, i.e., $S^T d \neq 0$.
- Edge mapping : If an edge e exists in the space representation or DG, then an edge $p^T e$ is introduced in the systolic array with $s^T e$ delays.
- A DG can be transformed to a space-time representation by interpreting one of the spatial dimensions as temporal dimension. For a 2-D DG, the general transformation is described by i' = t = 0, j' = $p^T I$, and t' = $s^T I$, i.e.,

$$\begin{pmatrix} i' \\ j' \\ t' \end{pmatrix} = T \begin{pmatrix} i \\ j \\ t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ & p' & 0 \\ & s' & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ t \end{pmatrix}$$

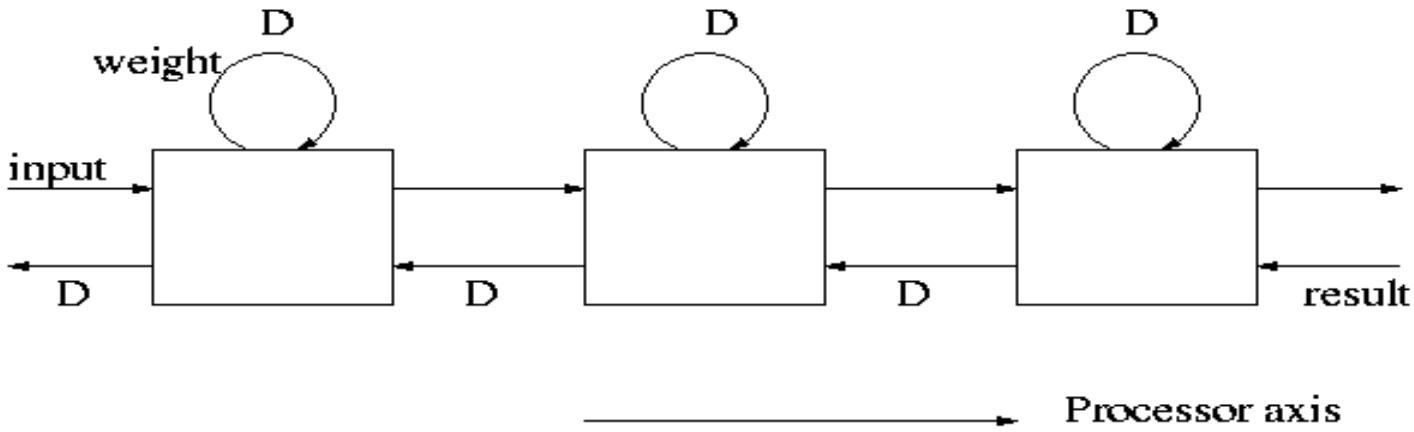
j' $\Rightarrow$ processor axis
t' $\Rightarrow$ scheduling time instance

FIR Filter Design B_1 (Broadcast Inputs, Move Results, Weights Stay)

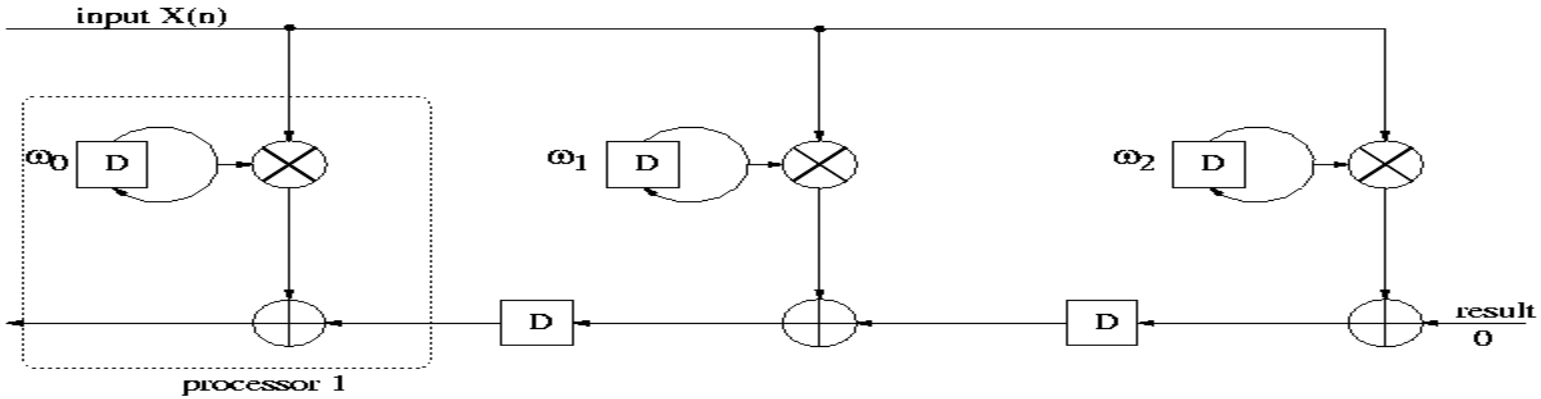
$$d^T = (1\ 0),\ p^T = (0\ 1),\ s^T = (1\ 0)$$

- Any node with index $I^T = (i\ ,\ j)$
  - is mapped to processor $p^T I = j$.
  - is executed at time $s^T I = i$.
- Since $s^T d = 1$ we have HUE = $1/|s^T d| = 1$.
- Edge mapping : The 3 fundamental edges corresponding to weight, input, and result can be mapped to corresponding edges in the systolic array as per the following table:

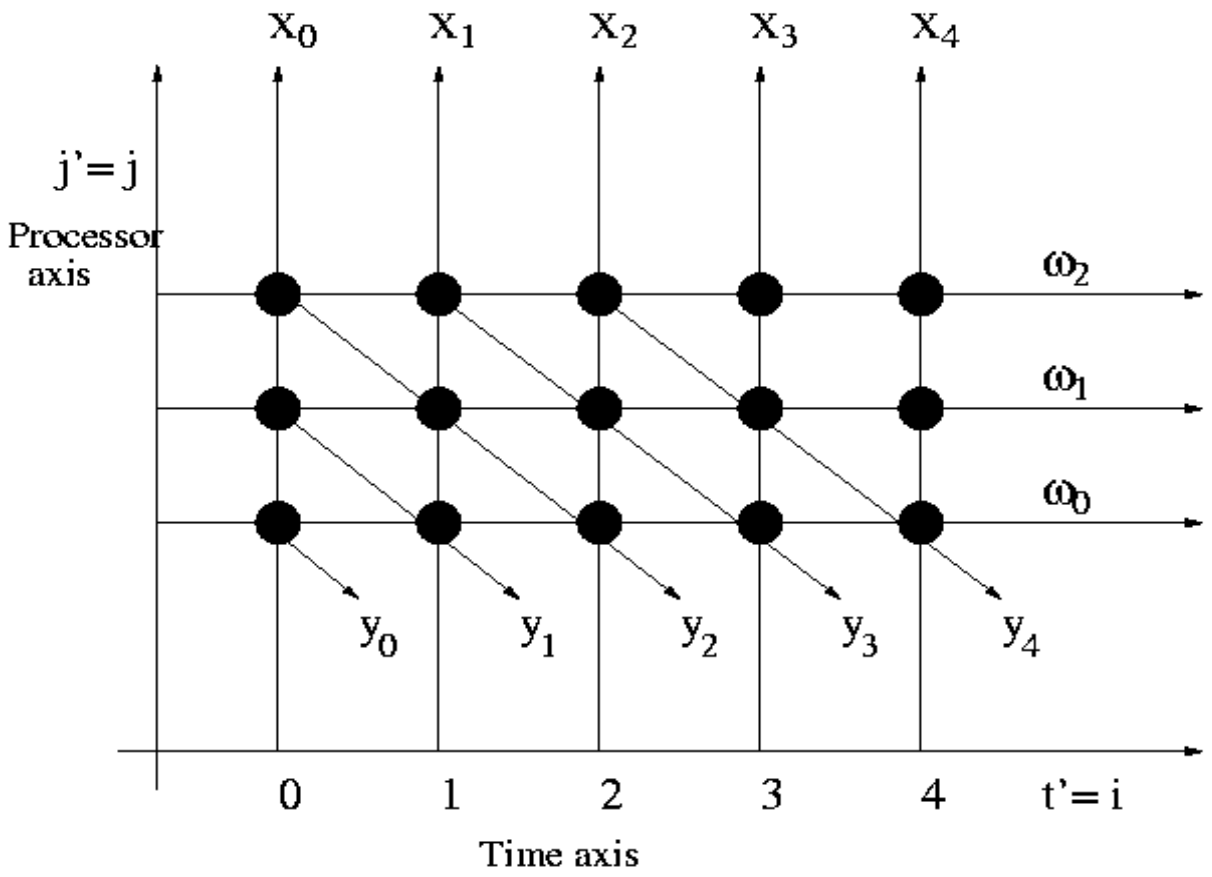
e	$p^T e$	$s^T e$
wt(1 0)	0	1
i/p(0 1)	1	0
result(1 –1)	-1	1



Block diagram of B_1 design



Low-level implementation of B_1 design



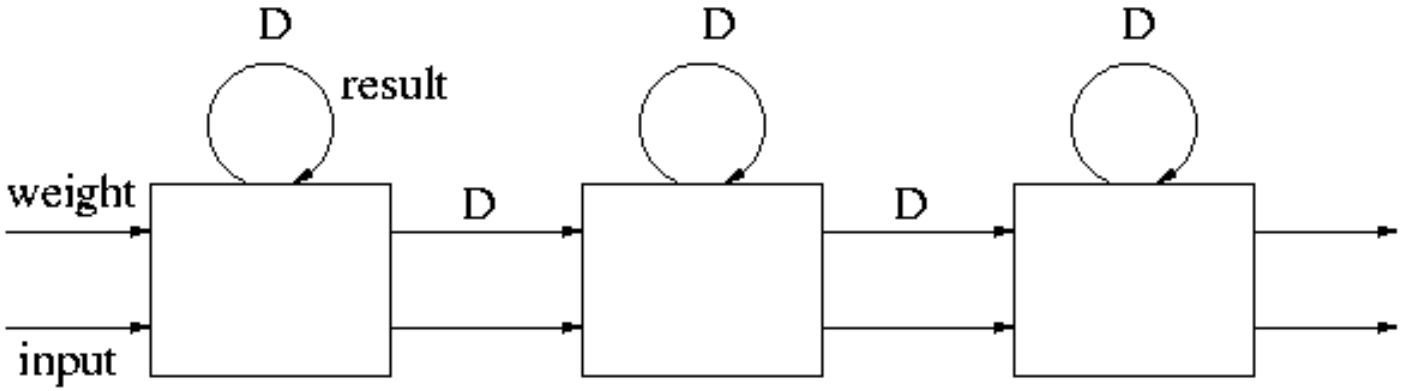
Space-time representation of B_1 design

Design B_2 (Broadcast Inputs, Move Weights, Results Stay)

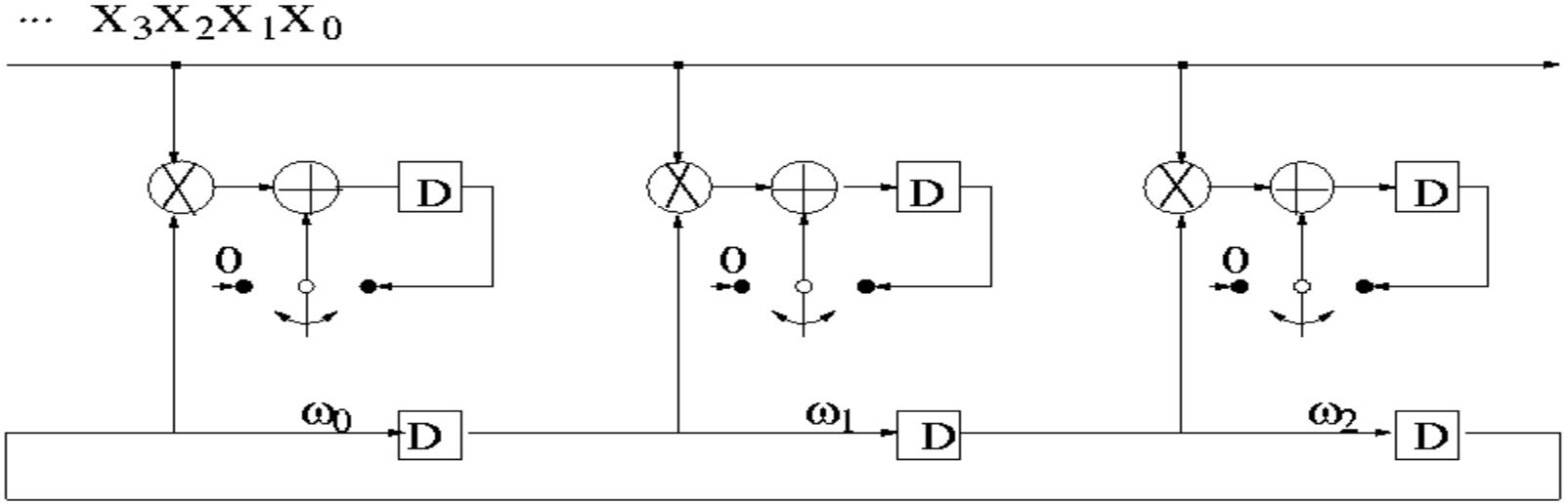
$$d^T = (1\ -1),\ p^T = (1\ 1),\ s^T = (1\ 0)$$

- Any node with index $I^T = (i\ ,\ j)$
  - is mapped to processor $p^T I = i+j$.
  - is executed at time $s^T I = i$.
- Since $s^T d = 1$ we have HUE = $1/|s^T d| = 1$.
- Edge mapping :

e	$p^T e$	$s^T e$
wt(1 0)	1	1
i/p(0 1)	1	0
result(1 –1)	0	1



Block diagram of B_2 design

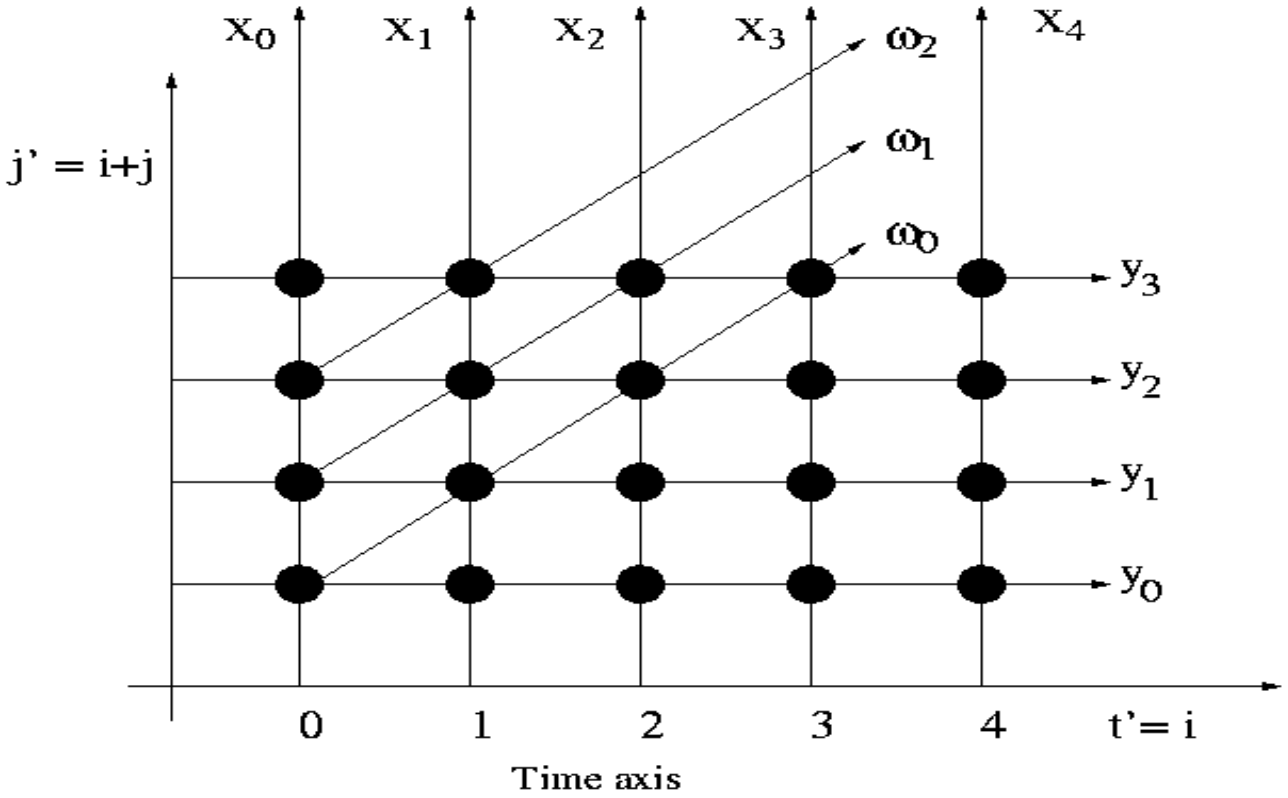


Low-level implementation of B_2 design

- Applying space time transformation we get :

$$j' = p^T(i\ j)^T = i + j$$

$$t' = s^T(i\ j)^T = i$$



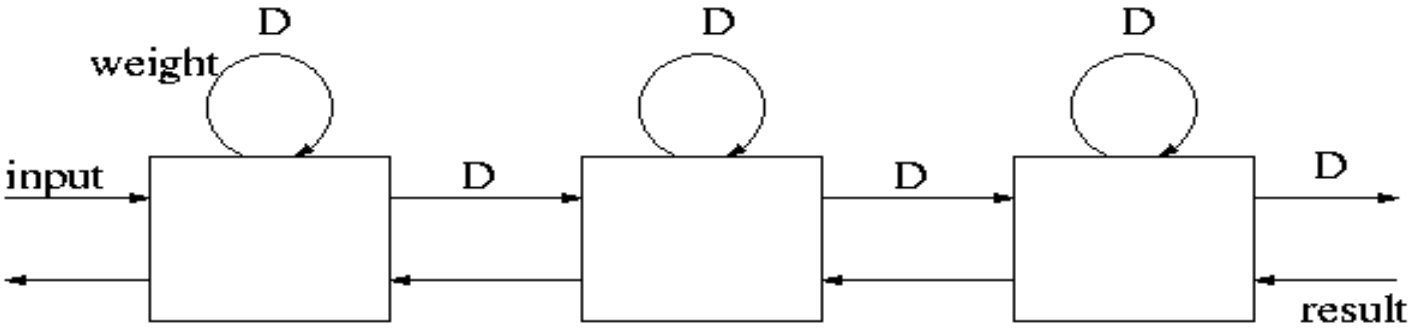
Space-time representation of B_2 design

Design F(Fan-In Results, Move Inputs, Weights Stay)

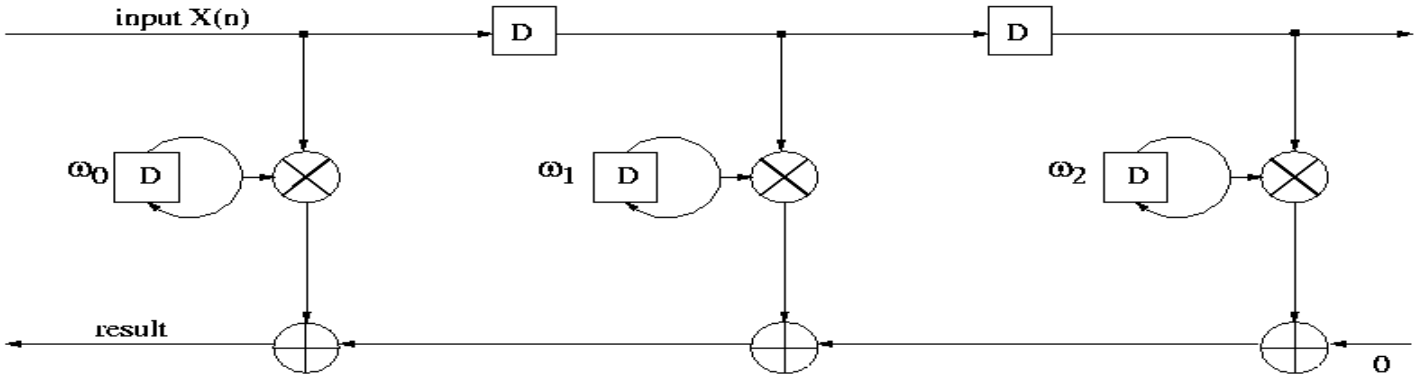
$d^T = (1\ 0)$, $p^T = (0\ 1)$, $s^T = (1\ 1)$

- Since $s^T d = 1$ we have HUE = $1/|s^T d| = 1$.
- Edge mapping :

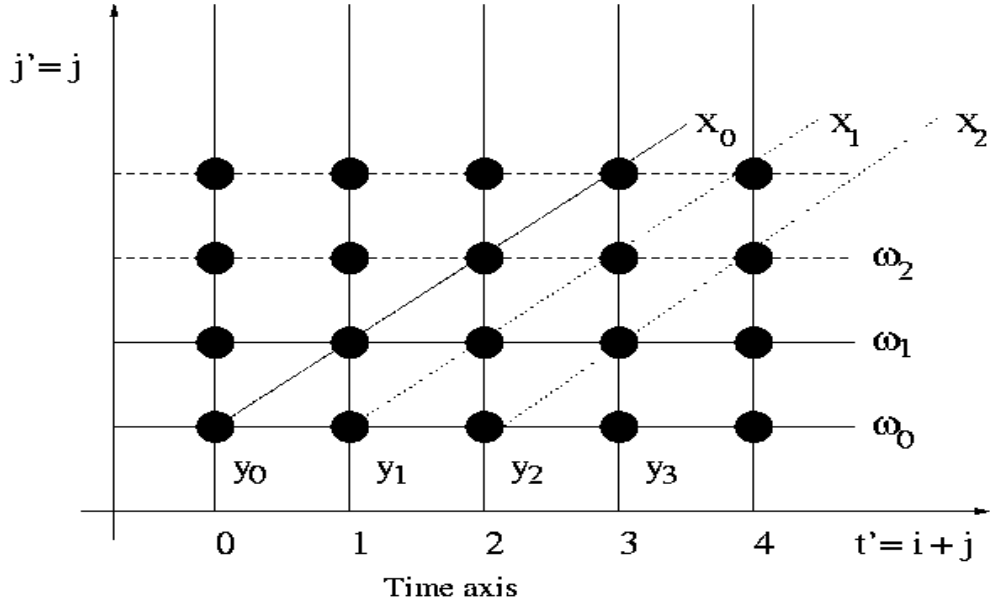
e	$p^T e$	$s^T e$
wt(1 0)	0	1
i/p(0 1)	1	1
result(1 –1)	-1	0



Block diagram of F design



Low-level implementation of F design



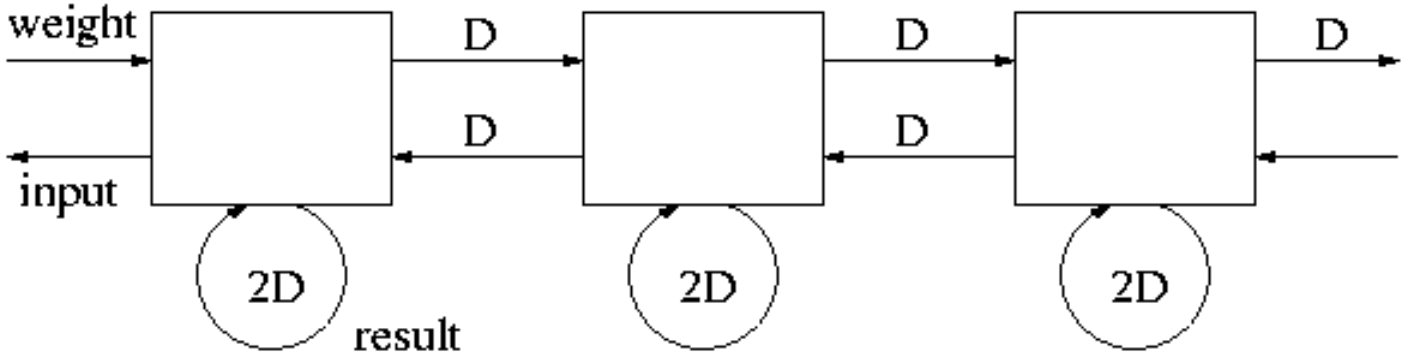
Space-time representation of F design

Design R_1 (Results Stay, Inputs and Weights Move in Opposite Direction)

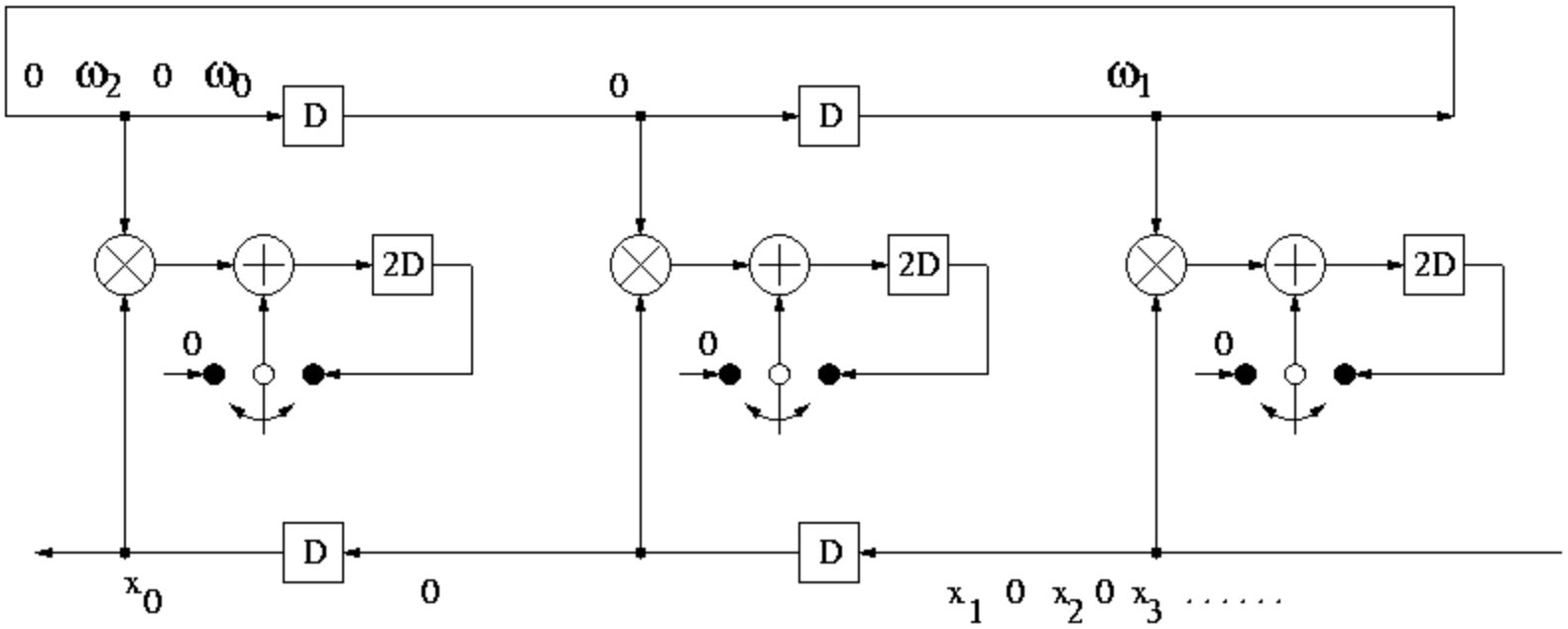
$$d^T = (1\ -1),\ p^T = (1\ 1),\ s^T = (1\ -1)$$

- Since $s^T d = 2$ we have HUE = $1/|s^T d| = ½$.
- Edge mapping :

e	$p^T e$	$s^T e$
wt(1 0)	1	1
i/p(0 -1)	-1	1
result(1 –1)	0	2



Block diagram of R_1 design



Low-level implementation of R_1 design

Note : R_1 can be obtained from B_2 by 2-slow transformation and then retiming after changing the direction of signal x.

Design R_2 and Dual R_2 (Results Stay, Inputs and Weights Move in Same Direction but at Different Speeds)

$d^T = (1\ -1)$, $p^T = (1\ 1)$,

$R_2 : s^T = (2\ 1)$; Dual $R_2 : s^T = (1\ 2)$;

- Since $s^T d = 1$ for both of them we have HUE = $1/|s^T d| = 1$ for both.
- Edge mapping :

R_2			Dual R_2		
e	$p^T e$	$s^T e$	e	$p^T e$	$s^T e$
wt(1, 0)	1	2	wt(1, 0)	1	1
i/p(0,1)	1	1	i/p(0,1)	1	2
result(1, -1)	0	1	result(-1, 1)	0	1

Note : The result edge in design dual R_2 has been reversed to Guarantee $s^T e \geq 0$.

Design W_1 (Weights Stay, Inputs and Results Move in Opposite Directions)

$$d^T = (1\ 0),\ p^T = (0\ 1),\ s^T = (2\ 1)$$

- Since $s^T d = 2$ for both of them we have HUE = $1/|s^T d| = ½$.
- Edge mapping :

e	$p^T e$	$s^T e$
wt(1 0)	0	2
i/p(0 -1)	1	1
result(1 –1)	-1	1

Design W_2 and Dual W_2 (Weights Stay, Inputs and Results Move in Same Direction but at Different Speeds)

$d^T = (1\ 0)$, $p^T = (0\ 1)$,

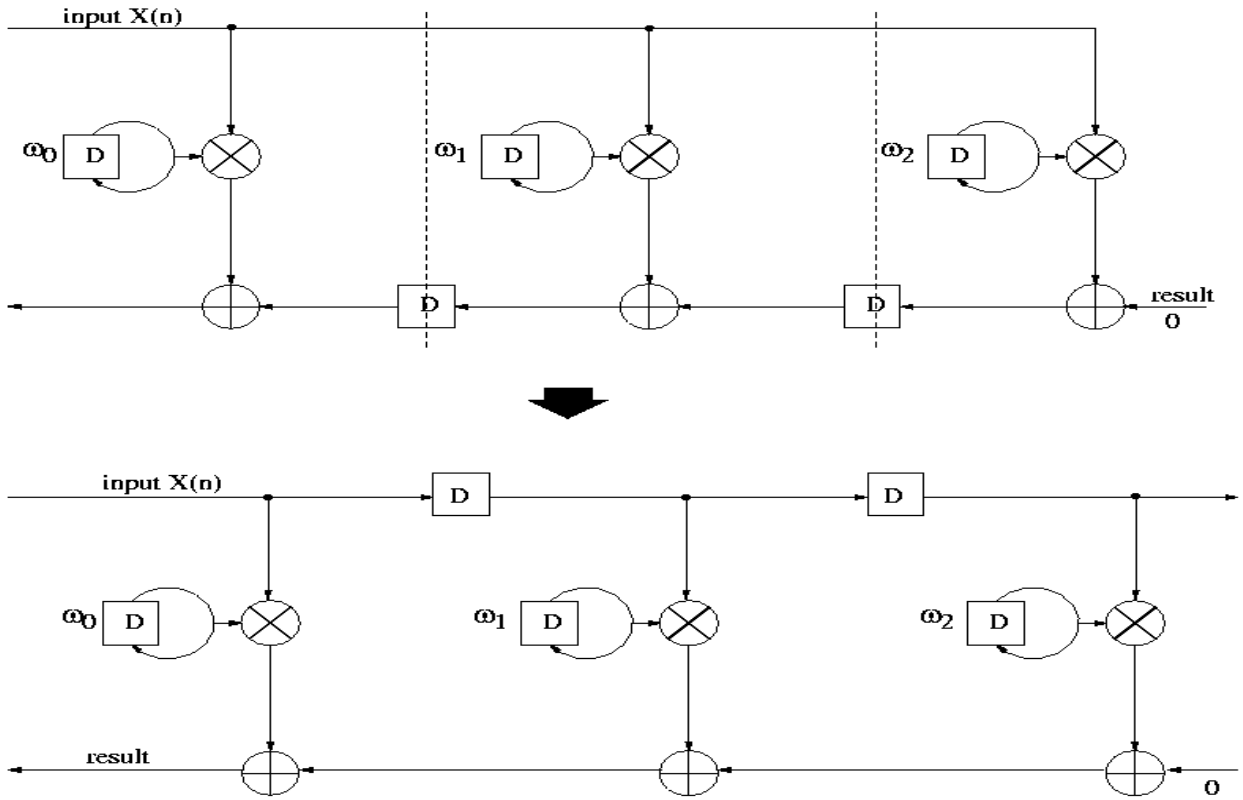
W_2 : $s^T = (1\ 2)$; Dual W_2 : $s^T = (1\ -1)$;

- Since $s^T d = 1$ for both of them we have HUE = $1/|s^T d| = 1$ for both.
- Edge mapping :

W_2			Dual W_2		
e	$p^T e$	$s^T e$	e	$p^T e$	$s^T e$
wt(1, 0)	0	1	wt(1, 0)	0	1
i/p(0,1)	1	2	i/p(0,-1)	-1	1
result(1, -1)	1	1	result(1, -1)	-1	2

- Relating Systolic Designs Using Transformations :
  - FIR systolic architectures obtained using the same projection vector and processor vector, but different scheduling vectors, can be derived from each other by using transformations like edge reversal, associativity, slow-down, retiming and pipelining.
- Example 1 : R_1 can be obtained from B_2 by slow-down, edge reversal and retiming.

- Example 2:



Derivation of design F from B_1 using cutset retiming

- Selection of s^T based on scheduling inequalities:

  For a dependence relation X → Y, where $I_x^T = (i_x, j_x)^T$ and $I_y^T = (i_y, j_y)^T$ are respectively the indices of the nodes X and Y. The scheduling inequality for this dependence is given by,

$$S_y \geq S_x + T_x$$

  where T_x is the computation time of node X. The scheduling equations can be classified into the following two types :

  - Linear scheduling, where

$$S_x = s^T I_x = (s_1\ s_2)(i_x\ \ j_x)^T$$
$$S_y = s^T I_y = (s_1\ s_2)(i_y\ \ j_y)^T$$

  - Affine Scheduling, where

$$S_x = s^T I_x + \gamma_x = (s_1\ s_2)(i_x\ \ j_x)^T + \gamma_x$$
$$S_x = s^T I_x + \gamma_y = (s_1\ s_2)(i_x\ \ j_x)^T + \gamma_y$$

  So scheduling equation for affine scheduling is as follows:

$$s^T I_x + \gamma_y \geq s^T I_x + \gamma_x + T_x$$

Each edge of a DG leads to an inequality for selection of the scheduling vectors which consists of 2 steps.

- Capture all fundamental edges. The reduced dependence graph (RDG) is used to capture the fundamental edges and the regular iterative algorithm (RIA) description of the corresponding problem is used to construct RDGs.
- Construct the scheduling inequalities according to

$$s^T I_x + \gamma_y \geq s^T I_x + \gamma_x + T_x$$

and solve them for feasible s^T.

- RIA Description : The RIA has two forms
  - $\Rightarrow$ The RIA is in standard input RIA form if the index of the inputs are the same for all equations.
  - $\Rightarrow$ The RIA is in standard output RIA form if all the output indices are the same.
- For the FIR filtering example we have,

$$W(i+1, j) = W(i, j)$$

$$X(i, j+1) = X(i, j)$$

$$Y(i+1, j-1) = Y(i, j) + W(i+1, j-1)X(i+1, j-1)$$

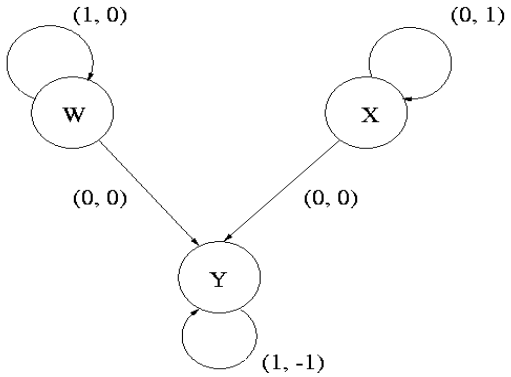
The FIR filtering problem cannot be expressed in standard input RIA form. Expressing it in standard output RIA form we get,

$$W(i, j) = W(i-1, j)$$

$$X(i, j) = X(i, j-1)$$

$$Y(i, j) = Y(i-1, j+1) + W(i, j)X(i, j)$$

- The reduced DG for FIR filtering is shown below.



Example :

$$T_{mult} = 5,\ T_{add} = 2,\ T_{com} = 1$$

Applying the scheduling equations to the five edges of the above figure we get ;

W-->Y : $e = (0\ 0)^T$, $\gamma_x - \gamma_w \geq 0$

X -->X : $e = (0\ 1)^T$, $s_2 + \gamma_x - \gamma_x \geq 1$

W-->W: $e = (1\ 0)^T$, $s_1 + \gamma_w - \gamma_w \geq 1$

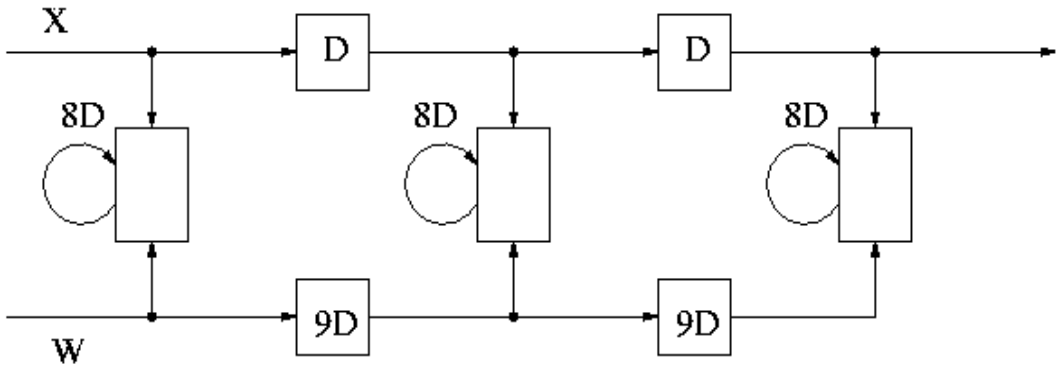
X -->Y : $e = (0\ 0)^T$, $\gamma_y - \gamma_x \geq 0$

Y --> Y: $e = (1\ -1)^T$, $s_1 - s_2 + \gamma_y - \gamma_y \geq 5 + 2 + 1$

For linear scheduling $\gamma_x = \gamma_y = \gamma_w = 0$. Solving we get, $s_1 \geq 1$, $s_2 \geq 1$ and $s_1 - s_2 \geq 8$.

- Taking s^T = (9 1), d = (1 -1) such that $s^T d \neq 0$ and p^T = (1,1) such that $p^T d = 0$ we get HUE = 1/8. The edge mapping is as follows :

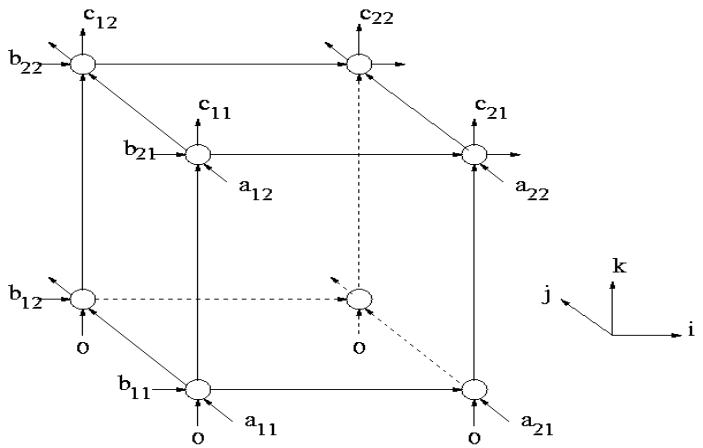
e	$p^T e$	$s^T e$
wt(1 0)	1	9
i/p(0 1)	1	1
result(1 –1)	0	8



Systolic architecture for the example

Matrix-Matrix multiplication and 2-D Systolic Array Design

$$C_{11} = a_{11}b_{11} + a_{12}\, b_{21}$$
$$C_{12} = a_{11}b_{12} + a_{12}\, b_{22}$$
$$C_{21} = a_{21}b_{11} + a_{22}\, b_{21}$$
$$C_{22} = a_{21}b_{12} + a_{22}\, b_{22}$$

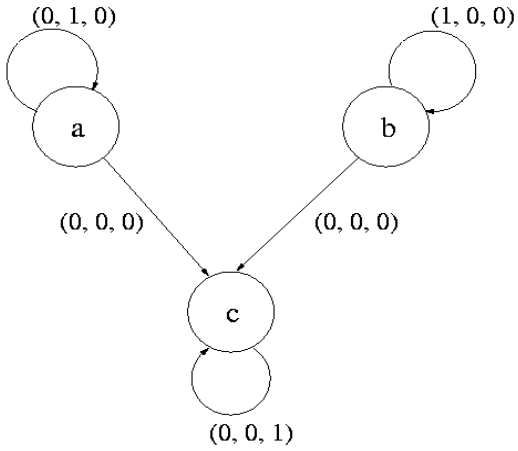


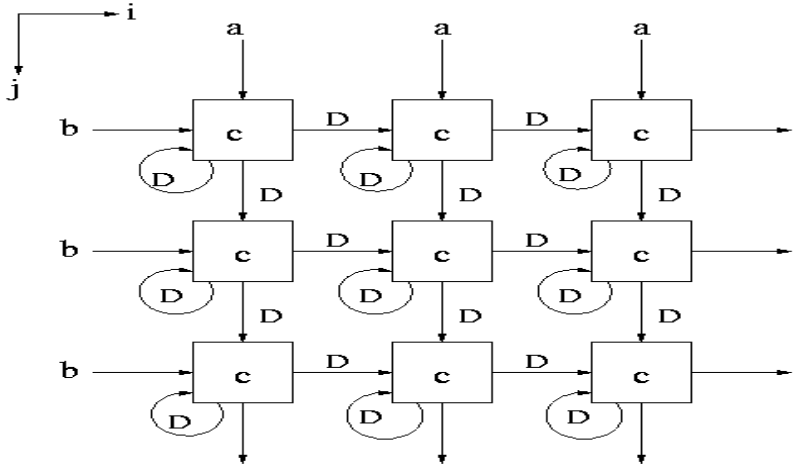
The iteration in standard output RIA form is as follows :

$$a(i,j,k) = a(i,j-1,k)$$
$$b(i,j,k) = b(i-1,j,k)$$
$$c(i,j,k) = c(i,j,k-1) + a(i,j,k)\, b(i,j,k)$$

- Applying scheduling inequality with $T_{mult-add} = 1$, and $T_{com} = 0$ we get $s_2 \geq 0$, $s_1 \geq 0$, $s_3 \geq 1$, $\gamma_c - \gamma_a \geq 0$ and $\gamma_c - \gamma_b \geq 0$. Take $\gamma_a = \gamma_b = \gamma_c = 0$ for linear scheduling.
- Solution 1 :

  $s^T = (1,1,1)$, $d^T = (0,0,1)$, $p_1 = (1,0,0)$, $p_2 = (0,1,0)$, $P^T = (p_1\ p_2)^T$

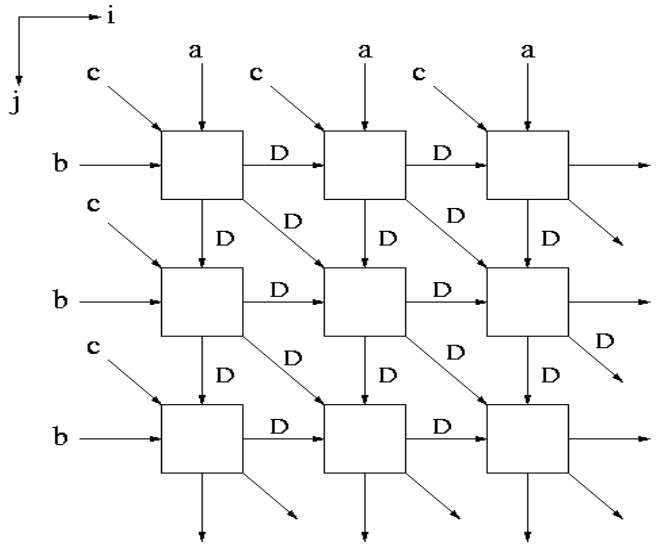




- Solution 2 :

  $s^T = (1,1,1)$, $d^T = (1,1,-1)$, $p_1 = (1,0,1)$,

  $p_2 = (0,1,1)$, $P^T = (p_1\ p_2)^T$



Sol. 1			Sol. 2		
e	$p^T e$	$s^T e$	e	$p^T e$	$s^T e$
a(0, 1, 0)	(0, 1)	1	a(0, 1, 0)	(0, 1)	1
b(1, 0, 0)	(1, 0)	1	b(1, 0, 0)	(1, 0)	1
C(0, 0, 1)	(0, 0)	1	C(0, 0, 1)	(1, 1)	1

Chapter 8: Fast Convolution

Keshab K. Parhi

Chapter 8 Fast Convolution

- Introduction
- Cook-Toom Algorithm and Modified Cook-Toom Algorithm
- Winograd Algorithm and Modified Winograd Algorithm
- Iterated Convolution
- Cyclic Convolution
- Design of Fast Convolution Algorithm by Inspection

Introduction

- **Fast Convolution**: implementation of convolution algorithm using fewer multiplication operations by **algorithmic strength reduction**
- **Algorithmic Strength Reduction:** Number of strong operations (such as multiplication operations) is reduced at the expense of an increase in the number of weak operations (such as addition operations). These are best suited for implementation using either programmable or dedicated hardware
- **Example:** Reducing the multiplication complexity in complex number multiplication:
  - Assume (a+jb)(c+dj)=e+jf, it can be expressed using the matrix form, which requires 4 multiplications and 2 additions:

$$\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix}$$

  - However, the number of multiplications can be reduced to 3 at the expense of 3 extra additions by using:

$$\begin{cases} ac - bd = a(c-d) + d(a-b) \\ ad + bc = b(c+d) + d(a-b) \end{cases}$$

- Rewrite it into matrix form, its coefficient matrix can be decomposed as the product of a 2X3(**C**), a 3X3(**H**)and a 3X2(**D**) matrix:

$$s=\begin{bmatrix} e \\ f \end{bmatrix}=\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}\cdot\begin{bmatrix} c-d & 0 & 0 \\ 0 & c+d & 0 \\ 0 & 0 & d \end{bmatrix}\cdot\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix}\cdot\begin{bmatrix} a \\ b \end{bmatrix}=C\cdot H\cdot D\cdot x$$

  - Where C is a post-addition matrix (requires 2 additions), D is a pre-addition matrix (requires 1 addition), and H is a diagonal matrix (requires 2 additions to get its diagonal elements)
- So, the arithmetic complexity is reduced to 3 multiplications and 3 additions (not including the additions in **H** matrix)

- In this chapter we will discuss two well-known approaches to the design of fast short-length convolution algorithms: the **Cook-Toom algorithm** (based on **Lagrange Interpolation**) and the **Winograd Algorithm** (based on the **Chinese remainder theorem**)

Cook-Toom Algorithm

- A linear convolution algorithm for polynomial multiplication based on the ***Lagrange Interpolation Theorem***
- Lagrange Interpolation Theorem:

Let $b_0, \ldots, b_n$ be a set of $n+1$ distinct points, and let $f(b_i)$, for i = 0, 1, …, n be given. There is exactly one polynomial $f(p)$ of degree n or less that has value $f(b_i)$ when evaluated at b_i for i = 0, 1, …, n. It is given by:

$$f(p) = \sum_{i=0}^{n} f(b_i) \frac{\prod_{j \neq i} (p - b_j)}{\prod_{j \neq i} (b_i - b_j)}$$

- The application of Lagrange interpolation theorem into linear convolution

  Consider an N-point sequence $h = \{h_0, h_1, ..., h_{N-1}\}$ and an L-point sequence $x = \{x_0, x_1, ..., x_{L-1}\}$. The linear convolution of h and x can be expressed in terms of polynomial multiplication as follows: $s(p) = h(p) \cdot x(p)$ where

  $$h(p) = h_{N-1} p^{N-1} + ... + h_1 p + h_0$$

  $$x(p) = x_{L-1} p^{L-1} + ... + x_1 p + x_0$$

  $$s(p) = s_{L+N-2} p^{L+N-2} + ... + s_1 p + s_0$$

  The output polynomial $s(p)$ has degree $L + N - 2$ and has $L + N - 1$ different points.

- (continued)

$s(p)$ can be uniquely determined by its values at $L+N-1$ different points. Let $\{b_0, b_1, ..., b_{L+N-2}\}$ be $L+N-1$ different real numbers. If $s(b_i)$ for $i = \{0,1,..., L+N-2\}$ are known, then $s(p)$ can be computed using the Lagrange interpolation theorem as:

$$s(p) = \sum_{i=0}^{L+N-2} s(b_i) \frac{\prod_{j \neq i} (p - b_j)}{\prod_{j \neq i} (b_i - b_j)}$$

It can be proved that this equation is the unique solution to compute linear convolution for $s(p)$ given the values of $s(b_i)$, for $i = \{0,1,..., L+N-2\}$.

- Cook-Toom Algorithm (Algorithm Description)

1. **Choose** $L+N-1$ **different real numbers** $b_0, b_1, \cdots b_{L+N-2}$
2. **Compute** $h(b_i)$ **and** $x(b_i)$ **, for** $i = \{0,1,\cdots, L+N-2\}$
3. **Compute** $s(b_i) = h(b_i) \cdot x(b_i)$ **, for** $i = \{0,1,\cdots, L+N-2\}$
4. **Compute** $s(p)$ **by using** $$s(p) = \sum_{i=0}^{L+N-2} s(b_i) \frac{\prod_{j \neq i} (p - b_j)}{\prod_{j \neq i} (b_i - b_j)}$$

- Algorithm Complexity
  - The goal of the fast-convolution algorithm is to reduce the multiplication complexity. So, if β_i `s (i=0,1,…,L+N-2) are chosen properly, the computation in step-2 involves some additions and multiplications by small constants
  - The multiplications are only used in step-3 to compute $s(\beta_i)$. So, only L+N-1 multiplications are needed

- By Cook-Toom algorithm, the number of multiplications is reduced from O(LN) to L+N-1 at the expense of an increase in the number of additions
- An adder has much less area and computation time than a multiplier. So, the Cook-Toom algorithm can lead to large savings in hardware (VLSI) complexity and generate computationally efficient implementation

- Example-1: (Example 8.2.1, p.230) Construct a 2X2 convolution algorithm using Cook-Toom algorithm with $\beta=\{0,1,-1\}$
  - Write 2X2 convolution in polynomial multiplication form as s(p)=h(p)x(p), where $h(p) = h_0 + h_1 p \quad x(p) = x_0 + x_1 p$

$$s(p) = s_0 + s_1 p + s_2 p^2$$

  - Direct implementation, which requires 4 multiplications and 1 additions, can be expressed in matrix form as follows:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

- Example-1 (continued)
  - Next we use C-T algorithm to get an efficient convolution implementation with reduced multiplication number

$$b_0 = 0, \quad h(b_0) = h_0, \quad x(b_0) = x_0$$
$$b_1 = 1, \quad h(b_1) = h_0 + h_1, \quad x(b_1) = x_0 + x_1$$
$$b_2 = 2, \quad h(b_2) = h_0 - h_1, \quad x(b_2) = x_0 - x_1$$

  - Then, $s(\beta_0)$, $s(\beta_1)$, and $s(\beta_2)$ are calculated, by using 3 multiplications, as

$$s(b_0) = h(b_0)x(b_0) \quad s(b_1) = h(b_1)x(b_1) \quad s(b_2) = h(b_2)x(b_2)$$

  - From the Lagrange Interpolation theorem, we get:

$$s(p) = s(b_0)\frac{(p-b_1)(p-b_2)}{(b_0-b_1)(b_0-b_2)} + s(b_1)\frac{(p-b_0)(p-b_1)}{(b_1-b_0)(b_1-b_2)}$$
$$+ s(b\ 2)\frac{(p-b_0)(p-b_1)}{(b_2-b_0)(b_2-b_1)}$$
$$= s(b_0) + p\left(\frac{s(b_1)-s(b_2)}{2}\right) + p^2\left(-s(b_0) + \frac{s(b_1)+s(b_2)}{2}\right)$$
$$= s_0 + ps_1 + p^2 s_2$$

- Example-1 (continued)
  - The preceding computation leads to the following matrix form

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} s(b_0) \\ s(b_1)/2 \\ s(b_2)/2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} h(b_0) & 0 & 0 \\ 0 & h(b_1)/2 & 0 \\ 0 & 0 & h(b_2)/2 \end{bmatrix} \cdot \begin{bmatrix} x(b_0) \\ x(b_1) \\ x(b_2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 \\ 0 & (h_0 + h_1)/2 & 0 \\ 0 & 0 & (h_0 - h_1)/2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

  - The computation is carried out as follows **(5 additions, 3 multiplications)**

1. $H_0 = h_0, \quad H_1 = \dfrac{h_0 + h_1}{2}, \quad H_2 = \dfrac{h_0 - h_1}{2}$ **(pre-computed)**
2. $X_0 = x_0, \quad X_1 = x_0 + x_1, \quad X_2 = x_0 - x_1$
3. $S_0 = H_0 X_0, \quad S_1 = H_1 X_1, \quad S_2 = H_2 X_2$
4. $s_0 = S_0, \quad s_1 = S_1 - S_2, \quad s_2 = -S_0 + S_1 + S_2$

- (Continued): Therefore, this algorithm needs 3 multiplications and 5 additions (ignoring the additions in the pre-computation), i.e., the number of multiplications is reduced by 1 at the expense of 4 extra additions
- **Example-2**, please see **Example 8.2.2 of Textbook (p.231)**

- Comments
  - Some additions in the **preaddition** or **postaddition** matrices can be **shared**. So, when we count the number of additions, we only count one instead of two or three.
  - If we take h_0, h_1 as the FIR filter coefficients and take x_0, x_1 as the signal (data) sequence, then the terms **H_0, H_1 need not be recomputed** each time the filter is used. They can be precomputed once offline and stored. So, we **ignore** these computations when counting the number of operations
  - From Example-1, We can understand the Cook-Toom algorithm as a matrix decomposition. In general, a convolution can be expressed in matrix-vector forms as $\begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} h_0 & 0 \\ h_1 & h_0 \\ 0 & h_1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ or $s = T \cdot x$

– Generally, the equation can be expressed as

$$s = T \cdot x = C \cdot H \cdot D \cdot x$$

  - Where C is the postaddition matrix, D is the preaddition matrix, and H is a diagonal matrix with H_i, i = 0, 1, …, L+N-2 on the main diagonal.

– Since **T=CHD**, it implies that the Cook-Toom algorithm provides a way to factorize the convolution matrix T into multiplication of 1 postaddition matrix **C**, 1 diagonal matrix **H** and 1 preaddition matrix **D**, such that the total number of multiplications is determined only by the non-zero elements on the main diagonal of the diagonal matrix **H**

– Although the number of multiplications is reduced, the number of additions has increased. The Cook-Toom algorithm can be modified in order to further reduce the number of additions

Modified Cook-Toom Algorithm

- The Cook-Toom algorithm is used to further reduce the number of addition operations in linear convolutions

  Define $s'(p) = s(p) - S_{L+N-2} p^{L+N-2}$. Notice that the degree of $s(p)$ is $L+N-2$ and S_{L+N-2} is its highest order coefficient. Therefore the degree of $s'(p)$ is $L+N-3$.

- Now consider the modified Cook-Toom Algorithm

- Modified Cook-Toom Algorithm

1. **Choose** $L+N-2$ **different real numbers** $b_0, b_1, \cdots b_{L+N-3}$
2. **Compute** $h(b_i)$ **and** $x(b_i)$ **, for** $i = \{0,1,\cdots, L+N-3\}$
3. **Compute** $s(b_i) = h(b_i) \cdot x(b_i)$ **, for** $i = \{0,1,\cdots, L+N-3\}$
4. **Compute** $s'(b_i) = s(b_i) - s_{L+N-2} b_i^{L+N-2}$ **, for** $i = \{0,1,\cdots, L+N-3\}$
5. **Compute** $s'(p)$ **by using** $$s'(p) = \sum_{i=0}^{L+N-2} s'(b_i) \frac{\prod_{j \neq i} (p - b_j)}{\prod_{j \neq i} (b_i - b_j)}$$
6. **Compute** $s(p) = s'(p) + s_{L+N-2} p^{L+N-2}$

- Example-3 (Example 8.2.3, p.234) Derive a 2X2 convolution algorithm using the modified Cook-Toom algorithm with β={0,-1}

  Consider the Lagrange interpolation for $s'(p) = s(p) - h_1 x_1 p^2$ at $\{b_0 = 0, b_1 = -1\}$.

  First, find $s'(b_i) = h(b_i)x(b_i) - h_1 x_1 b_i^2$

  $$b_0 = 0, \quad h(b_0) = h_0, \quad x(b_0) = x_0$$

  $$b_1 = -1, \quad h(b_1) = h_0 - h_1, \quad x(b_1) = x_0 - x_1$$

  – and

  $$s'(b_0) = h(b_0)x(b_0) - h_1 x_1 b_0^2 = h_0 x_0$$

  $$s'(b_1) = h(b_1)x(b_1) - h_1 x_1 b_1^2 = (h_0 - h_1)(x_0 - x_1) - h_1 x_1$$

  - Which requires 2 multiplications (not counting the $h_1 x_1$ multiplication)

  – Apply the Lagrange interpolation algorithm, we get:

  $$s'(p) = s'(b_0)\frac{(p - b_1)}{(b_0 - b_1)} + s'(b_1)\frac{(p - b_0)}{(b_1 - b_0)}$$

  $$= s'(b_0) + p(s'(b_0) - s'(b_1))$$

- Example-3 (cont'd)
  - Therefore, $s(p) = s'(p) + h_1 x_1 p^2 = s_0 + s_1 p + s_2 p^2$
  - Finally, we have the matrix-form expression:
  - Notice that
$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s'(b_0) \\ s'(b_1) \\ h_1 x_1 \end{bmatrix}$$
  - Therefore:
$$\begin{bmatrix} s'(b_0) \\ s'(b_1) \\ h_1 x_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s(b_0) \\ s(b_1) \\ h_1 x_1 \end{bmatrix}$$
$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s(b_0) \\ s(b_1) \\ h_1 x_1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 \\ 0 & h_0 - h_1 & 0 \\ 0 & 0 & h_1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

- Example-3 (cont'd)
  - The computation is carried out as the follows:
    1. $H_0 = h_0, \quad H_1 = h_0 - h_1, \quad H_2 = h_1$ **(pre-computed)**
    2. $X_0 = x_0, \quad X_1 = x_0 - x_1, \quad X_2 = x_1$
    3. $S_0 = H_0 X_0, \quad S_1 = H_1 X_1, \quad S_2 = H_2 X_2$
    4. $s_0 = S_0, \quad s_1 = S_0 - S_1 + S_2, \quad s_2 = S_2$
  - The total number of operations are 3 multiplications and 3 additions. Compared with the convolution algorithm in Example-1, the number of addition operations has been reduced by 2 while the number of multiplications remains the same.
- **Example-4 (Example 8.2.4, p. 236 of Textbook)**
- **Conclusion:** The Cook-Toom Algorithm is efficient as measured by the number of multiplications. However, as the size of the problem increases, it is not efficient because the number of additions increases greatly if β takes values other than $\{0, \pm 1, \pm 2, \pm 4\}$. This may result in complicated pre-addition and post-addition matrices. For large-size problems, the Winograd algorithm is more efficient.

Winograd Algorithm

- The Winograd short convolution algorithm: based on the CRT (Chinese Remainder Theorem) ---It's possible to uniquely determine a nonnegative integer given only its remainder with respect to the given moduli, provided that the moduli are relatively prime and the integer is known to be smaller than the product of the moduli
- Theorem: CRT for Integers

Given $c_i = R_{m_i}[c]$ **(represents the remainder when** c **is divided by** m_i **), for** $i = 0,1,\ldots,k$, **where** m_i **are moduli and are relatively prime, then**

$$c = \left(\sum_{i=0}^{k} c_i N_i M_i \right) \bmod M$$

, **where** $M = \prod_{i=0}^{k} m_i$, $M_i = M / m_i$, **and** N_i **is the solution of** $N_i M_i + n_i m_i = GCD(M_i, m_i) = 1$, **provided that** $0 \le c < M$

- **Theorem: CRT for Polynomials**

  Given $c^{(i)}(p) = R_{m^{(i)}}(p)[c(p)]$, for i=0, 1, …,k, where $m^{(i)}(p)$ are relatively prime, then $c(p) = \left(\sum_{i=0}^{k} c^{(i)}(p) N^{(i)}(p) M^{(i)}(p) \right) \bmod M(p)$, where $M(p) = \prod_{i=0}^{k} m^{(i)}(p)$, $M^{(i)}(p) = M(p) / m^{(i)}(p)$, and $N^{(i)}(p)$ is the solution of $N^{(i)}(p) M^{(i)}(p) + n^{(i)}(p) m^{(i)}(p) = GCD(M^{(i)}(p), m^{(i)}(p)) = 1$ Provided that the degree of $c(p)$ is less than the degree of $M(p)$

- **Example-5** (Example 8.3.1, p.239)**: using the CRT for integer**, Choose moduli m_0=3, m_1=4, m_2=5. Then $M = m_0 m_1 m_2 = 60$, and $M_i = M / m_i$. Then:

$$m_0 = 3, \quad M_0 = 20, \quad (-1)20 + 7(3) = 1$$
$$m_1 = 4, \quad M_1 = 15, \quad (-1)15 + (4)4 = 1$$
$$m_2 = 5, \quad M_2 = 12, \quad (-2)12 + (5)5 = 1$$

  – where N_i and n_i are obtained using the Euclidean GCD algorithm. Given that the integer c satisfying $0 \le c < M$, let $c_i = R_{m_i}[c]$.

- **Example-5 (cont'd)**
  - The integer c can be calculated as

$$c = \left(\sum_{i=0}^{k} c_i N_i M_i \right) \bmod M = (-20 * c_0 - 15 * c_1 - 24 * c_2) \bmod 60$$

  - For c=17, $c_0 = R_3(17) = 2, \quad c_1 = R_4(17) = 1, \quad c_2 = R_5(17) = 2$

$$c = (-20 * 2 - 15 * 1 - 24 * 2) \bmod 60 = (-103) \bmod 60 = 17$$

- **CRT for polynomials:** The remainder of a polynomial with regard to modulus $p^i + f(p)$, where $\deg(f(p)) \le i - 1$, can be evaluated by substituting p^i by $-f(p)$ in the polynomial
- **Example-6** (Example 8.3.2, pp239)

$$(a). \quad R_{x+2}\left[5x^2 + 3x + 5\right] = 5(-2)^2 + 3(-2) + 5 = 19$$

$$(b). \quad R_{x^2+2}\left[5x^2 + 3x + 5\right] = 5(-2) + 3x + 5 = 3x - 5$$

$$(c). \quad R_{x^2+x+2}\left[5x^2 + 3x + 5\right] = 5(-x - 2) + 3x + 5 = -2x - 5$$

- Winograd Algorithm
  - 1. Choose a polynomial $m(p)$ with degree higher than the degree of $h(p)x(p)$ and factor it into k+1 relatively prime polynomials with real coefficients, i.e., $m(p) = m^{(0)}(p)m^{(1)}(p)\cdots m^{(k)}(p)$
  - 2. Let $M^{(i)}(p) = m(p)/m^{(i)}(p)$. Use the Euclidean GCD algorithm to solve $N^{(i)}(p)M^{(i)}(p) + n^{(i)}(p)m^{(i)}(p) = 1$ for $N^{(i)}(p)$.
  - 3. Compute: $h^{(i)}(p) = h(p) \bmod m^{(i)}(p), \qquad x^{(i)}(p) = x(p) \bmod m^{(i)}(p)$ $for \quad i = 0,1,\cdots,k$
  - 4. Compute: $s^{(i)}(p) = h^{(i)}(p)x^{(i)}(p) \bmod m^{(i)}(p), \qquad for \quad i = 0,1,\cdots,k$
  - 5. Compute $s(p)$ by using:

$$s(p) = \sum_{i=0}^{k} s^{(i)}(p)N^{(i)}(p)M^{(i)}(p) \bmod m^{(i)}(p)$$

- Example-7 (Example 8.3.3, p.240) Consider a 2X3 linear convolution as in Example 8.2.2. Construct an efficient realization using Winograd algorithm with $m(p) = p(p-1)(p^2+1)$
  - Let: $m^{(0)}(p) = p, \quad m^{(1)}(p) = p-1, \quad m^{(2)}(p) = p^2+1$
  - Construct the following table using the relationships $M^{(i)}(p) = m(p)/m^{(i)}(p)$ and $N^{(i)}(p)M^{(i)}(p) + n^{(i)}(p)m^{(i)}(p) = 1 \quad for \quad i = 0,1,2$

i	$m^{(i)}(p)$	$M^{(i)}(p)$	$n^{(i)}(p)$	$N^{(i)}(p)$
0	p	$p^3 - p^2 + p - 1$	$p^2 - p + 1$	-1
1	$p-1$	$p^3 + p$	$-\frac{1}{2}\left(p^2 + p + 2\right)$	$\frac{1}{2}$
2	p^2+1	$p^2 - p$	$-\frac{1}{2}(p-2)$	$\frac{1}{2}(p-1)$

  - Compute residues from $h(p) = h_0 + h_1 p, \quad x(p) = x_0 + x_1 p + x_2 p^2$:

$$h^{(0)}(p) = h_0, \qquad x^{(0)}(p) = x_0$$
$$h^{(1)}(p) = h_0 + h_1, \qquad x^{(1)}(p) = x_0 + x_1 + x_2$$
$$h^{(2)}(p) = h_0 + h_1 p, \quad x_{(2)}(p) = (x_0 - x_2) + x_1 p$$

- Example-7 (cont'd)

$$s^{(0)}(p) = h_0 x_0 = s_0^{(0)}, \quad s^{(1)}(p) = (h_0 + h_1)(x_0 + x_1 + x_2) = s_0^{(1)}$$

$$s^{(2)}(p) = (h_0 + h_1 p)((x_0 - x_2) + x_1 p) \bmod (p^2 + 1)$$

$$= h_0(x_0 - x_2) - h_1 x_1 + (h_0 x_1 + h_1(x_0 - x_2))\, p = s_0^{(2)} + s_1^{(2)} p$$

  - Notice, we need 1 multiplication for $s^{(0)}(p)$, 1 for $s^{(1)}(p)$, and 4 for $s^{(2)}(p)$
  - However it can be further reduced to 3 multiplications as shown below:

$$\begin{bmatrix} s_0^{(2)} \\ s_1^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 \\ 0 & h_0 - h_1 & 0 \\ 0 & 0 & h_0 + h_1 \end{bmatrix} \cdot \begin{bmatrix} x_0 + x_1 - x_2 \\ x_0 - x_2 \\ x_1 \end{bmatrix}$$

  - Then:

$$s(p) = \sum_{i=0}^{2} s^{(i)}(p) N^{(i)}(p) M^{(i)}(p) \bmod m^{(i)}(p)$$

$$= \left[-s^{(0)}(p)(p^3 - p^2 + p - 1) + \frac{S^{(1)}(p)}{2}(p^3 + p) + \frac{S^{(2)}(p)}{2}(p^3 - 2p^2 + p) \right]$$

$$\bmod (p^4 - p^3 + p^2 - p)$$

- Example-7 (cont'd)
  - Substitute $s^{(0)}(p)$, $s^{(1)}(p)$, $s^{(2)}(p)$ into $s(p)$ to obtain the following table

p^0	p^1	p^2	p^3
$s_0^{(0)}$	$-s_0^{(0)}$	$s_0^{(0)}$	$-s_0^{(0)}$
0	$\frac{1}{2}s_0^{(1)}$	0	$\frac{1}{2}s_0^{(1)}$
0	$\frac{1}{2}s_0^{(2)}$	$-s_0^{(2)}$	$\frac{1}{2}s_0^{(2)}$
0	$\frac{1}{2}s_1^{(2)}$	0	$-\frac{1}{2}s_1^{(2)}$

  - Therefore, we have

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 \\ 1 & 0 & -2 & 0 \\ -1 & 1 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} s_0^{(0)} \\ \frac{1}{2}s_0^{(1)} \\ \frac{1}{2}s_0^{(2)} \\ \frac{1}{2}s_1^{(2)} \end{bmatrix}$$

- Example-7 (cont'd)
  - Notice that

$$\begin{bmatrix} s_0^{(0)} \\ \frac{1}{2} s_0^{(1)} \\ \frac{1}{2} s_0^{(2)} \\ \frac{1}{2} s_1^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 & 0 & 0 \\ 0 & \frac{h_0+h_1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{h_0}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{h_1-h_0}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{h_0+h_1}{2} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_0 + x_1 + x_2 \\ x_0 + x_1 - x_2 \\ x_0 - x_2 \\ x_1 \end{bmatrix}$$

  - So, finally we have:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 2 & 1 & -1 \\ 1 & 0 & -2 & 0 & 2 \\ -1 & 0 & 0 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 & 0 & 0 \\ 0 & \frac{h_0+h_1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{h_0}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{h_1-h_0}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{h_0+h_1}{2} \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

- Example-7 (cont'd)
  - In this example, the Winograd convolution algorithm requires 5 multiplications and 11 additions compared with 6 multiplications and 2 additions for direct implementation
- Notes:
  - The number of multiplications in Winograd algorithm is highly dependent on the degree of each $m^{(i)}(p)$. Therefore, the degree of m(p) should be as small as possible.
  - More efficient form (or a modified version) of the Winograd algorithm can be obtained by letting deg[m(p)]=deg[s(p)] and applying the CRT to

$$s'(p) = s(p) - h_{N-1} x_{L-1} m(p)$$

Modified Winograd Algorithm

- 1. Choose a polynomial $m(p)$ with degree equal to the degree of $s(p)$ and factor it into k+1 relatively prime polynomials with real coefficients, i.e., $m(p) = m^{(0)}(p)m^{(1)}(p)\cdots m^{(k)}(p)$
- 2. Let $M^{(i)}(p) = m(p)/m^{(i)}(p)$, use the Euclidean GCD algorithm to solve $N^{(i)}(p)M^{(i)}(p) + n^{(i)}(p)m^{(i)}(p) = 1$ for $N^{(i)}(p)$.
- 3. Compute: $h^{(i)}(p) = h(p) \bmod m^{(i)}(p), \qquad x^{(i)}(p) = x(p) \bmod m^{(i)}(p)$ $for \quad i = 0,1,\cdots,k$
- 4. Compute: $s'^{(i)}(p) = h^{(i)}(p)x^{(i)}(p) \bmod m^{(i)}(p), \qquad for \quad i = 0,1,\cdots,k$
- 5. Compute $s'(p)$ by using:
$$s'(p) = \sum_{i=0}^{k} s'^{(i)}(p)N^{(i)}(p)M^{(i)}(p) \bmod m^{(i)}(p)$$
- 6. Compute $s(p) = s'(p) + h_{N-1}x_{L-1}m(p)$

- Example-8 (Example 8.3.4, p.243): Construct a 2X3 convolution algorithm using modified Winograd algorithm with m(p)=p(p-1)(p+1)
  - Let $m^{(0)}(p) = p, \quad m^{(1)}(p) = p-1, \quad m^{(2)}(p) = p+1$
  - Construct the following table using the relationships $M^{(i)}(p) = m(p)/m^{(i)}(p)$ and $N^{(i)}(p)M^{(i)}(p) + n^{(i)}(p)m^{(i)}(p) = 1$

i	$m^{(i)}(p)$	$M^{(i)}(p)$	$n^{(i)}(p)$	$N^{(i)}(p)$
0	p	p^2-1	p	-1
1	$p-1$	p^2+p	$-\frac{1}{2}(p+2)$	$\frac{1}{2}$
2	$p+1$	p^2-p	$-\frac{1}{2}(p-2)$	$\frac{1}{2}$

  - Compute residues from $h(p) = h_0 + h_1 p, \quad x(p) = x_0 + x_1 p + x_2 p^2$:

$$h^{(0)}(p) = h_0, \qquad x^{(0)}(p) = x_0$$
$$h^{(1)}(p) = h_0 + h_1, \quad x^{(1)}(p) = x_0 + x_1 + x_2$$
$$h^{(2)}(p) = h_0 - h_1, \quad x_{(2)}(p) = x_0 - x_1 + x_2$$
$$s'^{(0)}(p) = h_0 x_0, \quad s'^{(1)}(p) = (h_0 + h_1)(x_0 + x_1 + x_2),$$
$$s'^{(2)}(p) = (h_0 - h_1)(x_0 - x_1 + x_2)$$

- Example-8 (cont'd)
  - Since the degree of $m^{(i)}(p)$ is equal to 1, $s'^{(i)}(p)$ is a polynomial of degree 0 (a constant). Therefore, we have:

$$\begin{aligned} s(p) &= s'(p) + h_1 x_2 m(p) \\ &= \left[-s'^{(0)}(-p^2+1) + \frac{s'^{(1)}}{2}(p^2+p) + \frac{s'^{(2)}}{2}(p^2-p) + h_1 x_2 (p^3-p) \right] \\ &= s'^{(0)} + p\left(\frac{s'(1)}{2} - \frac{s'(2)}{2} - h_1 x_2\right) + p^2\left(-s'^{(0)} + \frac{s'^{(1)}}{2} + \frac{s'^{(2)}}{2}\right) + p^3(h_1 x_2) \end{aligned}$$

  - The algorithm can be written in matrix form as:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s'^{(0)} \\ \frac{s'^{(1)}}{2} \\ \frac{s'^{(2)}}{2} \\ h_1 x_2 \end{bmatrix}$$

- Example-8 (cont'd)
  - (matrix form)

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} h_0 & 0 & 0 & 0 \\ 0 & \frac{h_0 + h_1}{2} & 0 & 0 \\ 0 & 0 & \frac{h_0 - h_1}{2} & 0 \\ 0 & 0 & 0 & h_1 \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

  - Conclusion: this algorithm requires 4 multiplications and 7 additions

Iterated Convolution

- I**terated convolution algorithm:** makes use of efficient short-length convolution algorithms **iteratively** to build long convolutions
- Does not achieve minimal multiplication complexity, but achieves a **good balance** between multiplications and addition complexity
- Iterated Convolution Algorithm (Description)
  - 1. Decompose the long convolution into several levels of short convolutions
  - 2. Construct fast convolution algorithms for short convolutions
  - 3. Use the short convolution algorithms to iteratively (hierarchically) implement the long convolution
  - **Note**: the order of short convolutions in the decomposition affects the complexity of the derived long convolution

- **Example-9** (Example 8.4.1, pp.245): Construct a 4X4 linear convolution algorithm using 2X2 short convolution
  - Let $h(p) = h_0 + h_1 p + h_2 p^2 + h_3 p^3, \quad x(p) = x_0 + x_1 p + x_2 p^2 + x_3 p^3$ and $s(p) = h(p)x(p)$
  - First, we need to decompose the 4X4 convolution into a 2X2 convolution
  - Define $h'_0(p) = h_0 + h_1 p, \quad h'_1(p) = h_2 + h_3 p$

    $x'_0(p) = x_0 + x_1 p, \quad x'_1(p) = x_2 + x_3 p$
  - Then, we have:

$$\Rightarrow \quad h(p) = h'_0(p) + h'_1(p)p^2, \quad i.e., h(p) = h(p,q) = h'_0(p) + h'_1(p)q$$
$$x(p) = x'_0(p) + x'_1(p)p^2, \quad i.e., x(p) = x(p,q) = x'_0(p) + x'_1(p)q$$

$$s(p) = h(p)x(p) = h(p,q)x(p,q)$$
$$\Rightarrow \quad = [h'_0(p) + h'_1(p)q] \cdot [x'_0(p) + x'_1(p)q]$$
$$= h'_0(p)x'_0(p) + [h'_0(p)x'_1(p) + h'_1(p)x'_0(p)]q + h'_1(p)x'_1(p)q^2$$
$$= s'_0(p) + s'_1(p)q + s'_2(p)q^2 = s(p,q)$$

- Example-9 (cont'd)
  - Therefore, the 4X4 convolution is decomposed into two levels of nested 2X2 convolutions
  - Let us start from the first convolution $s'_0(p) = h'_0(p) \cdot x'_0(p)$, we have:

$$h'_0(p) \cdot x'_0(p) \equiv h'_0 \cdot x'_0 = (h_0 + h_1 p) \cdot (x_0 + x_1 p)$$
$$= h_0 x_0 + h_1 x_1 p^2 + p[(h_0 + h_1) \cdot (x_0 + x_1) - h_0 x_0 - h_1 x_1]$$

  - We have the following expression for the third convolution:

$$s'_2(p) = h'_1(p) \cdot x'_1(p) \equiv h'_1 \cdot x'_1 = (h_2 + h_3 p) \cdot (x_2 + x_3 p)$$
$$= h_2 x_2 + h_3 x_3 p^2 + p[(h_2 + h_3) \cdot (x_2 + x_3) - h_2 x_2 - h_3 x_3]$$

  - For the second convolution, we get the following expression:

$$s'_1(p) = h'_0(p) \cdot x'_1(p) + h'_1(p) \cdot x'_0(p) \equiv h'_0 \cdot x'_1 + h'_1 \cdot x'_0$$
$$= [(h'_0 + h'_1) \cdot (x'_0 + x'_1) - h'_0 \cdot x'_0 - h'_1 \cdot x'_1]$$

—: multiplication ▼: addition

- Example-9 (Cont'd)
  - For $[(h'_0+h'_1)\cdot(x'_0+x'_1)]$, we have the following expression:

$$(h'_0+h'_1)\cdot(x'_0+x'_1)=[(h_0+h_2)+p(h_1+h_3)]\cdot[(x_0+x_2)+p(x_1+x_3)]$$
$$=(h_0+h_2)\cdot(x_0+x_2)+p^2(h_1+h_3)\cdot(x_1+x_3)$$
$$+p[(h_0+h_1+h_2+h_3)\cdot(x_0+x_1+x_2+x_3)$$
$$-(h_0+h_2)\cdot(x_0+x_2)-(h_1+h_3)\cdot(x_1+x_3)]$$

This requires 9 multiplications and 11 additions

  - If we rewrite the three convolutions as the following expressions, then we can get the following table (see the next page):

$$h'_0\,x'_0\equiv a_1+pa_2+p^2a_3$$
$$h'_1\,x'_1\equiv b_1+pb_2+p^2b_3$$
$$(h'_0+h'_1)\cdot(x'_0+x'_1)\equiv c_1+pc_2+p^2c_3$$

- Example-9 (cont'd)

p^0	p^1	p^2	p^3	p^4	p^5	p^6
a_1	a_2	a_3		b_1	b_2	b_3
		c_1	c_2	c_3		
		$-b_1$	$-b_2$	$-b_3$		
		$-a_1$	$-a_2$	$-a_3$		

Total 8 additions here

  - Therefore, the total number of operations used in this 4X4 iterated convolution algorithm is 9 multiplications and 19 additions

Cyclic Convolution

- Cyclic convolution: also known as circular convolution
- Let the filter coefficients be $h = \{h_0, h_1, \cdots, h_{n-1}\}$, and the data sequence be $x = \{x_0, x_1, \cdots, x_{n-1}\}$.
  - The cyclic convolution can be expressed as
    $$s(p) = h \mathrm{O}_n x = [h(p) \cdot x(p)] \bmod (p^n - 1)$$
  - The output samples are given by
    $$s_i = \sum_{k=0}^{n-1} h_{((i-k))} x_k, \qquad i = 0,1,\cdots,n-1$$
    - where $((i-k))$ denotes $(i-k) \bmod n$
- The cyclic convolution can be computed as a linear convolution reduced by modulo $p^n - 1$. (Notice that there are 2n-1 different output samples for this linear convolution). Alternatively, the cyclic convolution can be computed using CRT with $m(p) = p^n - 1$, which is much simpler.

- Example-10 (Example 8.5.1, p.246) Construct a 4X4 cyclic convolution algorithm using CRT with $m(p) = p^4 - 1 = (p-1)(p+1)(p^2+1)$
  - Let $h(p) = h_0 + h_1 p + h_2 p^2 + h_3 p^3, \quad x(p) = x_0 + x_1 p + x_2 p^2 + x_3 p^3$
  - Let $m^{(0)}(p) = p - 1, \quad m^{(1)}(p) = p + 1, \quad m^{(2)}(p) = p^2 + 1$
  - Get the following table using the relationships $M^{(i)}(p) = m(p) / m^{(i)}(p)$ and $N^{(i)}(p) M^{(i)}(p) + n^{(i)}(p) m^{(i)}(p) = 1$

i	$m^{(i)}(p)$	$M^{(i)}(p)$	$n^{(i)}(p)$	$N^{(i)}(p)$
0	$p-1$	p^3+p^2+p-1	$-\frac{1}{4}(p^2+2p+3)$	$\frac{1}{4}$
1	$p+1$	p^3-p^2+p-1	$\frac{1}{4}\left(p^2-2p+3\right)$	$-\frac{1}{4}$
2	p^2+1	p^2-1	$\frac{1}{2}$	$-\frac{1}{2}$

  - Compute the residues

$$h^{(0)}(p) = h_0 + h_1 + h_2 + h_3 = h_0^{(0)},$$
$$h^{(1)}(p) = h_0 - h_1 + h_2 - h_3 = h_0^{(1)},$$
$$h^{(2)}(p) = (h_0 - h_2) + (h_1 - h_3)p = h_0^{(2)} + h_1^{(2)} p$$

- Example-10 (cont'd) —: multiplication

$$x^{(0)}(p) = x_0 + x_1 + x_2 + x_3 = x_0^{(0)},$$
$$x^{(1)}(p) = x_0 - x_1 + x_2 - x_3 = x_0^{(1)},$$
$$x^{(2)}(p) = (x_0 - x_2) + (x_1 - x_3)p = x_0^{(2)} + x_1^{(2)} p$$

$\Rightarrow$

$$s^{(0)}(p) = h^{(0)}(p) \cdot x^{(0)}(p) = h_0^{(0)} \cdot x_0^{(0)} = s_0^{(0)},$$
$$s^{(1)}(p) = h^{(1)}(p) \cdot x^{(1)}(p) = h_0^{(1)} \cdot x_0^{(1)} = s_0^{(1)},$$
$$s^{(2)}(p) = s_0^{(2)} + s_1^{(2)} p = [h^{(2)}(p) \cdot x^{(2)}(p)] \bmod (p^2 + 1)$$
$$= \left(h_0^{(2)} \cdot x_0^{(2)} - h_1^{(2)} \cdot x_1^{(2)}\right) + p\left(h_0^{(2)} x_1^{(2)} + h_1^{(2)} x_0^{(2)}\right)$$

  - Since

$$s_0^{(2)} = h_0^{(2)} x_0^{(2)} - h_1^{(2)} x_1^{(2)} = h_0^{(2)}\left(x_0^{(2)} + x_1^{(2)}\right) - \left(h_0^{(2)} + h_1^{(2)}\right) x_1^{(2)},$$
$$s_1^{(2)} = h_0^{(2)} x_1^{(2)} + h_1^{(2)} x_0^{(2)} = h_0^{(2)}\left(x_0^{(2)} + x_1^{(2)}\right) + \left(h_1^{(2)} - h_0^{(2)}\right) x_0^{(2)},$$

  - or in matrix-form

$$\begin{bmatrix} s_0^{(2)} \\ s_1^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} h_0^{(2)} & 0 & 0 \\ 0 & h_1^{(2)} - h_0^{(2)} & 0 \\ 0 & 0 & h_0^{(2)} + h_1^{(2)} \end{bmatrix} \cdot \begin{bmatrix} x_0^{(2)} + x_1^{(2)} \\ x_0^{(2)} \\ x_1^{(2)} \end{bmatrix}$$

  - Computations so far require 5 multiplications

- Example-10 (cont'd)
  - Then

$$s(p) = \sum_{i=0}^{2} s^{(i)}(p) N^{(i)}(p) M^{(i)}(p) \bmod m^{(i)}(p)$$

$$= \left[s_0^{(0)} \left(\frac{p^3+p^2+p+1}{4} \right) + s_0^{(1)} \left(\frac{p^3-p^2+p-1}{-4} \right) + s_0^{(2)} \left(\frac{p^2-1}{-2} \right) \right] + s_1^{(2)} \left(p \cdot \frac{p^2-1}{-2} \right)$$

$$= \left(\frac{s_0^{(0)}}{4} + \frac{s_0^{(1)}}{4} + \frac{s_0^{(2)}}{2} \right) + p \left(\frac{s_0^{(0)}}{4} - \frac{s_0^{(1)}}{4} + \frac{s_1^{(2)}}{2} \right) + p^2 \left(\frac{s_0^{(0)}}{4} + \frac{s_0^{(1)}}{4} - \frac{s_0^{(2)}}{2} \right)$$

$$+ p^3 \left(\frac{1}{4} s_0^{(0)} - \frac{1}{4} s_0^{(1)} - \frac{1}{2} s_1^{(2)} \right)$$

  - So, we have

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{4} s_0^{(0)} \\ \frac{1}{4} s_0^{(1)} \\ \frac{1}{2} s_0^{(2)} \\ \frac{1}{2} s_1^{(2)} \end{bmatrix}$$

- Example-10 (cont'd)
  - Notice that:

$$\begin{bmatrix} \frac{1}{4}s_0^{(0)} \\ \frac{1}{4}s_0^{(1)} \\ \frac{1}{2}s_0^{(2)} \\ \frac{1}{2}s_1^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \cdot$$

$$\begin{bmatrix} \frac{1}{4}h_0^{(0)} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4}h_0^{(1)} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2}h_0^{(2)} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}\left(h_1^{(2)} - h_0^{(2)}\right) & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}\left(h_0^{(2)} - h_1^{(2)}\right) \end{bmatrix} \cdot \begin{bmatrix} x_0^{(0)} \\ x_0^{(1)} \\ x_0^{(2)} + x_1^{(2)} \\ x_0^{(2)} \\ x_1^{(2)} \end{bmatrix}$$

- Example-10 (cont'd)
  - Therefore, we have

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 \\ 1 & -1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 & 1 \\ 1 & -1 & -1 & -1 & 0 \end{bmatrix} \cdot$$

$$\begin{bmatrix} \frac{h_0+h_1+h_2+h_3}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{h_0-h_1+h_2-h_3}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{h_0-h_2}{2} & 0 & 0 \\ 0 & 0 & 0 & \frac{-h_0+h_1+h_2-h_3}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{h_0+h_1-h_2-h_3}{2} \end{bmatrix} \cdot$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- Example-10 (cont'd)
  - This algorithm requires 5 multiplications and 15 additions
  - The direct implementation requires 16 multiplications and 12 additions (see the following matrix-form. Notice that the cyclic convolution matrix is a circulant matrix)

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} h_0 & h_3 & h_2 & h_1 \\ h_1 & h_0 & h_3 & h_2 \\ h_2 & h_1 & h_0 & h_3 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- An efficient cyclic convolution algorithm can often be easily extended to construct efficient linear convolution

- Example-11 (Example 8.5.2, p.249) Construct a 3X3 linear convolution using 4X4 cyclic convolution algorithm

- Example-11 (cont'd)
  - Let the 3-point coefficient sequence be $h=\{h_0,h_1,h_2\}$, and the 3-point data sequence be $x=\{x_0,x_1,x_2\}$
  - First extend them to 4-point sequences as:

$$h=\{h_0,h_1,h_2,0\},\quad x=\{x_0,x_1,x_2,0\}$$

  - Then the 3X3 linear convolution of h and x is

$$\Rightarrow h\cdot x=\begin{bmatrix} h_0x_0 \\ h_1x_0+h_0x_1 \\ h_2x_0+h_1x_1+h_0x_2 \\ h_2x_1+h_1x_2 \\ h_2x_2 \end{bmatrix} \qquad h\mathrm{O}_4x=\begin{bmatrix} h_0x_0+h_2x_2 \\ h_1x_0+h_0x_1 \\ h_2x_0+h_1x_1+h_0x_2 \\ h_2x_1+h_1x_2 \end{bmatrix}$$

  - The 4X4 cyclic convolution of h and x, i.e. $h\mathrm{O}_4x$, is:

- Example-11 (cont'd)
  - Therefore, we have $s(p) = h(p) \cdot x(p) = h\mathrm{O}_n x + h_2 x_2 (p^4 - 1)$
  - Using the result of Example-10 for $h\mathrm{O}_4 x$, the following convolution algorithm for 3X3 linear convolution is obtained:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & -1 & -1 \\ 1 & -1 & 1 & 1 & 0 & 0 \\ 1 & 1 & -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot$$

(continued on the next page)

- Example-11 (cont'd)

$$\cdot \begin{bmatrix} \frac{h_0+h_1+h_2}{4} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{h_0-h_1+h_2}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{h_0-h_2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-h_0+h_1+h_2}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{h_0+h_1-h_2}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & h_2 \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

- Example-11 (cont'd)
  - So, this algorithm requires 6 multiplications and 16 additions

- Comments:
  - In general, an efficient linear convolution can be used to obtain an efficient cyclic convolution algorithm. Conversely, an efficient cyclic convolution algorithm can be used to derive an efficient linear convolution algorithm

Design of fast convolution algorithm by inspection

- When the Cook-Toom or the Winograd algorithms can not generate an efficient algorithm, sometimes a clever factorization by inspection may generate a better algorithm
- Example-12 (Example 8.6.1, p.250) Construct a 3X3 fast convolution algorithm by inspection
  - The 3X3 linear convolution can be written as follows, which requires 9 multiplications and 4 additions

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} h_0 x_0 \\ h_1 x_0 + h_0 x_1 \\ h_2 x_0 + h_1 x_1 + h_0 x_2 \\ h_2 x_1 + h_1 x_2 \\ h_2 x_2 \end{bmatrix}$$

- Example-12 (cont'd)
  - Using the following identities:

$$s_1 = h_1 x_0 + h_0 \cdot x_1 = (h_0 + h_1) \cdot (x_0 + x_1) - h_0 x_0 - h_1 x_1$$
$$s_2 = h_2 x_0 + h_1 x_1 + h_0 x_2 = (h_0 + h_2)(x_0 + x_2) - h_0 x_0 + h_1 x_1 - h_2 x_2$$
$$s_3 = h_2 x_1 + h_1 x_2 = (h_1 + h_2)(x_1 + x_2) - h_1 x_1 - h_2 x_2$$

  - The 3X3 linear convolution can be written as:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 \\ -1 & 1 & -1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \cdot$$

(continued on the next page)

- Example-12 (cont'd)

$$\cdot \begin{bmatrix} h_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & h_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & h_0 + h_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_0 + h_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & h_1 + h_2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

  - Conclusion: This algorithm, which can not be obtained by using the Cook-Toom or the Winograd algorithms, requires 6 multiplications and 10 additions

Chapter 9: Algorithmic Strength Reduction in Filters and Transforms

Keshab K. Parhi

Outline

- Introduction
- Parallel FIR Filters
  - Formulation of Parallel FIR Filter Using Polyphase Decomposition
  - Fast FIR Filter Algorithms
- Discrete Cosine Transform and Inverse DCT
  - Algorithm-Architecture Transformation
  - Decimation-in-Frequency Fast DCT for 2^M-point DCT

Introduction

- Strength reduction leads to a reduction in hardware complexity by exploiting substructure sharing and leads to less silicon area or power consumption in a VLSI ASIC implementation or less iteration period in a programmable DSP implementation
- Strength reduction enables design of parallel FIR filters with a less-than-linear increase in hardware
- DCT is widely used in video compression. Algorithm-architecture transformations and the decimation-in-frequency approach are used to design fast DCT architectures with significantly less number of multiplication operations

Parallel FIR Filters

Formulation of Parallel FIR Filters Using Polyphase Decomposition

- An N-tap FIR filter can be expressed in time-domain as

$$y(n) = h(n) * x(n) = \sum_{i=0}^{N-1} h(i)x(n-i), \quad n = 0,1,2,\cdots,\infty$$

  - where {x(n)} is an infinite length input sequence and the sequence $\{h(n)\}$ contains the FIR filter coefficients of length N
  - In Z-domain, it can be written as

$$Y(z) = H(z) \cdot X(z) = \left(\sum_{n=0}^{N-1} h(n)z^{-n} \right) \cdot \left(\sum_{n=0}^{\infty} x(n)z^{-n} \right)$$

- The Z-transform of the sequence x(n) can be expressed as:

$$\begin{aligned} X(z) &= x(0)+x(1)z^{-1}+x(2)z^{-2}+x(3)z^{-3}+\cdots \\ &= \left[x(0)+x(2)z^{-2}+x(4)z^{-4}+\cdots\right]+z^{-1}\left[x(1)+x(3)z^{-2}+x(5)z^{-4}+\cdots\right] \\ &= X_0(z^2)+z^{-1}X_1(z^2) \end{aligned}$$

  - where $X_0(z^2)$ and $X_1(z^2)$, the two polyphase components, are the z-transforms of the even time series $\{x(2k)\}$ and the odd time-series $\{x(2k+1)\}$, for $\{0 \le k < \infty\}$, respectively
- Similarly, the length-N filter coefficients H(z) can be decomposed as:

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2)$$

  - where $H_0(z^2)$ and $H_1(z^2)$ are of length N/2 and are referred as even and odd sub-filters, respectively
- The even-numbered output sequence {y(2k)} and the odd-numbered output sequence {y(2k+1)} for $\{0 \le k < \infty\}$ can be computed as

(continued on the next page)

- (cont'd)

$$\begin{aligned} Y(z) &= Y_0(z^2) + z^{-1}Y_1(z^2) \\ &= \left(X_0(z^2) + z^{-1}X_1(z^2)\right)\cdot\left(H_0(z^2) + z^{-1}H_1(z^2)\right) \\ &= X_0(z^2)H_0(z^2) + z^{-1}\left[X_0(z^2)H_1(z^2) + X_1(z^2)H_0(z^2)\right] \\ &\quad + z^{-2}\left[X_1(z^2)H_1(z^2)\right] \end{aligned}$$

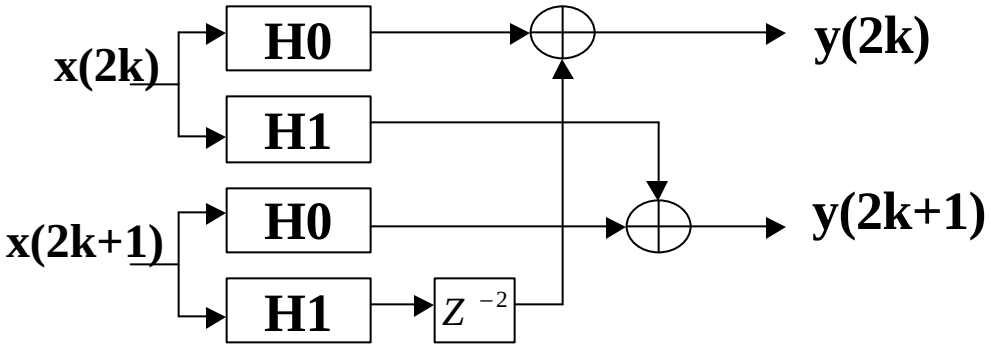
  - i.e.,

$$\begin{aligned} Y_0(z^2) &= X_0(z^2)H_0(z^2) + z^{-2}X_1(z^2)H_1(z^2) \\ Y_1(z^2) &= X_0(z^2)H_1(z^2) + X_1(z^2)H_0(z^2) \end{aligned}$$

  - where $Y_0(z^2)$ and $Y_1(z^2)$ correspond to y(2k) and y(2k+1) in time domain, respectively. This 2-parallel filter processes 2 inputs x(2k) and x(2k+1) and generates 2 outputs y(2k) and y(2k+1) every iteration. It can be written in matrix-form as:

$$Y = H\cdot X \quad \textbf{or} \quad \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-2}H_1 \\ H_1 & H_0 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \qquad (9.1)$$

– The following figure shows the traditional 2-parallel FIR filter structure, which requires 2N multiplications and 2(N-1) additions



- For 3-phase poly-phase decomposition, the input sequence X(z) and the filter coefficients H(z) can be decomposed as follows

$$X(z) = X_0(z^3) + z^{-1}X_1(z^3) + z^{-2}X_2(z^3),$$

$$H(z) = H_0(z^3) + z^{-1}H_1(z^3) + z^{-2}H_2(z^3)$$

– where $\{X_0(z^3), X_1(z^3), X_2(z^3)\}$ correspond to x(3k),x(3k+1) and x(3k+2) in time domain, respectively; and $\{H_0(z^3), H_1(z^3), H_2(z^3)\}$ are the three sub-filters of H(z) with length N/3.

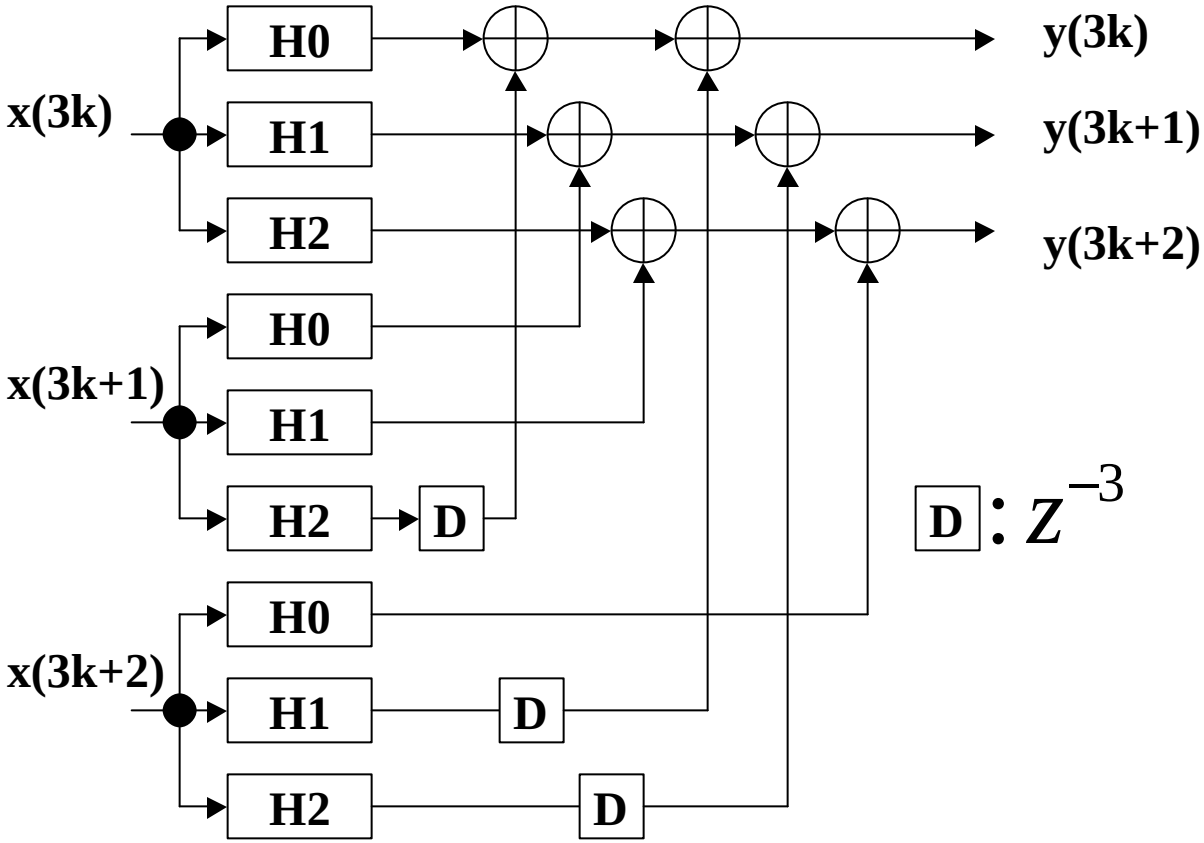
– The output can be computed as:

$$\begin{aligned} Y(z) &= Y_0(z^3) + z^{-1}Y_1(z^3) + z^{-2}Y_2(z^3) \\ &= \left(X_0 + z^{-1}X_1 + z^{-2}X_2\right)\cdot\left(H_0 + z^{-1}H_1 + z^{-2}H_2\right) \\ &= \left[X_0H_0 + z^{-3}\left(X_1H_2 + X_2H_1\right)\right] + z^{-1}\left[X_0H_1 + X_1H_0 + z^{-3}X_2H_2\right] \\ &\quad + z^{-2}\left[X_0H_2 + X_1H_1 + X_2H_0\right] \end{aligned}$$

– In every iteration, this 3-parallel FIR filter processes 3 input samples x(3k), x(3k+1) and x(3k+2), and generates 3 outputs y(3k), y(3k+1) and y(3k+2), and can be expressed in matrix form as:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-3}H_2 & z^{-3}H_1 \\ H_1 & H_0 & z^{-3}H_2 \\ H_2 & H_1 & H_0 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} \qquad (9.2)$$

– The following figure shows the traditional 3-parallel FIR filter structure, which requires 3N multiplications and 3(N-1) additions



- **Generalization:**
  - The outputs of an L-Parallel FIR filter can be computed as:

$$Y_k = z^{-L}\left(\sum_{i=k+1}^{L-1} H_i X_{L+k-i}\right) + \left(\sum_{i=0}^{k} H_i x_{k-i}\right), \quad 0 \le k \le L-2$$
$$Y_{L-1} = \sum_{i=0}^{L-1} H_i X_{L-1-i} \qquad (9.3)$$

  - This can also be expressed in Matrix form as

$$Y = H \cdot X$$
$$\begin{bmatrix} Y_0 \\ Y_1 \\ \cdots \\ Y_{L-1} \end{bmatrix} = \begin{bmatrix} H_0 & z^{-L}H_{L-1} & \cdots & z^{-L}H_1 \\ H_1 & H_0 & \cdots & z^{-L}H_2 \\ \cdots & \cdots & \cdots & \cdots \\ H_{L-1} & H_{L-2} & \cdots & H_0 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \\ \cdots \\ X_{L-1} \end{bmatrix} \qquad (9.4)$$

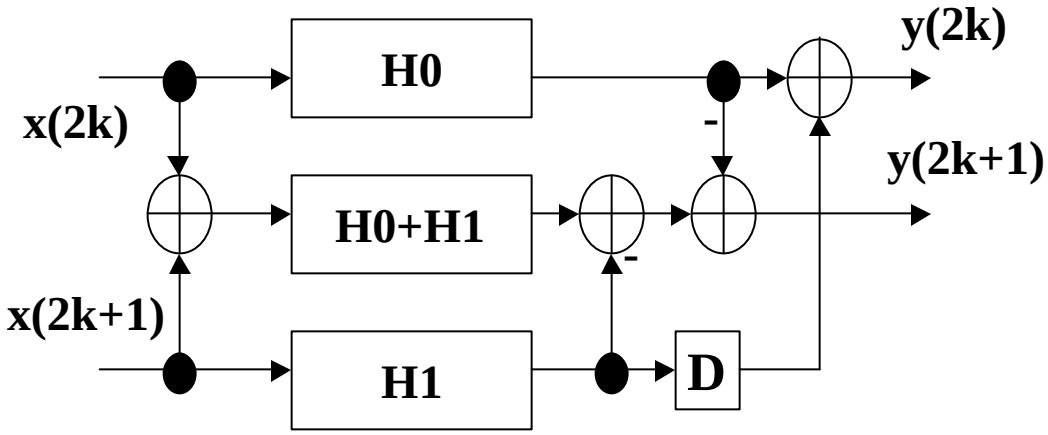
Note: H is a pseudo-circulant matrix

Two-parallel and Three-parallel Low-Complexity FIR Filters

- **Two-parallel Fast FIR Filter**
  - The 2-parallel FIR filter can be rewritten as

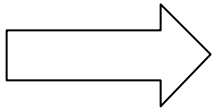
$$Y_0 = H_0 X_0 + z^{-2} H_1 X_1 \qquad (9.5)$$
$$Y_1 = (H_0 + H_1)\cdot(X_0 + X_1) - H_0 X_0 - H_1 X_1$$

  - This 2-parallel fast FIR filter contains 3 sub-filters. The 2 sub-filters H_0X_0 and H_1X_1 are shared for the computation of Y_0 and Y_1



- This 2-parallel filter requires 3 distinct sub-filters of length N/2 and 4 pre/post-processing additions. It requires 3N/2 = 1.5N multiplications and 3(N/2-1)+4=1.5N+1 additions. [The traditional 2-parallel filter requires 2N multiplications and 2(N-1) additions]
- Example-1: when N=8 and $H=\{h_0,h_1,\cdots,h_6,h_7\}$, the 3 sub-filters are

$$H_0=\{h_0,h_2,h_4,h_6\}$$



$$H_1=\{h_1,h_3,h_5,h_7\}$$

$$H_0+H_1=\{h_0+h_1,h_2+h_3,h_4+h_5,h_6+h_7\}$$

- The subfilter H_0+H_1 can be precomputed
- The 2-parallel filter can also be written in matrix form as

$$Y_2=Q_2\cdot H_2\cdot P_2\cdot X_2 \qquad (9.6)$$

Q_2 is a post-processing matrix which determines the manner in which the filter outputs are combined to correctly produce the parallel outputs and P_2 is a pre-processing matrix which determines the manner in which the inputs should be combined

– **(matrix form)**

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & z^{-2} \\ -1 & 1 & -1 \end{bmatrix} \cdot diag\left(\begin{matrix} H_0 \\ H_0 + H_1 \\ H_1 \end{matrix} \right) \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \quad (9.7)$$

– where diag(h*) represents an NXN diagonal matrix H_2 with diagonal elements h*.

– **Note:** the application of FFA diagonalizes the original pseudo-circulant matrix H. The entries on the diagonal of H_2 are the sub-filters required in this parallel FIR filter

– Many different equivalent parallel FIR filter structures can be obtained. For example, this 2-parallel filter can be implemented using sub-filters $\{H_0, H_0 - H_1, H_1\}$ which may be more attractive in narrow-band low-pass filters since the sub-filter $H_0 - H_1$ requires fewer non-zero bits than $H_0 + H_1$. The parallel structure containing $H_0 + H_1$ is more attractive for narrow-band high-pass filters.

- **3-Parallel Fast FIR Filter**
  - A fast 3-parallel FIR algorithm can be derived by recursively applying a 2-parallel fast FIR algorithm and is given by

$$Y_0 = H_0X_0 - z^{-3}H_2X_2 + z^{-3}\left[(H_1+H_2)(X_1+X_2) - H_1X_1\right]$$

$$Y_1 = \left[(H_0+H_1)(X_0+X_1) - H_1X_1\right] - \left[H_0X_0 - z^{-3}H_2X_2\right] \qquad (9.8)$$

$$Y_2 = \left[(H_0+H_1+H_2)(X_0+X_1+X_2)\right] - \left[(H_0+H_1)(X_0+X_1) - H_1X_1\right] - \left[(H_1+H_2)(X_1+X_2) - H_1X_1\right]$$

  - The 3-parallel FIR filter is constructed using 6 sub-filters of length N/3, including H_0X_0, H_1X_1, H_2X_2, $(H_0+H_1)(X_0+X_1)$, $(H_1+H_2)(X_1+X_2)$ and $(H_0+H_1+H_2)(X_0+X_1+X_2)$
  - With 3 pre-processing and 7 post-processing additions, this filter requires 2N multiplications and 2N+4 additions which is 33% less than a traditional 3-parallel filter

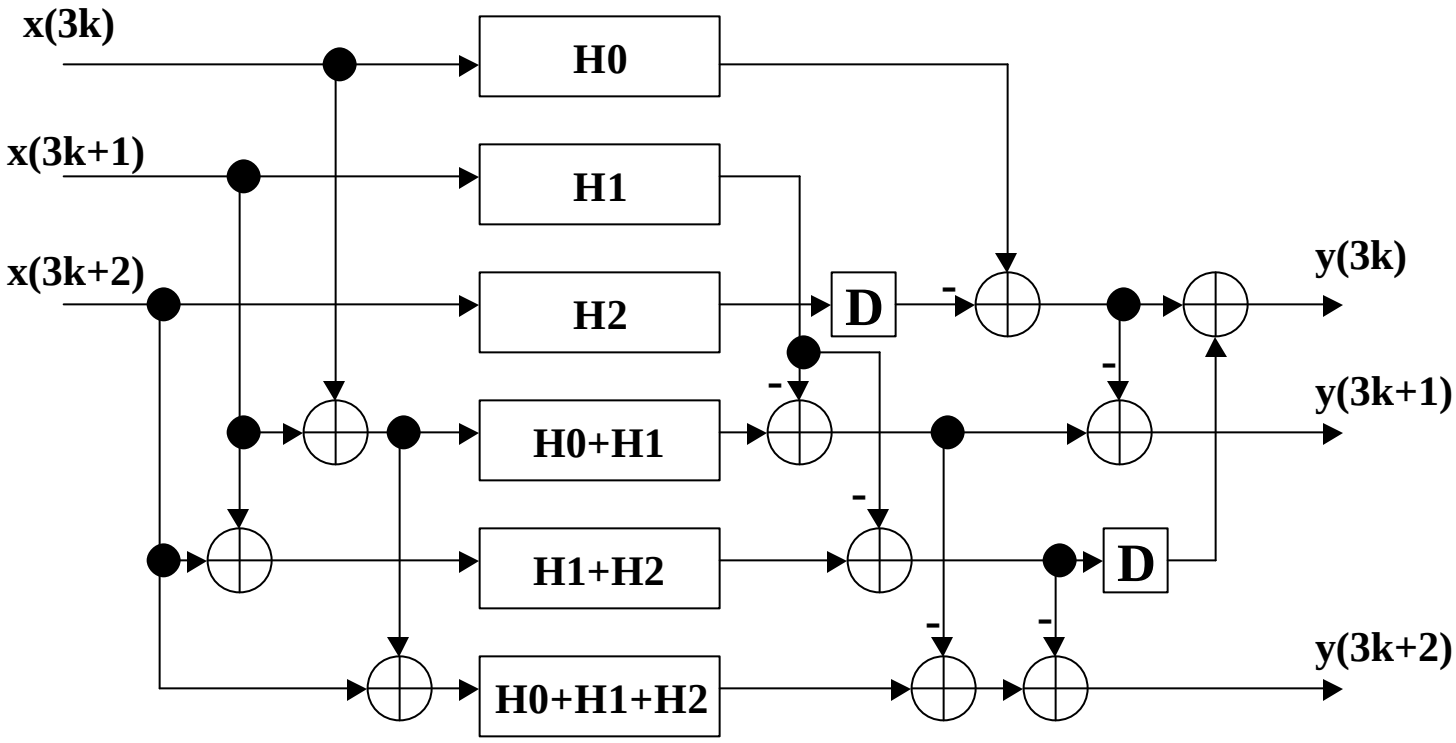
– The 3-parallel filter can be expressed in matrix form as

$$Y_3 = Q_3 \cdot H_3 \cdot P_3 \cdot X_3$$

$$Y_3 = \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix}, \quad Q_3 = \begin{bmatrix} 1 & 0 & z^{-3} & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -z^{-3} & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_3 = diag\begin{bmatrix} H_0 \\ H_1 \\ H_2 \\ H_0 + H_1 \\ H_1 + H_2 \\ H_0 + H_1 + H_2 \end{bmatrix}, \quad P_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad X_3 = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} \qquad (9.9)$$

– Reduced-complexity 3-parallel FIR filter structure



Parallel FIR Filters (cont'd)

Parallel Filters by Transposition

- Any parallel FIR filter structure can be used to derive another parallel equivalent structure by transpose operation (or transposition). Generally, the transposed architecture has the same hardware complexity, but different finite word-length performance
- Consider the L-parallel filter in matrix form Y=HX (9.4), where H is an LXL matrix. An equivalent realization of this parallel filter can be generated by taking the transpose of the H matrix and flipping the vectors X and Y:

$$Y_F = H^T \cdot X_F$$

  - where $\begin{cases} X_F = [X_{L-1} \quad X_{L-2} \quad \cdots \quad X_0]^T \\ Y_F = [Y_{L-1} \quad Y_{L-2} \quad \cdots \quad Y_0]^T \end{cases}$ (9.10)

- Examples:
  - the 2-parallel FIR filter in (9.1) can be reformulated by using transposition as follows:

$$\begin{bmatrix} Y_1 \\ Y_0 \end{bmatrix} = \begin{bmatrix} H_0 & H_1 \\ z^{-2}H_1 & H_0 \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_0 \end{bmatrix}$$

  - Transposition of the 2-parallel fast filter in (9.6) leads to another equivalent structure:

$$Y_2 = Q_2 \cdot H_2 \cdot P_2 \cdot X_2 \Longrightarrow \begin{aligned} Y_{2_F} &= \left(Q_2 \cdot H_2 \cdot P_2\right)^T \cdot X_{2_F} \\ &= P_2^T \cdot H_2^T \cdot Q_2^T \cdot X_{2_F} \end{aligned}$$

$$\begin{bmatrix} Y_1 \\ Y_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \cdot diag\begin{pmatrix} H_0 \\ H_0 + H_1 \\ H_1 \end{pmatrix} \cdot \begin{bmatrix} 1 & -1 \\ 0 & 1 \\ z^{-2} & -1 \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_0 \end{bmatrix} \quad (9.11)$$

  - The reduced-complexity 2-parallel FIR filter structure by transposition is shown on next page

- **Signal-flow graph of the 2-parallel FIR filter**
- **Transposed signal-flow graph**

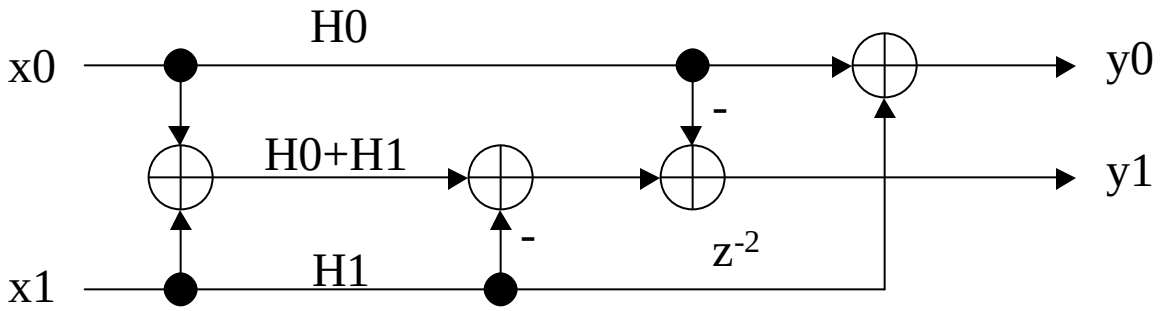


Fig. (a)

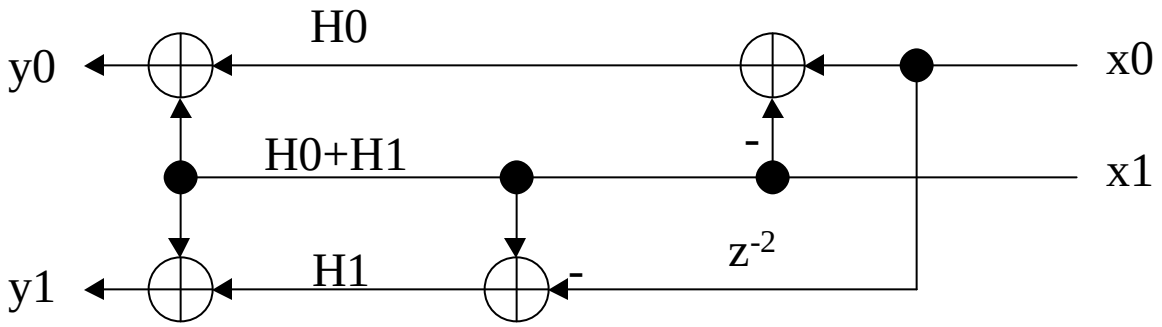


Fig. (b)

(c) Block diagram of the transposed reduced-complexity 2-parallel FIR filter

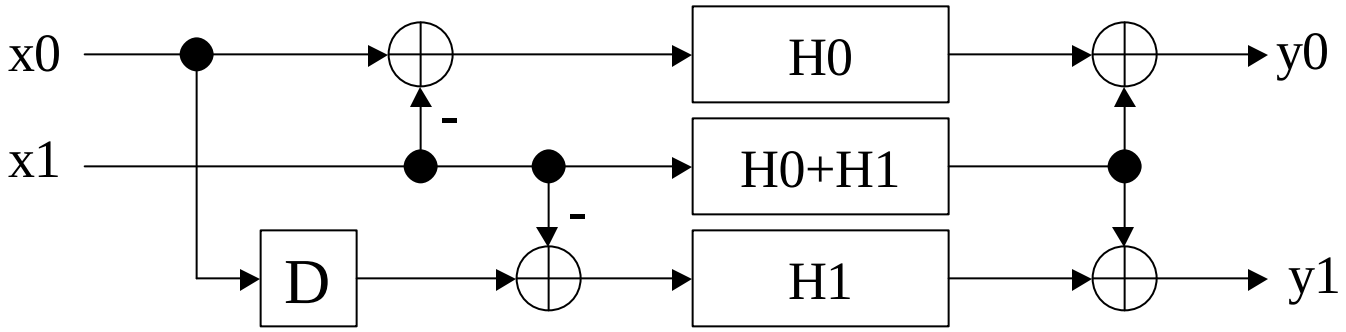


Fig. (c)

Parallel FIR Filters (cont'd)

Parallel Filter Algorithms from Linear Convolutions

- Any LXL convolution algorithm can be used to derive an L-parallel fast filter structure
- Example: the transpose of the matrix in a 2X2 linear convolution algorithm (9.12) can be used to obtain the 2-parallel filter (9.13):

$$\begin{bmatrix} s_2 \\ s_1 \\ s_0 \end{bmatrix} = \begin{bmatrix} h_1 & 0 \\ h_0 & h_1 \\ 0 & h_0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_0 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_1 & H_0 & 0 \\ 0 & H_1 & H_0 \end{bmatrix} \cdot \begin{bmatrix} z^{-2} X_1 \\ X_0 \\ X_1 \end{bmatrix}$$

(9. 12) (9.13)

- Example: To generate a 2-parallel filter using 2X2 fast convolution, consider the following optimal 2X2 linear convolution:

$$s = C \cdot H \cdot A \cdot X$$

$$\begin{bmatrix} s_2 \\ s_1 \\ s_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot diag \begin{pmatrix} h_1 \\ h_0 + h_1 \\ h_0 \end{pmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_0 \end{bmatrix} \qquad (9.14)$$

  - **Note:** Flipping the samples in the sequences {s}, {h}, and {x} preserves the convolution formulation (i.e., the same **C** and **A** matrices can be used with the flipped sequences)
  - Taking the transpose of this algorithm, we can get the matrix form of the reduced-complexity 2-parallel filtering structure:

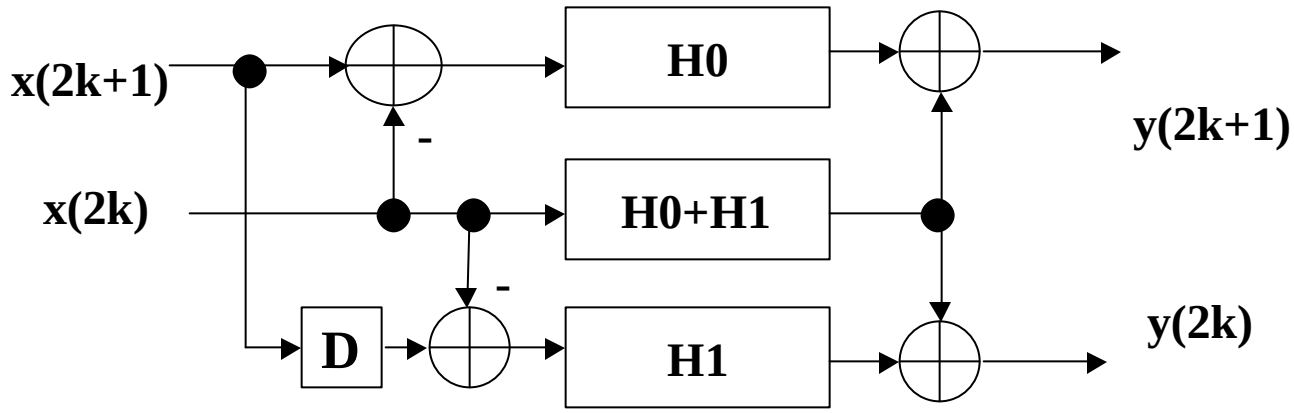
$$Y = \left(C \cdot H \cdot A\right)^T \cdot X = Q \cdot H \cdot P \cdot X$$

– The matrix form of the reduced-complexity 2-parallel filtering structure

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \cdot diag\left(\begin{matrix} H_1 \\ H_0 + H_1 \\ H_0 \end{matrix} \right) \cdot \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} z^{-2} X_1 \\ X_0 \\ X_1 \end{bmatrix} \quad (9.15)$$

– The 2-parallel architecture resulting from the matrix form is shown as follows

– **Conclusion:** this method leads to the same architecture that was obtained using the direct transposition of the 2-parallel FFA



Parallel FIR Filters (cont'd)

Fast Parallel FIR Algorithms for Large Block Sizes

- Parallel FIR filters with long block sizes can be designed by cascading smaller length fast parallel filters
- Example: an m-parallel FFA can be cascaded with an n-parallel FFA to produce an $(m \times n)$-parallel filtering structure. The set of FIR filters resulting from the application of the m-parallel FFA can be further decomposed, one at a time, by the application of the n-parallel FFA. The resulting set of filters will be of length $N/(m \times n)$.
- When cascading the FFAs, it is important to keep track of both the number of multiplications and the number of additions required for the filtering structure

– The number of required multiplications for an L-parallel filter with $L = L_1 L_2 \cdots L_r$ is given by:

$$M = \frac{N}{\prod_{i=1}^{r} L_i} \prod_{i=1}^{r} M_i \qquad (9.16)$$

- where r is the number of levels of FFAs used, L_i is the block size of the FFA at level-i, M_i is the number of filters that result from the applications of the i-th FFA and N is the length of the filter

– The number of required additions can be calculated as follows:

$$A = A_i \prod_{i=2}^{r} L_i + \sum_{i=2}^{r} \left[A_i \left(\prod_{j=i+1}^{r} L_j \right) \left(\prod_{k=1}^{i-1} M_k \right) \right] + \left(\prod_{i=1}^{r} M_i \right) \left(\frac{N}{\prod_{i=1}^{r} L_i} - 1 \right) \qquad (9.17)$$

- where A_i is the number of pre/post-processing adders required by the i-th FFA

– For example: consider the case of cascading two 2-parallel reduce-complexity FFAs, the resulting 4-parallel filtering structure would require a total of 9N/4 multiplications and 20+9(N/4-1) additions. Compared with the traditional 4-parallel filter which requires 4N multiplications. This results in a 44% hardware (area) savings

- **Example:** (Example 9.2.1, p.268) Calculating the hardware complexity
  – Calculate the number of multiplications and additions required to implement a 24-tap filter with block size of L=6 for both the cases $\{L_1 = 2, L_2 = 3\}$ and $\{L_1 = 3, L_2 = 2\}$:
    - For the case $\{L_1 = 2, L_2 = 3\}$:

$$M_1 = 3, \quad A_1 = 4, \quad M_2 = 6, \quad A_2 = 10,$$

$$M = \frac{24}{(2\times 3)} \times (3\times 6) = 72, \quad A = (4\times 3) + (10\times 3) + (3\times 6)\left[\frac{24}{(2\times 3)} - 1\right] = 96$$

- For the case $\{L_1 = 3, L_2 = 2\}$:

$$M_1 = 6, \quad A_1 = 10, \quad M_2 = 3, \quad A_2 = 4,$$

$$M = \frac{24}{(3\times 2)} \times (6\times 3) = 72, \quad A = (10\times 2) + (4\times 6) + (6\times 3)\left[\frac{24}{(3\times 2)} - 1\right] = 98$$

- How are the FFAs cascaded?
  - Consider the design of a parallel FIR filter with a block size of 4, using (9.3), we have

$$\begin{aligned} Y &= Y_0 + z^{-1}Y_1 + z^{-2}Y_2 + z^{-3}Y_3 \\ &= \left(X_0 + z^{-1}X_1 + z^{-2}X_2 + z^{-3}X_3\right)\cdot \\ &\quad \left(H_0 + z^{-1}H_1 + z^{-2}H_2 + z^{-3}H_3\right) \end{aligned} \qquad \textbf{(9.18)}$$

  - The reduced-complexity 4-parallel filtering structure is obtained by first applying the 2-parallel FFA to (9.18), then applying the FFA a second time to each of the filtering operations that result from the first application of the FFA
  - From (9.18), we have (see the next page):

– (cont'd) $Y=\left(X'_0+z^{-1}X'_1\right)\cdot\left(H'_0+z^{-1}H'_1\right)$

- where $\left\{\begin{array}{ll} X'_0=X_0+z^{-2}X_{2,} & X'_1=X_1+z^{-2}X_3 \\ H'_0=H_0+z^{-2}H_{2,} & H'_1=H_1+z^{-2}H_3 \end{array}\right\}$

– Application-1

$$Y=X_0^{'}H_0^{'}+z^{-1}\left[\left(X_0^{'}+X_1^{'}\right)\cdot\left(H_0^{'}+H_1^{'}\right)-X_0^{'}H_0^{'}-X_1^{'}H_1^{'}\right]+z^{-2}X_1^{'}H_1^{'} \quad (9.19)$$

- The 2-parallel FFA is then applied a second time to each of the filtering operations $\left\{X'_0H'_0, X'_1H'_1, \left(X'_0+X'_1\right)\cdot\left(H'_0+H'_1\right)\right\}$of (9.19)

– Application-2

- Filtering Operation $\left\{X'_0H'_0\right\}$

$$X_0^{'}H_0^{'}=\left(X_0+z^{-2}X_2\right)\left(H_0+z^{-2}H_2\right)$$

$$=X_0H_0+z^{-2}\left[\left(X_0+X_2\right)\cdot\left(H_0+H_2\right)-X_0H_0-X_2H_2\right]+z^{-4}X_2H_2$$

- Filtering Operation $\{X'_1 H'_1\}$

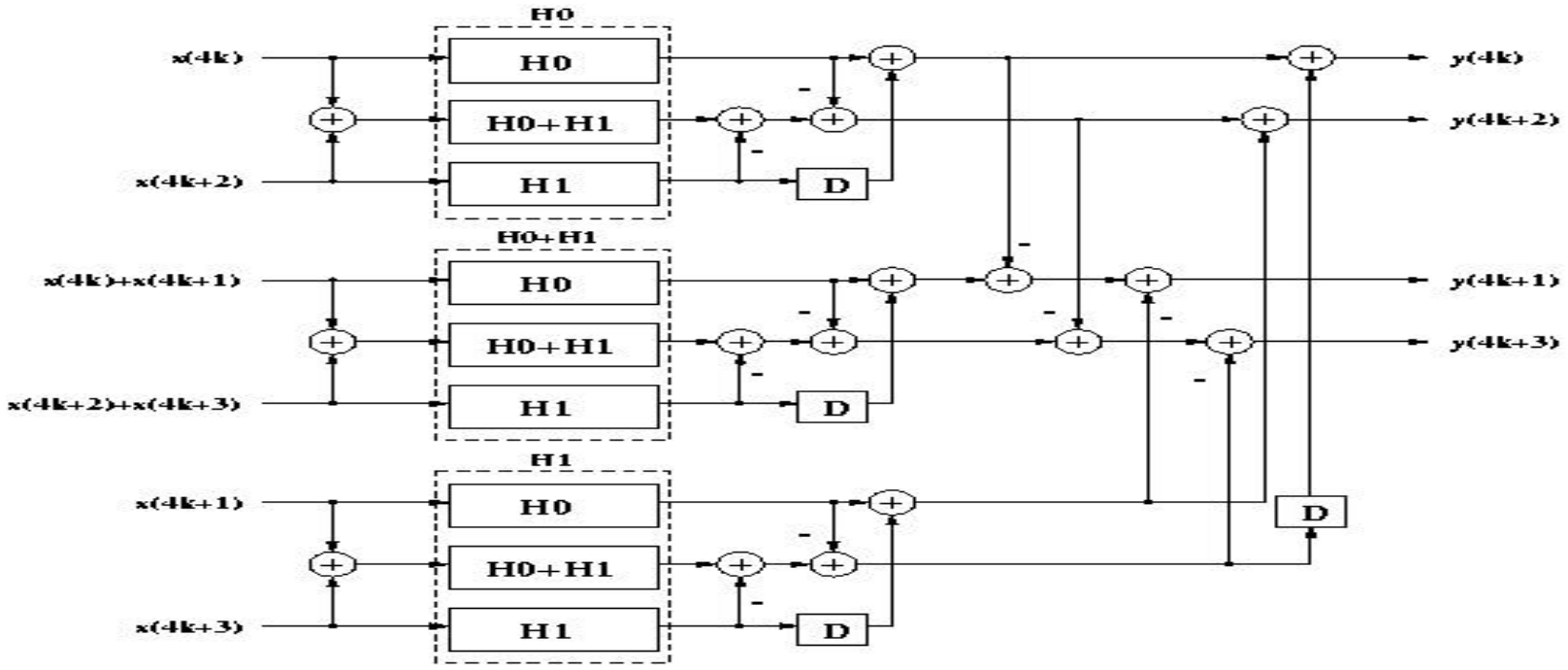
$$X'_1 H'_1 = (X_1 + z^{-2} X_3)(H_1 + z^{-2} H_3)$$
$$= X_1 H_1 + z^{-2}[(X_1 + X_3)\cdot(H_1 + H_3) - X_1 H_1 - X_3 H_3] + z^{-4} X_3 H_3$$

- Filtering Operation $\{(X'_0 + X'_1)(H'_0 + H'_1)\}$

$$(X'_0 + X'_1)(H'_0 + H'_1) = [(X_0 + X_1) + z^{-2}(X_2 + X_3)]\cdot[(H_0 + H_1) + z^{-2}(H_2 + H_3)]$$
$$= [(X_0 + X_1)(H_0 + H_1)] + z^{-4}[(X_2 + X_3)(H_2 + H_3)]$$
$$+ z^{-2}\begin{bmatrix}(X_0 + X_1 + X_2 + X_3)(H_0 + H_1 + H_2 + H_3)\\ -(X_0 + X_1)(H_0 + H_1) - (X_2 + X_3)(H_2 + H_3)\end{bmatrix}$$

– The second application of the 2-parallel FFA leads to the 4-parallel filtering structure (shown on the next page), which requires 9 filtering operations with length N/4

Reduced-complexity 4-parallel FIR filter (cascaded 2 by 2)



Discrete Cosine Transform and Inverse DCT

- The discrete cosine transform (DCT) is a frequency transform used in still or moving video compression. We discuss the fast implementations of DCT based on algorithm-architecture transformations and the decimation-in-frequency approach
- Denote the DCT of the data sequence x(n), n=0, 1,…, N-1, by X(k), k=0, 1, …, N-1. The DCT and inverse DCT (IDCT) are described by the following equations:
  - DCT:
    $$X(k) = e(k)\sum_{n=0}^{N-1} x(n)\cos\left[\frac{(2n+1)k}{2N}p\right], \quad k = 0,1,\cdots,N-1 \qquad (9.20)$$
  - IDCT:
    $$x(n) = \frac{2}{N}\sum_{k=0}^{N-1} e(k)X(k)\cos\left[\frac{(2n+1)k}{2N}p\right], \quad n = 0,1,\cdots,N-1 \qquad (9.21)$$

- where $$e(k) = \begin{cases} 1/\sqrt{2}, & k = 0 \\ 1, & otherwise \end{cases}$$

- Note: DCT is an orthogonal transform, i.e., the transformation matrix for IDCT is a scaled version of the transpose of that for the DCT and vice versa. Therefore, the DCT architecture can be obtained by "transposing" the IDCT, i.e., reversing the direction of the arrows in the flow graph of IDCT, and the IDCT can be obtained by "transposing" the DCT
- Direct implementation of DCT or IDCT requires N(N-1) multiplication operations, i.e., $O(N^2)$, which is hardware expensive.
- Strength reduction can reduce the multiplication complexity of a 8-point DCT from 56 to 13.

- Example (Example 9.3.1, p.277) Consider the 8-point DCT

$$X(k) = e(k)\sum_{n=0}^{7} x(n)\cos\left[\frac{(2n+1)k}{16}p\right], \quad k = 0,1,\cdots,7$$

$$\textit{where} \quad e(k) = \begin{cases} 1/\sqrt{2}, & k = 0 \\ 1, & \textit{otherwise} \end{cases}$$

  - It can be written in matrix form as follows: (where $c_i = \cos ip/16$)

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 \\ c_1 & c_3 & c_5 & c_7 & c_9 & c_{11} & c_{13} & c_{15} \\ c_2 & c_6 & c_{10} & c_{14} & c_{18} & c_{22} & c_{26} & c_{30} \\ c_3 & c_9 & c_{15} & c_{21} & c_{27} & c_1 & c_7 & c_{13} \\ c_4 & c_{12} & c_{20} & c_{28} & c_4 & c_{12} & c_{20} & c_{28} \\ c_5 & c_{15} & c_{25} & c_3 & c_{13} & c_{23} & c_1 & c_{11} \\ c_6 & c_{18} & c_{30} & c_{10} & c_{22} & c_2 & c_{14} & c_{26} \\ c_7 & c_{21} & c_3 & c_{17} & c_{31} & c_{13} & c_{27} & c_9 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}$$

– The algorithm-architecture mapping for the 8-point DCT can be carried out in three steps
  - **First Step**: Using trigonometric properties, the 8-point DCT can be rewritten as in next page

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 & c_4 \\ c_1 & c_3 & c_5 & c_7 & -c_7 & -c_5 & -c_3 & -c_1 \\ c_2 & c_6 & -c_6 & -c_2 & -c_2 & -c_6 & c_6 & c_2 \\ c_3 & -c_7 & -c_1 & -c_5 & c_5 & c_1 & c_7 & -c_3 \\ c_4 & -c_4 & -c_4 & c_4 & c_4 & -c_4 & -c_4 & c_4 \\ c_5 & -c_1 & c_7 & c_3 & -c_3 & -c_7 & c_1 & -c_5 \\ c_6 & -c_2 & c_2 & -c_6 & -c_6 & c_2 & -c_2 & c_6 \\ c_7 & -c_5 & c_3 & -c_1 & c_1 & -c_3 & c_5 & -c_7 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} \quad (9.22)$$

– (continued)

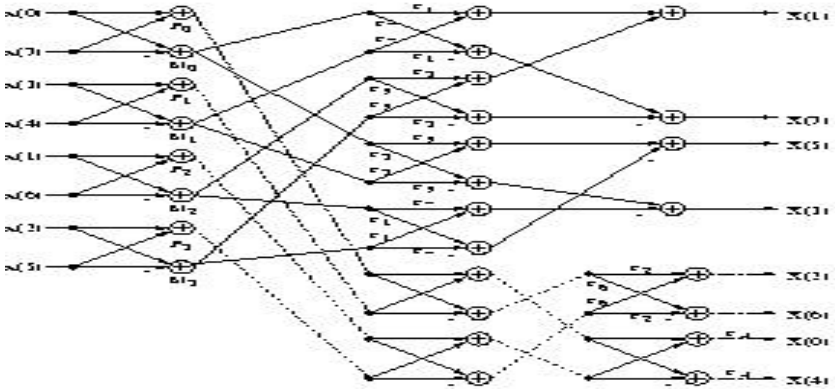
$$X(1) = M_0 c_1 + M_1 c_7 + M_2 c_3 + M_3 c_5, \quad X(2) = M_{10} c_2 + M_{11} c_6$$
$$X(7) = M_0 c_7 - M_1 c_1 + M_2 c_5 + M_3 c_3, \quad X(6) = M_{10} c_6 - M_{11} c_2$$
$$X(3) = M_0 c_3 - M_1 c_5 - M_2 c_7 - M_3 c_1, \quad X(4) = M_{100} \cdot c_4 \qquad (9.23)$$
$$X(5) = M_0 c_5 + M_1 c_3 - M_2 c_1 + M_3 c_7, \quad X(0) = P_{100} \cdot c_4$$

– where

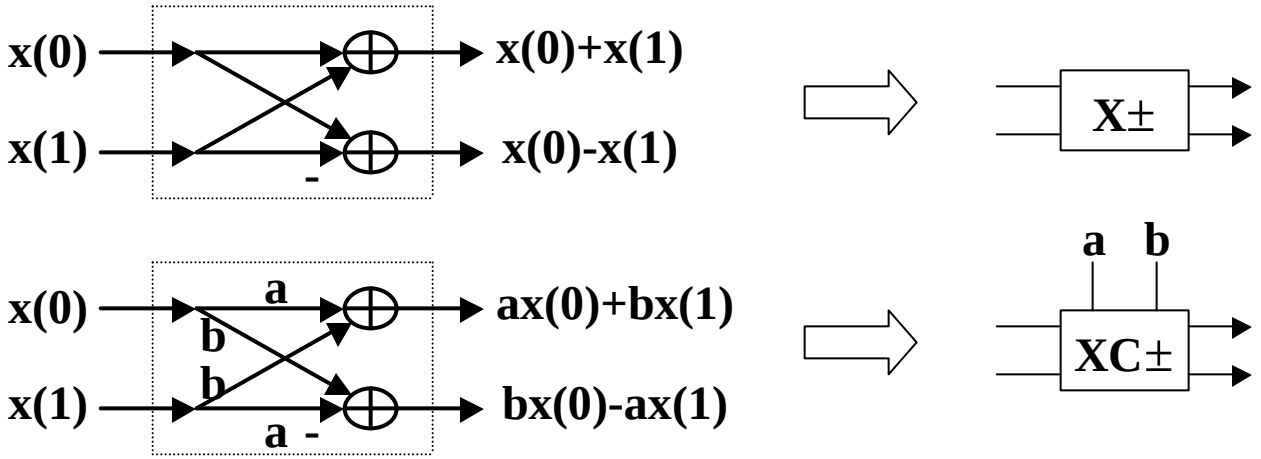
$$M_0 = x_0 - x_7, \quad M_1 = x_3 - x_4, \quad M_2 = x_1 - x_6, \quad M_3 = x_2 - x_5,$$
$$P_0 = x_0 + x_7, \quad P_1 = x_3 + x_4, \quad P_2 = x_1 + x_6, \quad P_3 = x_2 + x_5, \qquad (9.24)$$
$$M_{10} = P_0 - P_1, \quad M_{11} = P_2 - P_3, \quad P_{10} = P_0 + P_1, \quad P_{11} = P_2 + P_3,$$
$$M_{100} = P_{10} - P_{11}, \quad P_{100} = P_{10} + P_{11}$$

– The following figure (on the next page) shows the DCT architecture according to **(9.23) and (9.24)** with 22 multiplications.

Figure: The implementation of 8-point DCT structure in the first step (also see Fig. 9.10, p.279)

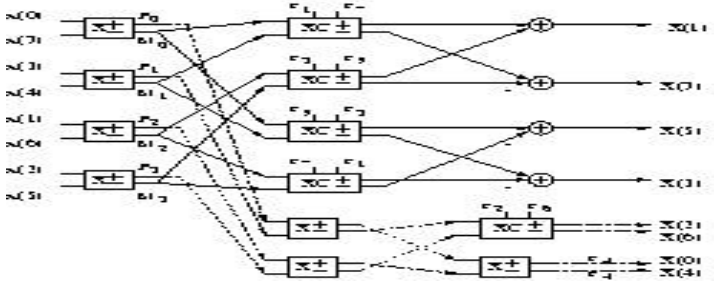


- **Second step**, the DCT structure (see Fig. 9.10, p.279) is grouped into different functional units represented by blocks and then the whole DCT structure is transformed into a block diagram
  - Two major blocks are defined as shown in the following figure

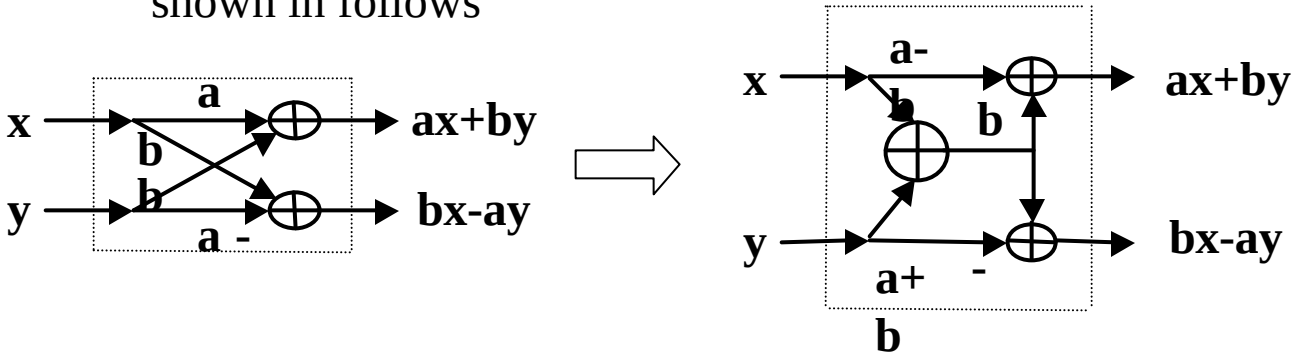


  - The transformed block diagram for an 8-point DCT is shown in the next page (also see Fig. 9.12 in p.280 of text book)

Figure: The implementation of 8-point DCT structure in the second step (also see Fig. 9.12, p.280)

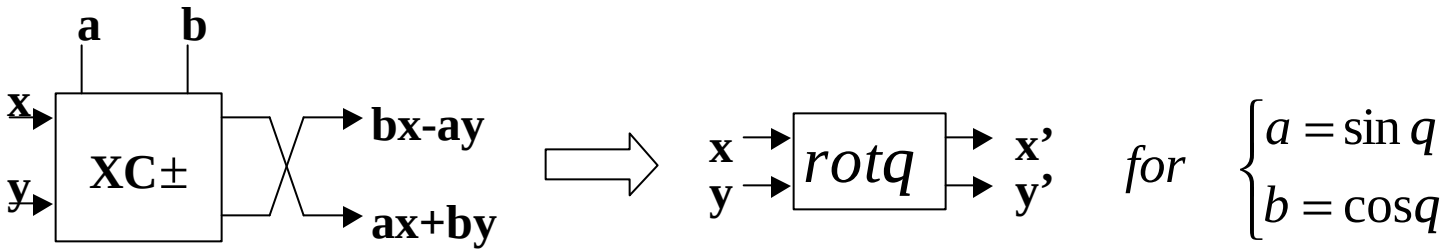


- **Third step**: Reduced-complexity implementations of various blocks are exploited (see Fig. 9.13, p.281)
  - The block **XC±** can be realized using 3 multiplications and 3 additions instead of using 4 multiplications and 2 additions, as shown in follows

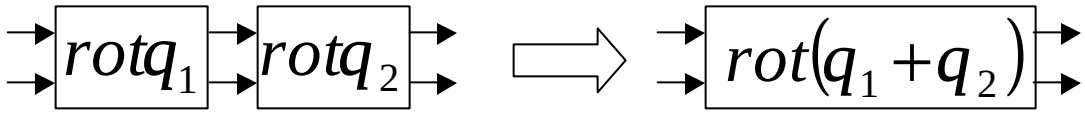


  - Define the block **XC±** with $\{a = \sin q, \quad b = \cos q\}$ and reversed outputs as a rotator block *rot q* that performs the following computation:

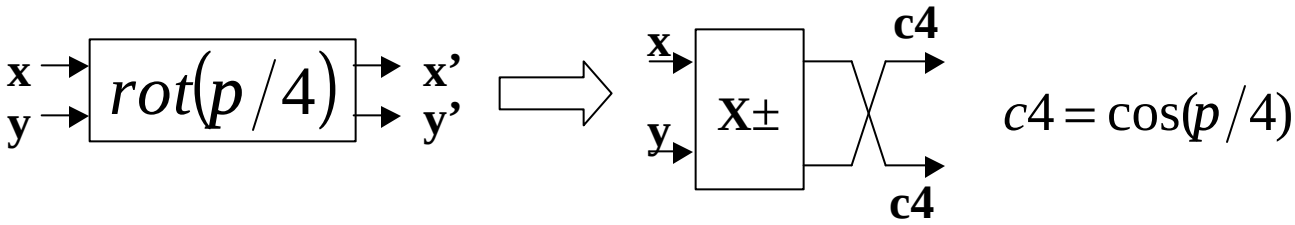
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos q & -\sin q \\ \sin q & \cos q \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix}$$



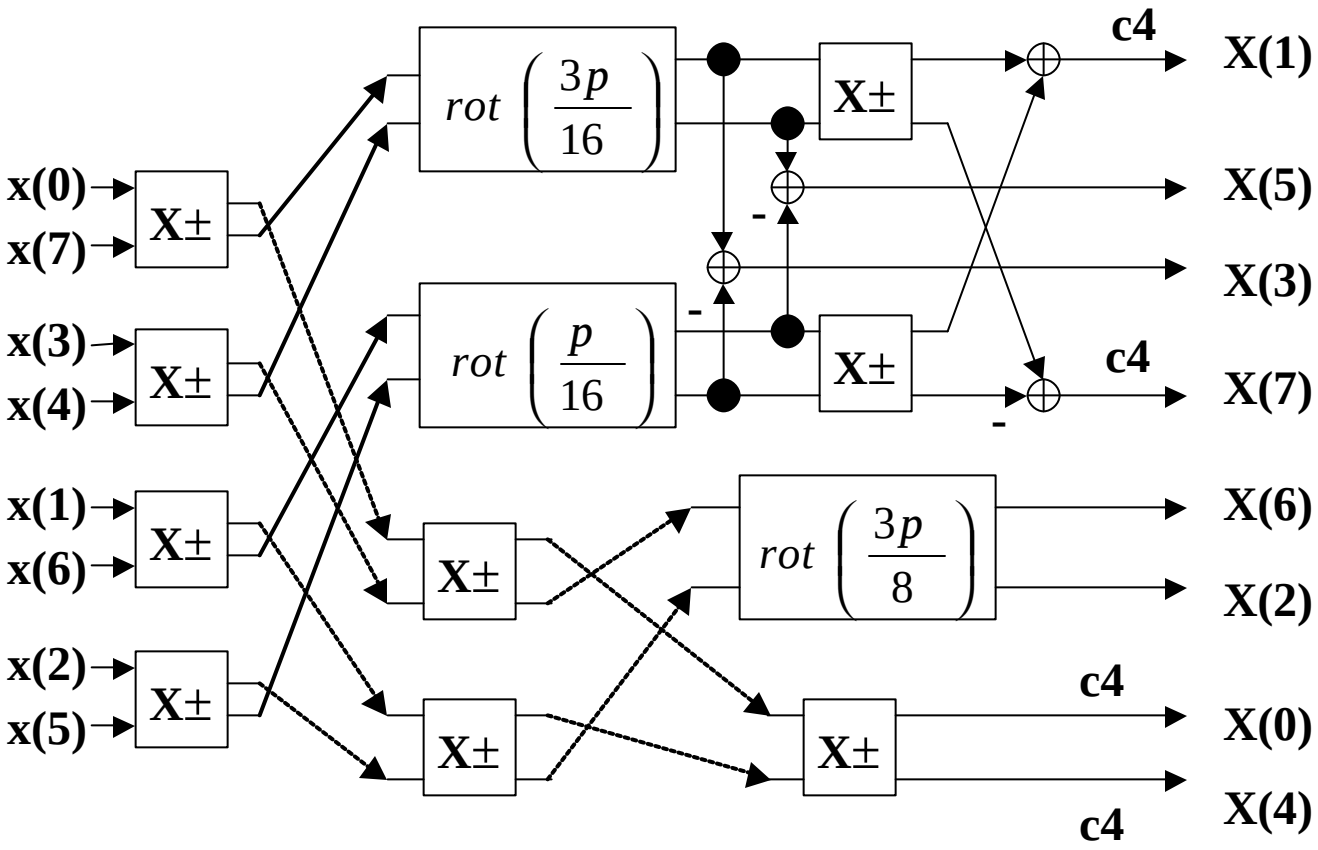
– Note: The angles of cascaded rotators can be simply added, as shown in the transformation block as follows:



– Note: Based on the fact that a rotator with $\{q = p/4\}$ is just like the block X± , we modify it as the following structure:



– From the three steps, we obtain the final structure where only 13 multiplications are required (also see Fig. 9.14, p.282)



Discrete Cosine Transform and Inverse DCT

Decimation-in-Frequency Fast DCT for 2^m-Point DCT

- The fast 2^m-point DCT/IDCT structures can be derived by the decimation-in-frequency approach, which is commonly used to derive the FFT structure to compute the discrete-Fourier transform (DFT). By power-of-2 decomposition, this algorithm reduces the number of multiplications to about

$$\{(N/2)\log_2 N\}$$

- We only derive the fast IDCT computation (The fast DCT structure can be obtained from IDCT by "transposition" according to their computation symmetry). For simplicity, the 2/N scaling factor in (9.21) is ignored in the derivation.

– Define $\hat{X}(k) = e(k) \cdot X(k)$ and decompose x(n) into even and odd indexes of k as follows

$$x(n) = \sum_{k=0}^{N-1} \hat{X}(k) \cos\left[\frac{(2n+1)k}{2N}\pi\right]$$

$$= \sum_{k=0}^{N/2-1} \hat{X}(2k) \cos\left[\frac{(2n+1)\pi(2k)}{2N}\right] + \sum_{k=0}^{N/2-1} \hat{X}(2k+1) \cos\left[\frac{(2n+1)\pi(2k+1)}{2N}\right]$$

$$= \sum_{k=0}^{N/2-1} \hat{X}(2k) \cos\left[\frac{(2n+1)\pi(2k)}{2N}\right] + \frac{1}{2\cos[(2n+1)\pi/2N]} \bullet$$

$$\sum_{k=0}^{N/2-1} 2\hat{X}(2k+1) \cos\left[\frac{(2n+1)\pi(2k+1)}{2N}\right] \cos\left[\frac{(2n+1)\pi}{2N}\right]$$

– Notice

$$2\cos\frac{(2n+1)\pi(2k+1)}{2N} \cdot \cos\frac{(2n+1)\pi}{2N} = \cos\frac{(2n+1)\pi(k+1)}{N} + \cos\frac{(2n+1)\pi k}{N}$$

– Therefore, (since $\cos\left[(2n+1)p\left(N/2-1+1\right)/N\right]=0$)

$$\sum_{k=0}^{N/2-1} 2\hat{X}(2k+1)\cos\left[\frac{(2n+1)p(2k+1)}{2N}\right]\cos\left[\frac{(2n+1)p}{2N}\right]$$

$$=\sum_{k=0}^{N/2-1}\hat{X}(2k+1)\cos\left[\frac{(2n+1)p(k+1)}{N}\right]+\sum_{k=0}^{N/2-1}\hat{X}(2k+1)\cos\left[\frac{(2n+1)pk}{N}\right]$$

$$=\sum_{k=0}^{N/2-2}\hat{X}(2k+1)\cos\left[\frac{(2n+1)p(k+1)}{N}\right]+\sum_{k=0}^{N/2-1}\hat{X}(2k+1)\cos\left[\frac{(2n+1)pk}{N}\right]$$

– Substitute k'=k+1 into the first term, we obtain

$$\sum_{k=0}^{N/2-2}\hat{X}(2k+1)\cos\left[\frac{(2n+1)p(k+1)}{N}\right]$$

$$=\sum_{k'=1}^{N/2-1}\hat{X}(2k'-1)\cos\left[\frac{(2n+1)pk'}{N}\right]=\sum_{k=0}^{N/2-1}\hat{X}(2k'-1)\cos\left[\frac{(2n+1)pk'}{N}\right]$$

- where $\hat{X}(-1)=0$

– Then, the IDCT can be rewritten as

$$x(n) = \sum_{k=0}^{N/2-1} \hat{X}(2k)\cos\left[\frac{(2n+1)p k}{2(N/2)}\right] + \frac{1}{2\cos[(2n+1)p/2N]}\cdot$$

$$\sum_{k=0}^{N/2-1}\left[\hat{X}(2k+1)+\hat{X}(2k-1)\right]\cos\left[\frac{(2n+1)p k}{2(N/2)}\right]$$

– Define

$$\begin{cases} G(k) \equiv \hat{X}(2k), \\ H(k) \equiv \hat{X}(2k+1)+\hat{X}(2k-1), \end{cases} \quad k = 0,1,\cdots, N/2-1 \qquad (9.25)$$

– and

$$\begin{cases} g(n) \equiv \sum_{k=0}^{N/2-1} \hat{X}(2k)\cos\left[\frac{(2n+1)p k}{2(N/2)}\right], & n = 0,1,\cdots, N/2-1 \\ h(n) \equiv \sum_{k=0}^{N/2-1}\left[\hat{X}(2k+1)+\hat{X}(2k-1)\right]\cos\left[\frac{(2n+1)p k}{2(N/2)}\right] \end{cases} \qquad (9.26)$$

– Clearly, G(k) & H(k) are the DCTs of g(n) & h(n), respectively.

– Since $\begin{cases} \cos\dfrac{(2(N-1-n)+1)pk}{N} = \cos\dfrac{(2n+1)pk}{N} \\ \cos\dfrac{(2(N-1-n)+1)p}{N} = -\cos\dfrac{(2n+1)p}{N} \end{cases}$

– Finally, we can get

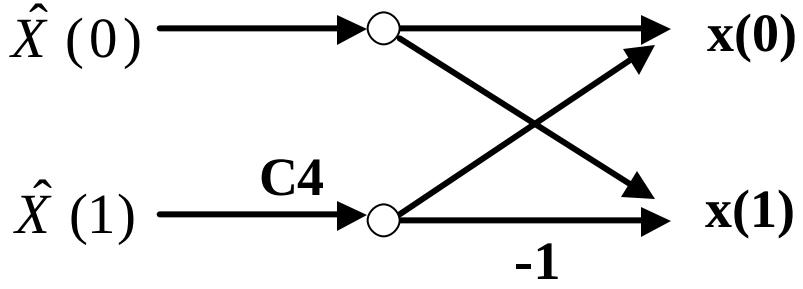
$$\begin{cases} x(n) = g(n) + \dfrac{1}{2\cos[(2n+1)p/(2N)]}h(n), \\ x(N-1-n) = g(n) - \dfrac{1}{2\cos[(2n+1)p/(2N)]}h(n), \end{cases} \quad n = 0,1,\cdots,N/2-1 \tag{9.27}$$

– Therefore, the N-point IDCT in (9.21) has been expressed in terms of two N/2-point IDCTs in (9.26). By repeating this process, the IDCT can be decomposed further until it can be expressed in terms of 2-point IDCTs. (The DCT algorithm can also be decomposed similarly. Alternatively, it can be obtained by transposing the IDCT)

- Example (see Example 9.3.2, p.284) Construct the 2-point IDCT butterfly architecture.
  - The 2-point IDCT can be computed as

$$\begin{cases} x(0) = \hat{X}(0) + \hat{X}(1)\cos(p/4), \\ x(1) = \hat{X}(0) - \hat{X}(1)\cos(p/4), \end{cases}$$

  - The 2-point IDCT can be computed using the following butterfly architecture



- Example (Example 9.3.3, p.284) Construct the 8-point fast DCT architecture using 2-point IDCT butterfly architecture.
  - With N=8, the 8-point fast DCT algorithm can be rewritten as:

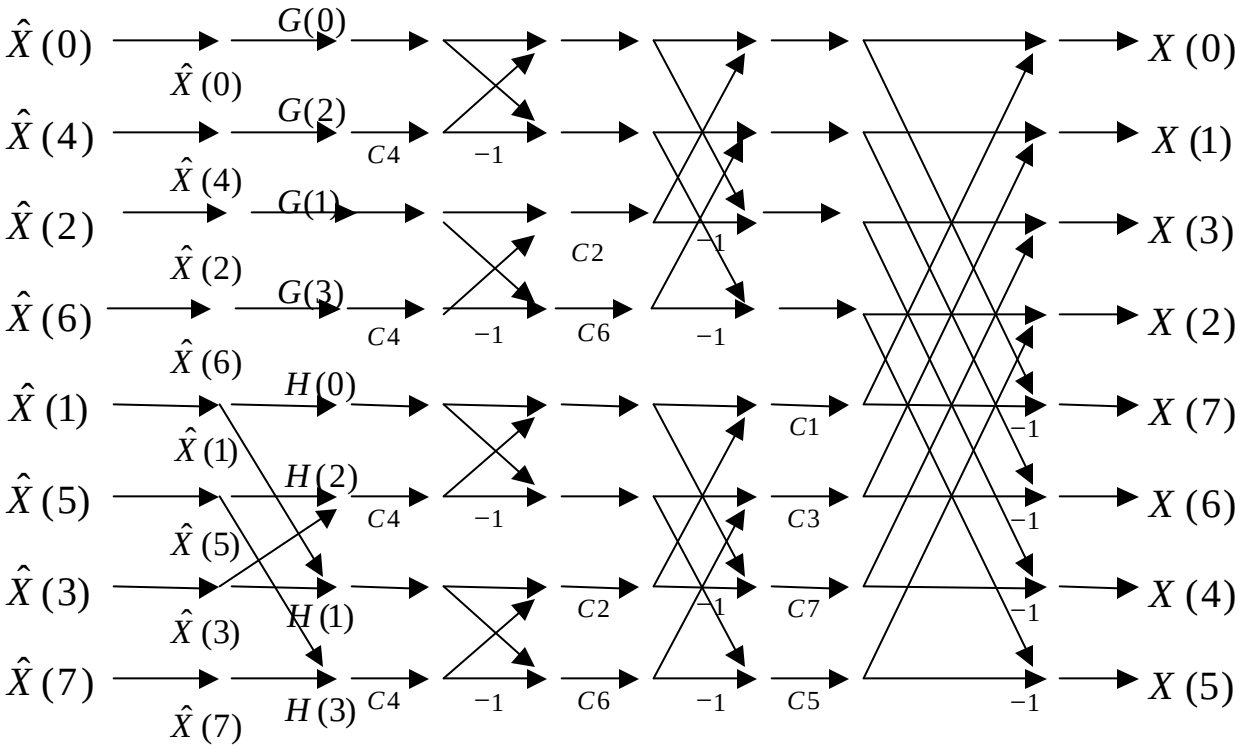
$$\begin{cases} G(k) \equiv \hat{X}(2k), \\ H(k) \equiv \hat{X}(2k+1) + \hat{X}(2k-1), \end{cases} \qquad k = 0,1,2,3$$

  - and

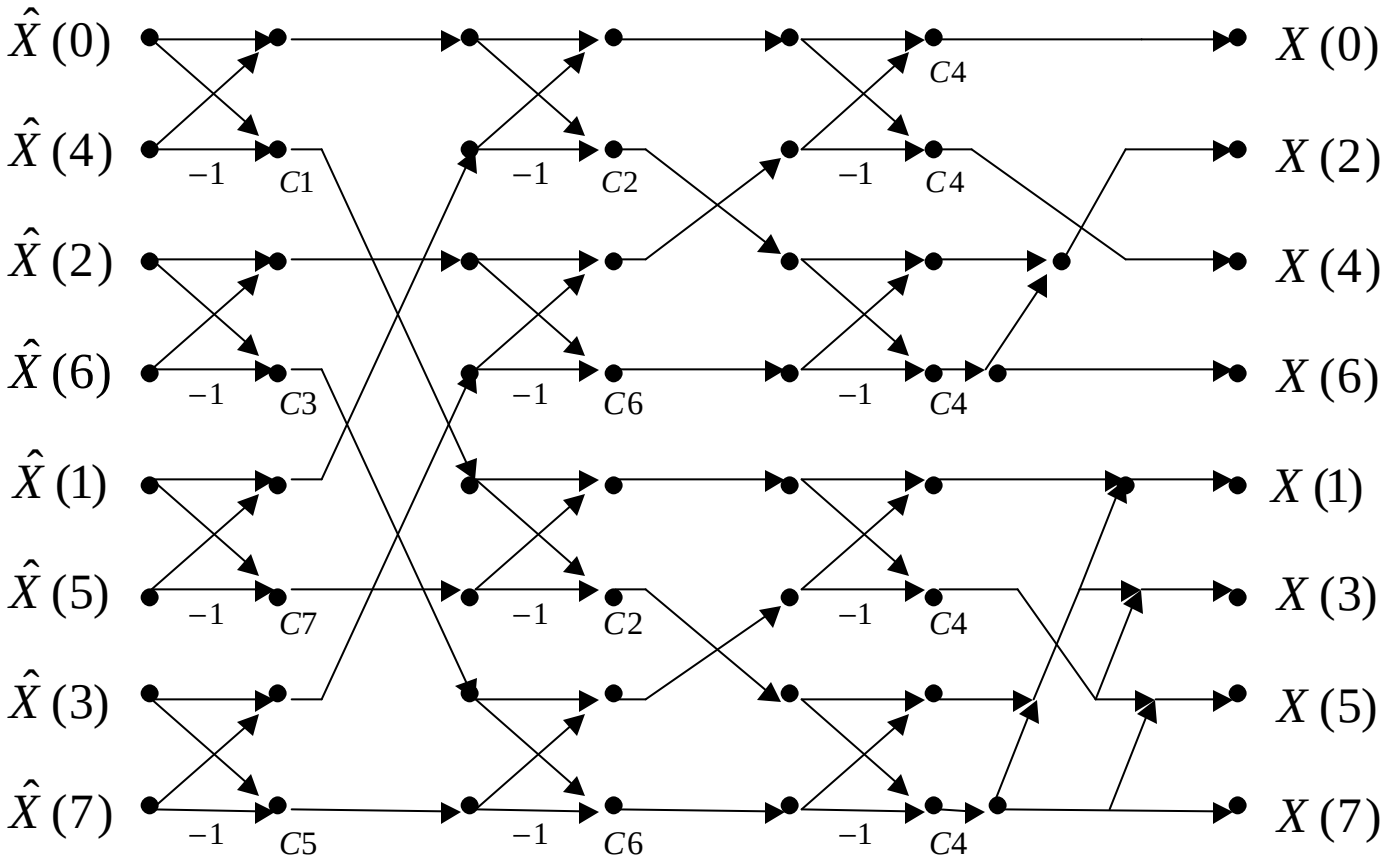
$$\begin{cases} g(n) = \sum_{k=0}^{3} G(k) \cos\left[\frac{(2n+1)\pi k}{8}\right], \quad n = 0,1,\cdots, N/2-1 \\ h(n) = \sum_{k=0}^{3-1} H(k) \cos\left[\frac{(2n+1)\pi k}{8}\right] \end{cases}$$

$$\begin{cases} x(n) = g(n) + \frac{1}{2\cos[(2n+1)\pi/16]} h(n), \\ x(N-1-n) = g(n) - \frac{1}{2\cos[(2n+1)\pi/16]} h(n), \end{cases}$$

– The 8-point fast IDCT is shown below (also see Fig.9.16, p.285), where only 13 multiplications are needed. This structure can be transposed to get the fast 8-point DCT architecture as shown on the next page (also see Fig. 9.17, p.286) (Note: for N=8, $C4 = 1/[2\cos(4p/16)] = \cos(p/4)$ in both figures)



Fast 8-point DCT Architecture



Chapter 10: Pipelined and Parallel Recursive and Adaptive Filters

Keshab K. Parhi

Outline

- Introduction
- Pipelining in 1st-Order IIR Digital Filters
- Pipelining in Higher-Order IIR Digital Filters
- Parallel Processing for IIR Filters
- Combined Pipelining and Parallel Processing for IIR Filters

Look-Ahead Computation

First-Order IIR Filter

- Consider a 1st-order linear time-invariant recursion (see Fig. 1)

$$y(n+1) = a \cdot y(n) + b \cdot u(n) \qquad (10.1)$$

- The iteration period of this filter is $\{T_m + T_a\}$, where $\{T_m, T_a\}$ represent word-level multiplication time and addition time
- In *look-ahead transformation*, the linear recursion is first iterated a few times to create additional concurrency.
- By recasting this recursion, we can express y(n+2) as a function of y(n) to obtain the following expression (see Fig. 2(a))

$$y(n+2) = a\left[ay(n) + bu(n)\right] + bu(n+1) \qquad (10.2)$$

- The iteration bound of this recursion is $2\left(T_m + T_a\right)/2$, the same as the original version, because the amount of computation and the number of logical delays inside the recursive loop have both doubled

- Another recursion equivalent to (10.2) is (10.3). Shown on Fig.2(b), its iteration bound is $(T_m + T_a)/2$, a factor of 2 lower than before.

$$y(n+2) = a^2 \cdot y(n) + ab \cdot u(n) + b \cdot u(n+1) \qquad (10.3)$$

- Applying *(M-1)* steps of look-ahead to the iteration of (10.1), we can obtain an equivalent implementation described by (see Fig. 3)

$$y(n+M) = a^M \cdot y(n) + \sum_{i=0}^{M-1} a^i \cdot b \cdot u(n+M-1-i) \qquad (10.4)$$

  - Note: the loop delay is z^{-M} instead of z^{-1}, which means that the loop computation must be completed in *M clock cycles (not 1 clock cycle)*. The iteration bound of this computation is $(T_m + T_a)/M$, which corresponds to a sample rate *M* times higher than that of the original filter
  - The terms $\{ab, a^2 b, \cdots, a^{M-1} b, a^M\}$ in (10.4) can be pre-computed (referred to as *pre-computation terms*). The second term in RHS of (10.4) is the look-ahead computation term (referred to as the *look-ahead complexity*); it is non-recursive and can be easily pipelined

Fig. 1

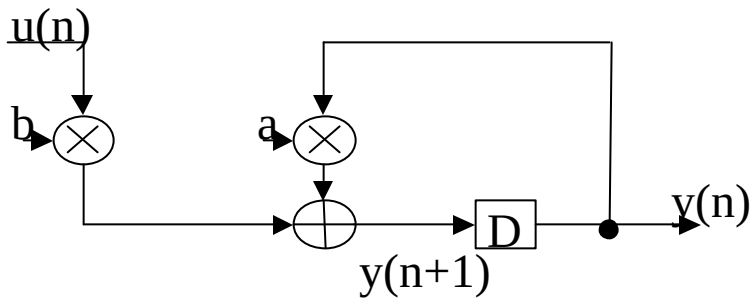


Fig.2.(a)

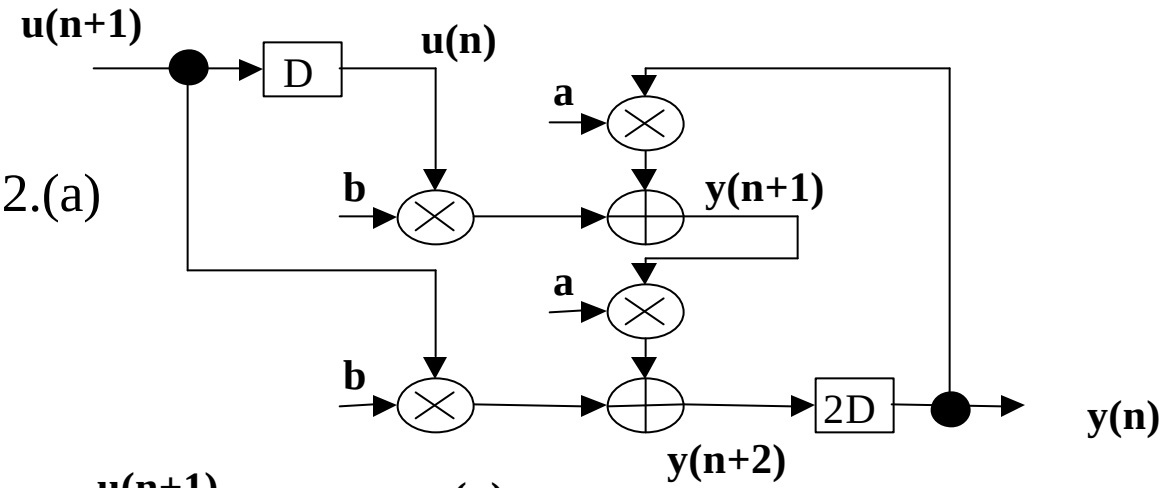
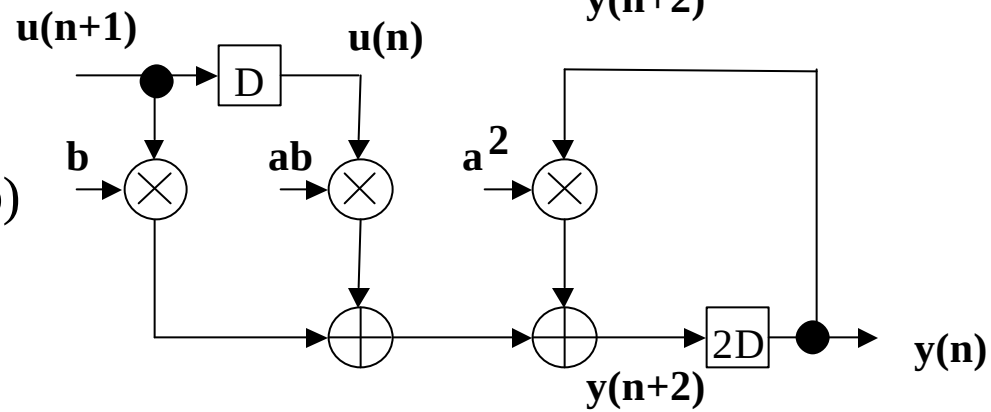


Fig.2.(b)



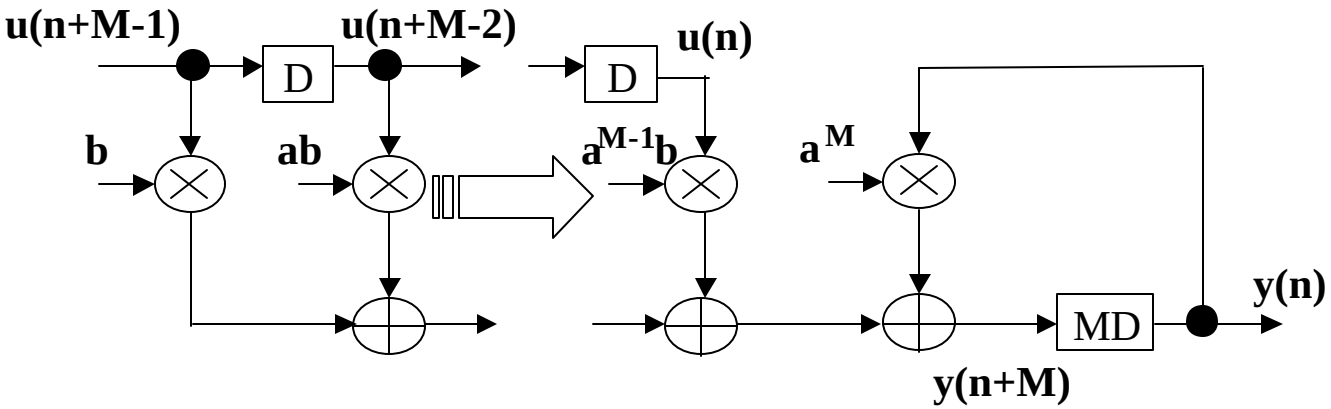


Fig. 3: M-stage Pipelinable 1st-Order IIR Filter

- Look-ahead computation has allowed a single serial computation to be transformed into M independent concurrent computations, and to pipeline the feedback loop to achieve high speed filtering of a single time series while maintaining full hardware utilization.
- Provided the multiplier and the adder can be conveniently pipelined, the iteration bound can be achieved by *retiming* or *cutset transformation* (see Chapter 4)

Pipelining in 1st-Order IIR Digital Filters

- **Example: Consider the 1st-order IIR filter transfer function**

$$H(z) = \frac{1}{1 - a \cdot z^{-1}} \qquad (10.5)$$

  - **The output sample y(n) can be computed using the input sample u(n) and the past output sample as follows:**

$$y(n) = a \cdot y(n-1) + u(n) \qquad (10.6)$$

  - **The sample rate of this recursive filter is limited by the computation time of one multiply-add operation**

- *Look-ahead techniques* add canceling poles and zeros with equal angular spacing at a distance from the origin which is same as that of the original pole. The pipelined filters are always stable provided that the original filter is stable
- The pipelined realizations require a *linear increase in complexity* but decomposition techniques can be used to obtain an implementation with logarithmic increase in hardware with respect to the number of loop pipeline stages

Pipelining in 1st-Order IIR Digital Filters (continued)

1. Look-Ahead Pipelining for 1st-Order IIR Filters

- Look-ahead pipelining adds canceling poles and zeroes to the transfer function such that the coefficients of $\{z^{-1}, \cdots, z^{-(M-1)}\}$ in the denominator of the transfer function are zero. Then, the output sample *y(n)* can be computed using the inputs and the output sample *y(n-M)* such that there are *M* delay elements in the critical loop, which in turn can be used to pipeline the critical loop by *M* stages and the sample rate can be increased by a factor *M*
- Example: Consider the 1st-order filter, $H(z) = 1/\left(1 - a \cdot z^{-1}\right)$, which has a pole at z=a (a≤1). A 3-stage pipelined equivalent stable filter can be derived by adding poles and zeroes at $z = ae^{\pm(j2p/3)}$, and is given by

$$H(z) = \frac{1 + a \cdot z^{-1} + a^2 \cdot z^{-2}}{1 - a^3 \cdot z^{-3}}$$

Pipelining in 1st-Order IIR Digital Filters (continued)

2. Look-Ahead Pipelining with Power-of-2 Decomposition

- With power-of-2 decomposition, an M-stage (for power-of-2 M) pipelined implementation for 1st-order IIR filter can be obtained by $\log_2 M$ sets of transformations
- Example: Consider a 1st-order recursive filter transfer function described by $H(z) = \left(b \cdot z^{-1}\right) / \left(1 - a \cdot z^{-1}\right)$. The equivalent pipelined transfer function can be described using the decomposition technique as follows

$$H(z) = \frac{b \cdot z^{-1} \prod_{i=0}^{\log_2 M - 1} \left(1 + a^{2^i} \cdot z^{-2^i}\right)}{1 - a^M \cdot z^{-M}} \qquad (10.7)$$

  - This pipelined implementation is derived by adding (M-1) poles and zeros at identical locations.
  - The original transfer function has a single pole at $z = a$ (see Fig.4(a)).

– The pipelined transfer function has poles at the following locations (see Fig.4(b) for M=8):

$$\left\{a, ae^{j2p/M}, \cdots, ae^{j(M-1)\cdot(2p)/M}\right\}$$

– The decomposition of the canceling zeros is shown in Fig.4(c). The i-th stage of the decomposed non-recursive portion implements 2^i zeros located at:

$$z = a\exp\left(j(2n+1)p/\left(2^i\right)\right), \quad n = 0,1,\cdots,\left(2^i - 1\right) \qquad (10.8)$$

– The i-th stage of the decomposed non-recursive portion requires a single pipelined multiplication operation independent of the stage number i

– The multiplication complexity of the pipelined implementation is

$$\left(\log_2 M + 2\right)$$

– The finite-precision pipelined filters suffer from inexact pole-zero cancellation, which leads to magnitude and phase error. These errors can be reduced by increasing the wordlength (see p. 323)

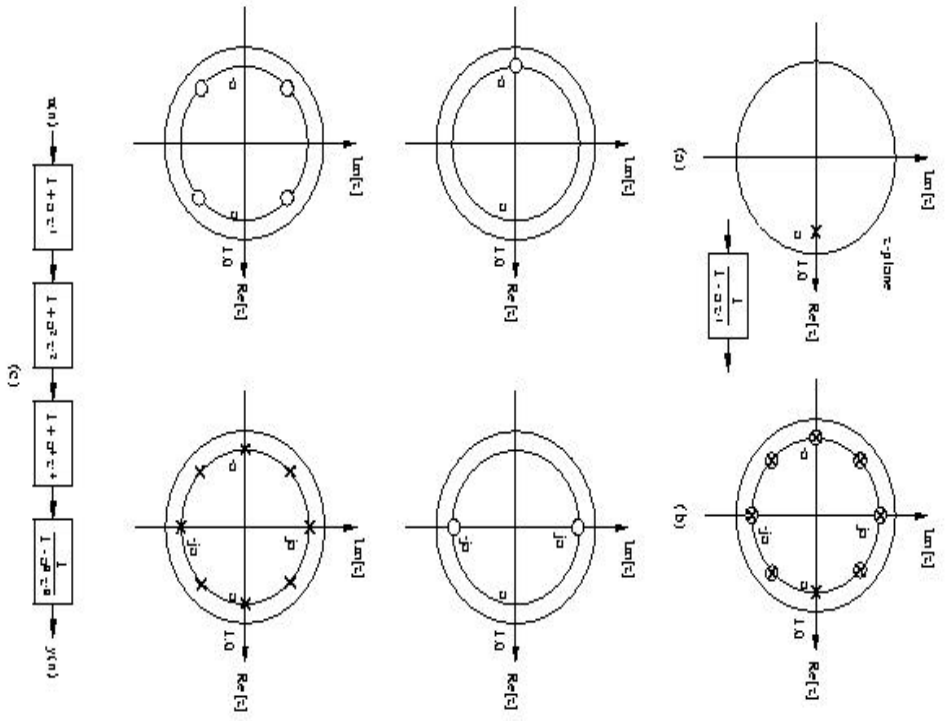


Fig. 4

- Pole representation of a 1^{ST}-order recursive filter
- Pole/zero representation of a 1^{ST}-order IIR with 8 loop stages
- Decomposition based on pipelined IIR for M = 8

Pipelining in 1st-Order IIR Digital Filters (continued)

3. Look-Ahead Pipelining with General Decomposition

- The idea of decomposition can be extended to any arbitrary number of loop pipelining stages M. If $M = M_1 M_2 \cdots M_p$, then the non-recursive stages implement $(M_1 - 1)$, $M_1(M_2 - 1)$, $\cdots$, $M_1 M_2 \cdots M_{p-1}(M_p - 1)$ zeros, respectively, totaling (M-1) zeros
- Example (Example 10.3.3, p.325) Consider the 1st-order IIR

$$H(z) = \frac{1}{1 - a \cdot z^{-1}}$$

  - A 12-stage pipelined decomposed implementation is given by

$$H(z) = \frac{\sum_{i=0}^{11} a^i \cdot z^{-i}}{1 - a^{12} \cdot z^{-12}} = \frac{\left(1 + az^{-1}\right)\left(1 + a^2 z^{-2} + a^4 z^{-4}\right)\left(1 + a^6 z^{-6}\right)}{1 - a^{12} \cdot z^{-12}}$$

- This implementation is based on a $2 \times 3 \times 2$ decomposition (see Fig. 5)

– The first section implements 1 zero at –a, the second section implements 4 zeros at $\left\{ae^{\pm jp/3}, ae^{\pm j2p/3}\right\}$, and the third section implements 6 zeros at $\left\{\pm ja, ae^{\pm jp/6}, ae^{\pm j5p/6}\right\}$

– Another decomposed implementation ($2\times2\times3$ decomposition) is given by

$$H(z)=\frac{\left(1+az^{-1}\right)\left(1+a^2z^{-2}\right)\left(1+a^4z^{-4}+a^8z^{-8}\right)}{1-a^{12}\cdot z^{-12}}$$

• The first section implements 1 zero at –a, the second section implements 2 zeros at $\pm ja$, and the third section implements 8 zeros at $\left\{ae^{\pm jp/6}, ae^{\pm jp/3}, ae^{\pm j2p/3}, ae^{\pm j5p/6}\right\}$

– The third decomposition ($3\times2\times2$ decomposition) is given by

$$H(z)=\frac{\left(1+az^{-1}+a^2z^{-2}\right)\left(1+a^3z^{-3}\right)\left(1+a^6z^{-6}\right)}{1-a^{12}\cdot z^{-12}}$$

• The first section implements 2 zeros at $\left\{ae^{\pm j2p/3}\right\}$, the second section implements 3 zeros at $\left\{-a, ae^{\pm jp/3}\right\}$, and the third section implements 6 zero at $\left\{ae^{\pm jp/6}, \pm ja, ae^{\pm j5p/6}\right\}$

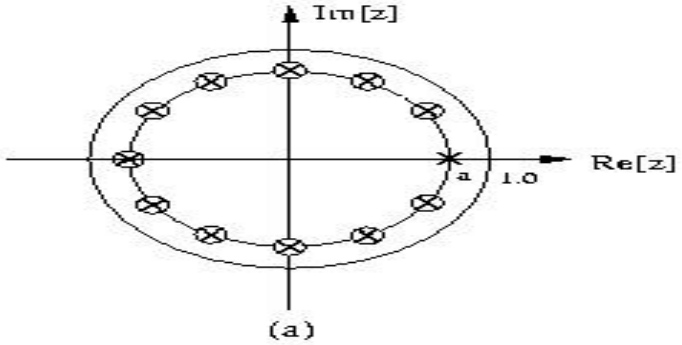


Fig.5(a)

Pole-zero location of a 12-stage pipelined 1^{ST}-order IIR

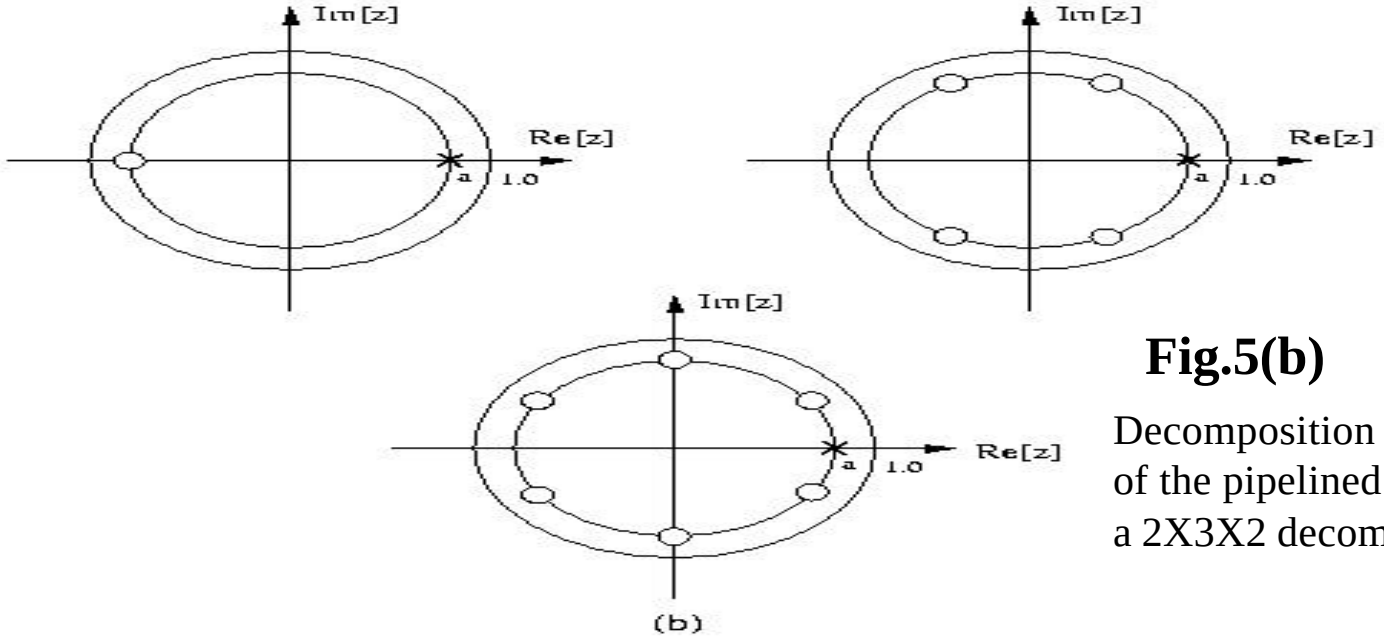


Fig.5(b)

Decomposition of the zeros of the pipelined filter for a 2X3X2 decomposition

Pipelining in Higher-order IIR Digital Filters

- Higher-order IIR digital filters can be pipelined by using clustered look-ahead or scattered look-ahead techniques. (For 1st-order IIR filters, these two look-ahead techniques reduce to the same form)
  - Clustered look-ahead: Pipelined realizations require a linear complexity in the number of loop pipelined stages and are not always guaranteed to be stable
  - Scattered look-ahead: Can be used to derive stable pipelined IIR filters
  - Decomposition technique: Can also be used to obtain area-efficient implementation for higher-order IIR for scattered look-ahead filters
  - Constrained filter design techniques: Achieve pipelining without pole-zero cancellation

- The transfer function of an N-th order direct-form recursive filter is described by

$$H(z) = \frac{\sum_{i=0}^{N} b_i z^{-i}}{1 - \sum_{i=1}^{N} a_i z^{-i}} \qquad (10.9)$$

- Equivalently, the output y(n) can be described in terms of the input sample u(n) and the past input/output samples as follows

$$\begin{aligned} y(n) &= \sum_{i=1}^{N} a_i y(n-i) + \sum_{i=0}^{N} b_i u(n-i) \\ &= \sum_{i=1}^{N} a_i y(n-i) + z(n) \end{aligned} \qquad (10.10)$$

  - The sample rate of this IIR filter realization is limited by the throughput of 1 multiplication and 1 addition, since the critical path contains a single delay element

Pipelining in Higher-order IIR Digital Filters (cont'd)

1. Clustered Look-Ahead Pipelining

- The basic idea of clustered look-ahead:
  - Add canceling poles and zeros to the filter transfer function such that the coefficients of $\{z^{-1}, \cdots, z^{-(M-1)}\}$ in the denominator of the transfer function are 0, and the output samples y(n) can be described in terms of the cluster of N past outputs:

    $\{y(n-M), \cdots, y(n-M-N+1)\}$
  - Hence the critical loop of this implementation contains M delay elements and a single multiplication. Therefore, this loop can be pipelined by M stages, and the sample rate can be increased by a factor M. This is referred to as *M-stage clustered look-ahead pipelining*

- **Example:** (Example 10.4.1, p.327) Consider the all-pole 2nd-order IIR filter with poles at $\{1/2, 3/4\}$. The transfer function of this filter is

$$H(z) = \frac{1}{1 - \frac{5}{4} z^{-1} + \frac{3}{8} z^{-2}} \qquad (10.11)$$

  - A 2-stage pipelined equivalent IIR filter can be obtained by eliminating the z^{-1} term in the denominator (i.e., multiplying both the numerator and denominator by $\left(1 + 5/4\ z^{-1}\right)$. The transformed transfer function is given by:

$$H(z) = \frac{1}{1 - \frac{5}{4} z^{-1} + \frac{3}{8} z^{-2}} \cdot \frac{1 + \frac{5}{4} z^{-1}}{1 + \frac{5}{4} z^{-1}}$$

$$= \frac{1 + \frac{5}{4} z^{-1}}{1 - \frac{19}{16} z^{-2} + \frac{15}{32} z^{-3}} \qquad (10.12)$$

  - From, the transfer function, we can see that the coefficient of z^{-1} in the denominator is zero. Hence, the critical path of this filter contains 2 delay elements and can be pipelined by 2 stages

– Similarly, a 3-stage pipelined realization can be derived by eliminating the terms of $\{z^{-1}, z^{-2}\}$ in the denominator of (10.21), which can be done by multiplying both numerator and denominator by

$$\left(1+5/4\,z^{-1}+19/16\,z^{-2}\right)$$

– The new transfer function is given by:

$$H(z) = \frac{1+\frac{5}{4}z^{-1}+\frac{19}{16}z^{-2}}{1-\frac{65}{64}z^{-3}+\frac{57}{128}z^{-4}} \qquad (10.13)$$

- **Computation complexity:** The numerator (non-recursive portion) of this pipelined filter needs (N+M) multiplications, and the denominator (recursive portion) needs N multiplications. Thus, the total complexity of this pipelined implementation is (N+N+M) multiplications
- **Stability:** The canceling poles and zeros are utilized for pipelining IIR filters. However, when the additional poles lie outside the unit circle, the filter becomes unstable. Note that the filters in (10.12) and (10.13) are unstable.

2. Stable Clustered Look-Ahead Filter Design

- If the desired pipeline delay M does not produce a stable filter, M should be increased until a stable pipelined filter is obtained. To obtain the optimal pipelining level M, numerical search methods are generally used
- Example (Example 10.4.3, p.330) Consider a 5-level (*M=5*) pipelined implementation of the following 2nd-order transfer function

$$H(z)=\frac{1}{1-1.5336z^{-1}+0.6889z^{-2}} \qquad (10.14)$$

  - By the stability analysis, it is shown that (*M=5*) does not meet the stability condition. Thus M is increased to *M=6* to obtain the following stable pipelined filter as

$$H(z)=\frac{1+1.5336z^{-1}+1.6630z^{-2}+1.4939z^{-3}+1.1454z^{-4}+0.7275z^{-5}}{1-1.3265z^{-6}+0.5011z^{-7}} \qquad (10.15)$$

3. Scattered Look-Ahead Pipelining

- Scattered look-ahead pipelining: The denominator of the transfer function in (10.9) is transformed in a way that it contains the N terms $\{z^{-M}, z^{-2M}, \cdots, z^{-NM}\}$. Equivalently, the state y(n) is computed in terms of N past scattered states y(n-M),y(n-2M),…, and y(n-NM)
- In scattered look-ahead, for each poles in the original filter, we introduce (M-1) canceling poles and zeros with equal angular spacing at a distance from the origin the same as that of the original pole.
  - Example: if the original filter has a pole at $z=p$, we add (M-1) poles and zeros at $\{z = p\exp(j2pk/M), \quad k = 1,2,\cdots,M-1\}$ to derive a pipelined realization with M loop pipeline stages
- Assume that the denominator of the transfer function can be factorized as follows:

$$D(z) = \prod_{i=1}^{N}(1 - p_i z^{-1}) \qquad (10.16)$$

(continued)

– Then, the pipelining process using the scattered look-ahead approach can be described by

$$H(z) = \frac{N(z)}{D(z)}$$

$$= \frac{N(z)\prod_{i=1}^{N}\prod_{k=1}^{M-1}\left(1 - p_i e^{j2kp/M} z^{-1}\right)}{\prod_{i=1}^{N}\prod_{k=0}^{M-1}\left(1 - p_i e^{j2kp/M} z^{-1}\right)} = \frac{N'(z)}{D'(z^M)} \quad (10.17)$$

- Example (Example 10.4.5, p.332) Consider the 2nd-order filter with complex conjugate poles at $z = re^{\pm jq}$. The filter transfer function is given by

$$H(z) = \frac{1}{1 - 2r\cos q z^{-1} + r^2 z^{-2}}$$

– We can pipeline this filter by 3 stages by introducing 4 additional poles and zeros at $\left\{z = re^{\pm j(q+2p/3)},\ \ z = re^{\pm j(q-2p/3)}\right\}$ if $q \neq 2p/3$

– (cont'd) The equivalent pipelined filter is then given by

$$H(z)=\frac{1+2r\cos q\cdot z^{-1}+(1+2\cos 2q)r^2\cdot z^{-2}+2r^3\cos q\cdot z^{-3}+r^4z^{-4}}{1-2r^3\cos(3q)\cdot z^{-3}+r^6z^{-6}}$$

– When $q=2p/3$, then only 1 additional pole and zero at $z=r$ is required for 3-stage pipelining since $z=re^{\pm j(q+2p/3)}=re^{\pm jq}$ and $z=re^{\pm j(q-2p/3)}=r$. The equivalent pipelined filter is then given by

$$H(z)=\frac{1-r\cdot z^{-1}}{(1+r\cdot z^{-1}+r^2\cdot z^{-2})(1-r\cdot z^{-1})}=\frac{1-r\cdot z^{-1}}{1-r^3\cdot z^{-3}}$$

- Example (Example 10.4.6, p.332) Consider the 2nd-order filter with real poles at $\{z=r_1, z=r_2\}$. The transfer function is given by

$$H(z)=\frac{1}{1-(r_1+r_2)\cdot z^{-1}+r_1r_2\cdot z^{-2}}$$

– (cont'd) A 3-stage pipelined realization is derived by adding poles(and zeros) at $\left\{z = r_1 e^{\pm j2p/3}, \quad z = r_2 e^{\pm j2p/3}\right\}$. The pipelined realization is given by

$$H(z) = \frac{1 + (r_1 + r_2)z^{-1} + (r_1^{\,2} + r_1 r_2 + r_2^{\,2})z^{-2} + r_1 r_2 (r_1 + r_2)z^{-3} + r_1^{\,2} r_2^{\,2} z^{-4}}{1 - (r_1^{\,3} + r_2^{\,3}) \cdot z^{-3} + r_1^{\,3} r_2^{\,3} z^{-6}}$$

– The pole-zero locations of a 3-stage pipelined 2nd-order filter with poles at z=1/2 and z=3/4 are shown in Fig. 6

- **CONCLUSIONS:**
  – If the original filter is stable, then the scattered look-ahead approach leads to stable pipelined filters, because the distance of the additional poles from the original is the same as that of the original filter
  – Multiplication Complexity: (NM+1) for the non-recursive portion in (10.17), and N for the recursive portion. Total pipelined filter multiplication complexity is (NM+N+1)

– (cont'd) The multiplication complexity is linear with respect to M. and is much greater than that of clustered look-ahead
– Also the latch complexity is square in M, because each multiplier is pipelined by M stages

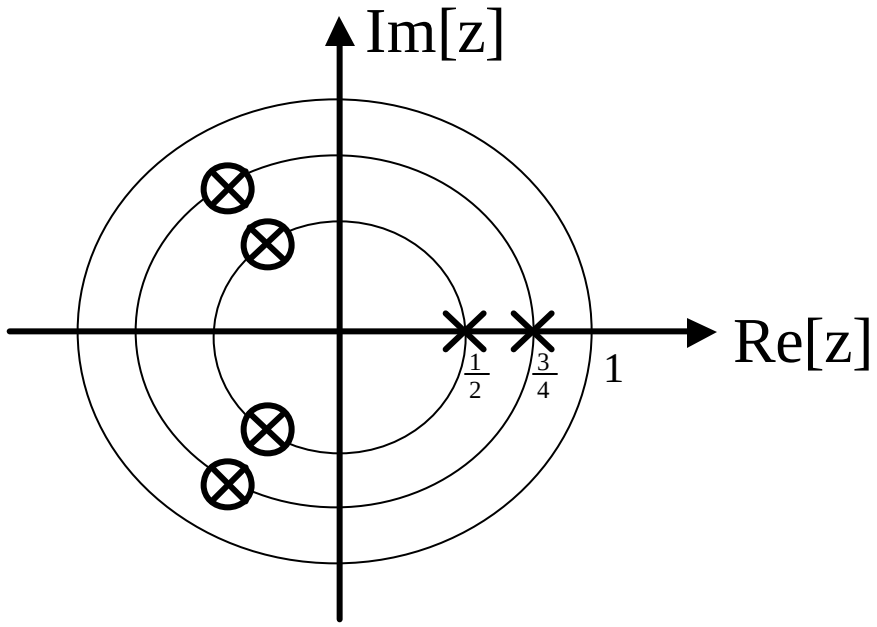


Fig. 6: Pole-zero representation of a 3-stage pipelined equivalent stable filter derived using scattered look-ahead approach

4. Scattered Look-Ahead Pipelining with Power-of-2 Decomposition

- This kind of decomposition technique will lead to a logarithmic increase in multiplication complexity (hardware) with respect to the level of pipelining
- Let the transfer function of a recursive digital filter be

$$H(z) = \frac{\sum_{i=0}^{N} b_i z^{-i}}{1 - \sum_{i=1}^{N} a_i \cdot z^{-i}} = \frac{N(z)}{D(z)} \qquad (10.18)$$

  - A 2-stage pipelined implementation can be obtained by multiplying by $\left(1 - \sum_{i=1}^{N} (-1)^i a_i \cdot z^{-i}\right)$ in the numerator and denominator. The equivalent 2-stage pipelined implementation is described by

$$H(z) = \frac{\left(\sum_{i=0}^{N} b_i z^{-i}\right)\left(1 - \sum_{i=1}^{N} (-1)^i a_i \cdot z^{-i}\right)}{\left(1 - \sum_{i=1}^{N} a_i \cdot z^{-i}\right)\left(1 - \sum_{i=1}^{N} (-1)^i a_i \cdot z^{-i}\right)} = \frac{N'(z)}{D'(z)} \qquad (10.19)$$

- (cont'd) Similarly, subsequent transformations can be applied to obtain 4, 8, and 16 stage pipelined implementations, respectively
- Thus, to obtain an M-stage pipelined implementation *(for power-of-2 M)*, $\log_2 M$ sets of such transformations need to be applied.
- By applying $(\log_2 M - 1)$ sets of such transformations, an equivalent transfer function (with M pipelining stages inside the recursive loop) can be derived, which requires a complexity of $(2N + N\log_2 M + 1)$ multiplications, a logarithmic complexity with respect to M.
- Note: the number of delays (or latches) is linear: the total number of delays (or latches) is approximately $NM(\log_2 M + 1)$, about NM delays in non-recursive portion, and $NM\log_2 M$ delays for pipelining each of the $N\log_2 M$ multipliers by M stages
- In the decomposed realization, the 1st stage implements an N-th order non-recursive section, and the subsequent stages respectively implement 2N, 4N,…, NM/2-order non-recursive sections

- Example (Example 10.4.7, p.334) Consider a 2nd-order recursive filter described by

$$H(z)=\frac{Y(z)}{U(z)}=\frac{b_0+b_1z^{-1}+b_2z^{-2}}{1-2r\cos q\cdot z^{-1}+r^2z^{-2}}$$

  - The poles of the system are located at $\left\{z=re^{\pm jq}\right\}$ (see Fig. 7(a)).
  - The pipelined filter requires $\left(2\log_2 M+5\right)$ multiplications and is described by $\left\{where\quad q\neq 2p/M\right\}$

$$H(z)=\frac{\sum_{i=0}^{2}b_iz^{-i}}{1-2r^M\cos Mq\cdot z^{-M}+r^{2M}z^{-2M}}\times$$
$$\prod_{i=0}^{\log_2 M-1}\left(1+2r^{2^i}\cos 2^iq\cdot z^{-2^i}+r^{2^{i+1}}z^{-2^{i+1}}\right)$$

  - The 2M poles of the transformed transfer function (shown in Fig. 7(b)) are located at

$$z=re^{\pm j\left(q+i\left(2p/M\right)\right)},\quad i=0,1,2\cdots,\left(M-1\right)$$

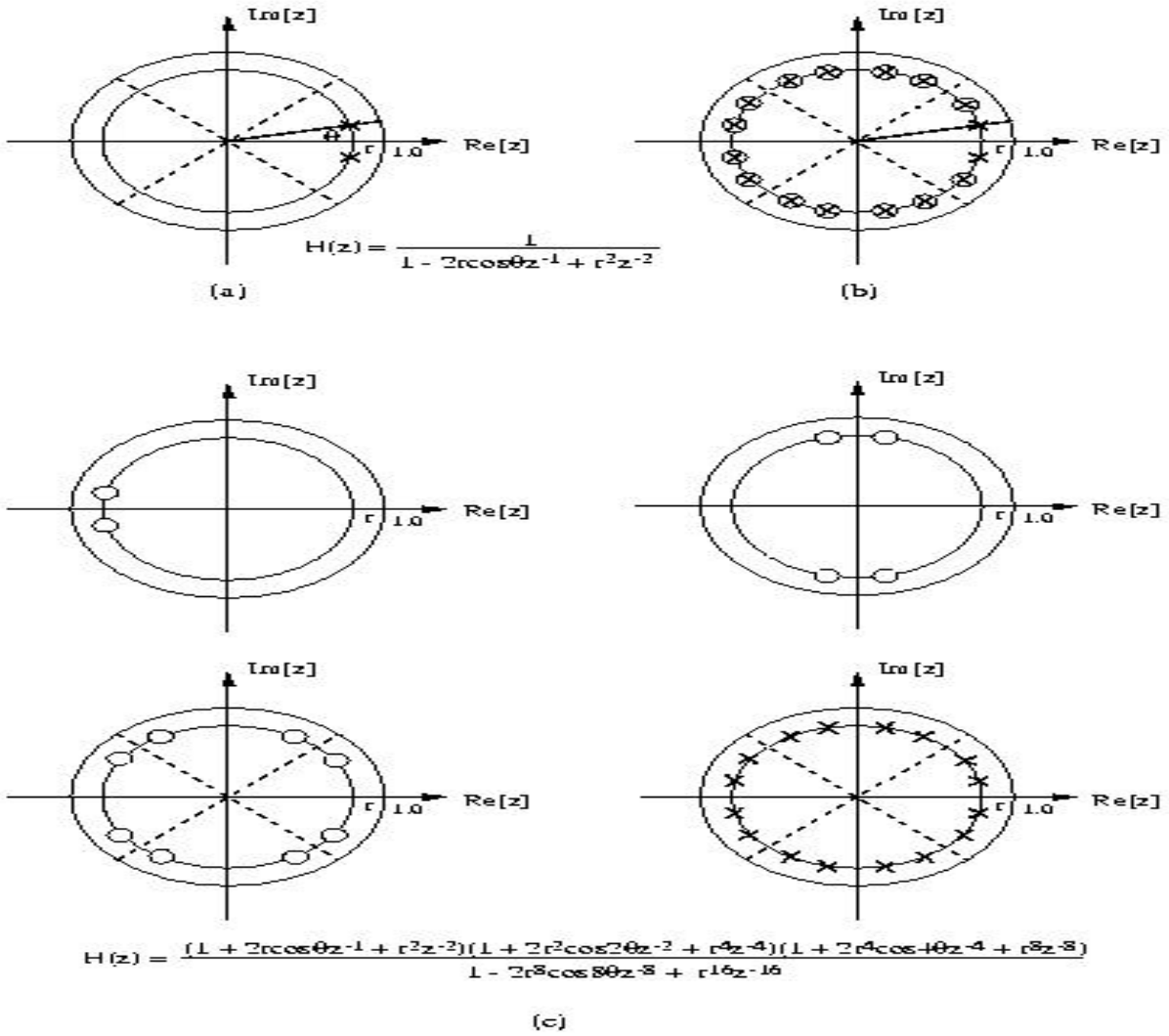


Fig. 7: (also see Fig.10.11, p.336) Pole-zero representation for Example 10.4.7, the decompositions of the zeros is shown in (c)

Parallel Processing in IIR Filters

- Parallel processing can also be used in design of IIR filters
- First discuss parallel processing for a simple 1st-order IIR filter, then we discuss higher order filters
- Example: (Example 10.5.1, p.339) Consider the transfer function of a 1st-order IIR filter given by

$$H(z) = \frac{z^{-1}}{1 - az^{-1}}$$

  - where $|a| \leq 1$ for stability. This filter has only 1 pole located at $z = a$. The corresponding input-output can be written as $y(n+1)=ay(n)+u(n)$
  - Consider the design of a 4-parallel architecture (L=4) for the foregoing filter. Note that in the parallel system, each delay element is referred to as a block delay, where the clock period of the block system is 4 times the sample period. Therefore, the loop update equation should update $y(n+4)$ by using inputs and $y(n)$.

– By iterating the recursion (or by applying look-ahead technique), we get

$$y(n+4) = a^4 y(n) + a^3 u(n) + a^2 u(n+1) + au(n+2) + u(n+3) \qquad (10.20)$$

– Substituting $n = 4k$

$$y(4k+4) = a^4 y(4k) + a^3 u(4k) + a^2 u(4k+1) + au(4k+2) + u(4k+3)$$

– The corresponding architecture is shown in Fig.8.

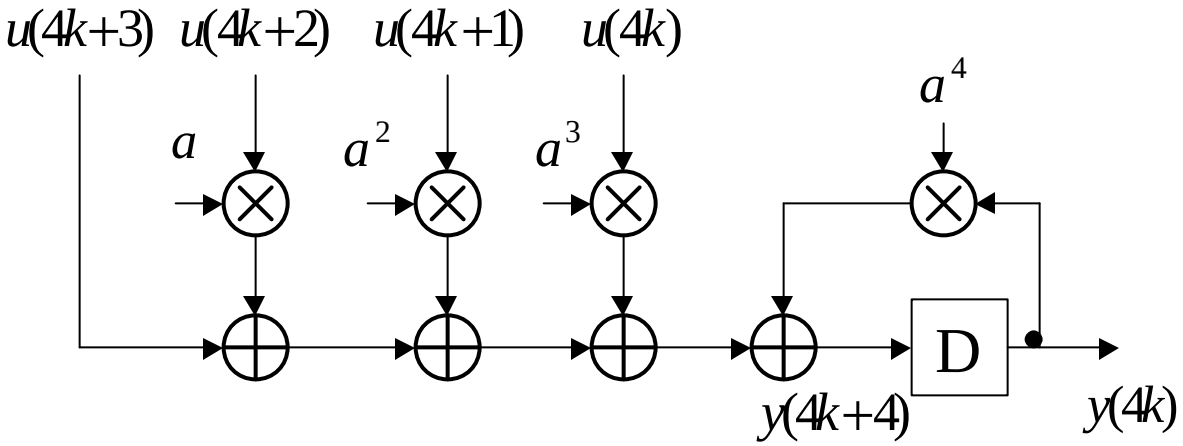


Fig. 8: (also see Fig.10.14, p. 340)

- The pole of the original filter is at $z = a$, whereas the pole for the parallel system is at $z = a^4$, which is much closer to the origin since

$$\left\{ \left| a^4 \right| \le |a|, \quad \text{since} \quad |a| \le 1 \right\}$$

- An important implication of this pole movement is the improved robustness of the system to the round-off noise
- A straightforward block processing structure for L=4 obtained by substituting n=4k+4, 4k+5, 4k+6 and 4k+7 in (10.20) is shown in Fig. 9.
- Hardware complexity of this architecture: L^2 multiply-add operations (Because L multiply-add operations are required for each output and there are L outputs in total)
- The square increase in hardware complexity can be reduced by exploiting the concurrency in the computation (the decomposition property in the scattered look-ahead mode can not be exploited in the block processing mode because one hardware delay element represents L sample delays)

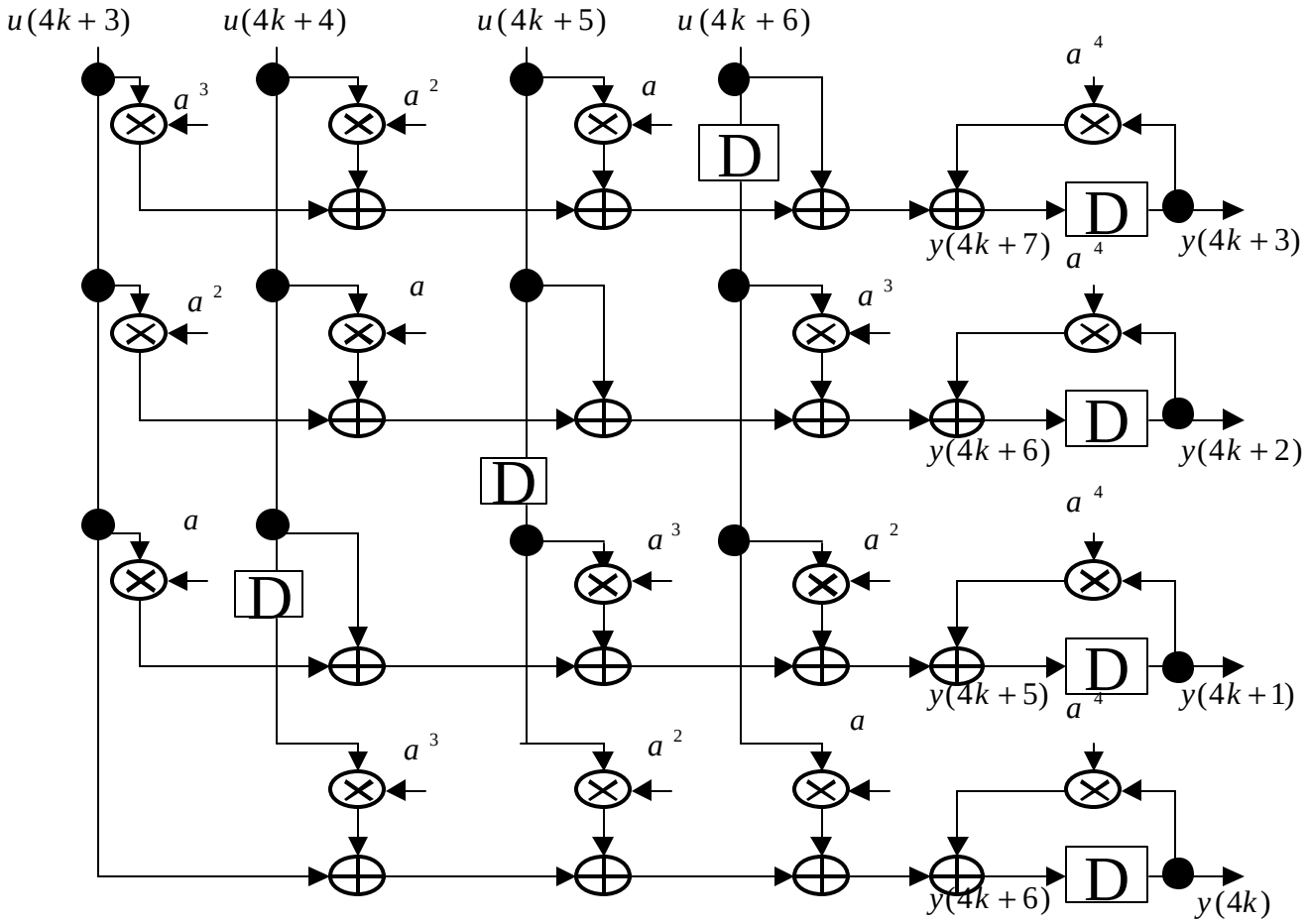


Fig. 9: (also Fig.10.15, p.341) A 4-parallel 1st-order recursive filter

- Trick:
  - Instead of computing *y(4k+1)*, *y(4k+2)* & *y(4k+3)* independently, we can use *y(4k)* to compute *y(4k+1)*, use *y(4k+1)* to compute *y(4k+2)*, and use *y(4k+2)* to compute *y(4k+3)*, at the expense of an increase in the system latency, which leads to a significant reduction in hardware complexity.
  - This method is referred as *incremental block processing*, and *y(4k+1)*, *y(4k+2) and y(4k+3)* are computed *incrementally*.
- Example (Example 10.5.2, p.341) Consider the same 1st-order filter in last example. To derive its 4-parallel filter structure with the minimum hardware complexity instead of simply repeating the hardware 4 times as in Fig.15, the incremental computation technique can be used to reduce hardware complexity
  - First, design the circuit for computing *y(4k)* (same as Fig.14)
  - Then, derive *y(4k+1)* from *y(4k)*, *y(4k+2)* from *y(4k+1)*, *y(4k+3)* from *y(4k+2)* by using

$$\begin{cases} y(4k+1) = ay(4k) + u(4k) \\ y(4k+2) = ay(4k+1) + u(4k+1) \\ y(4k+3) = ay(4k+2) + u(4k+2) \end{cases}$$

– The complete architecture is shown in Fig.10
– The hardware complexity has reduced from L^2 to *(2L-1)* at the expense of an increase in the computation time for *y(4k+1)*, *y(4k+2)* and *y(4k+3)*

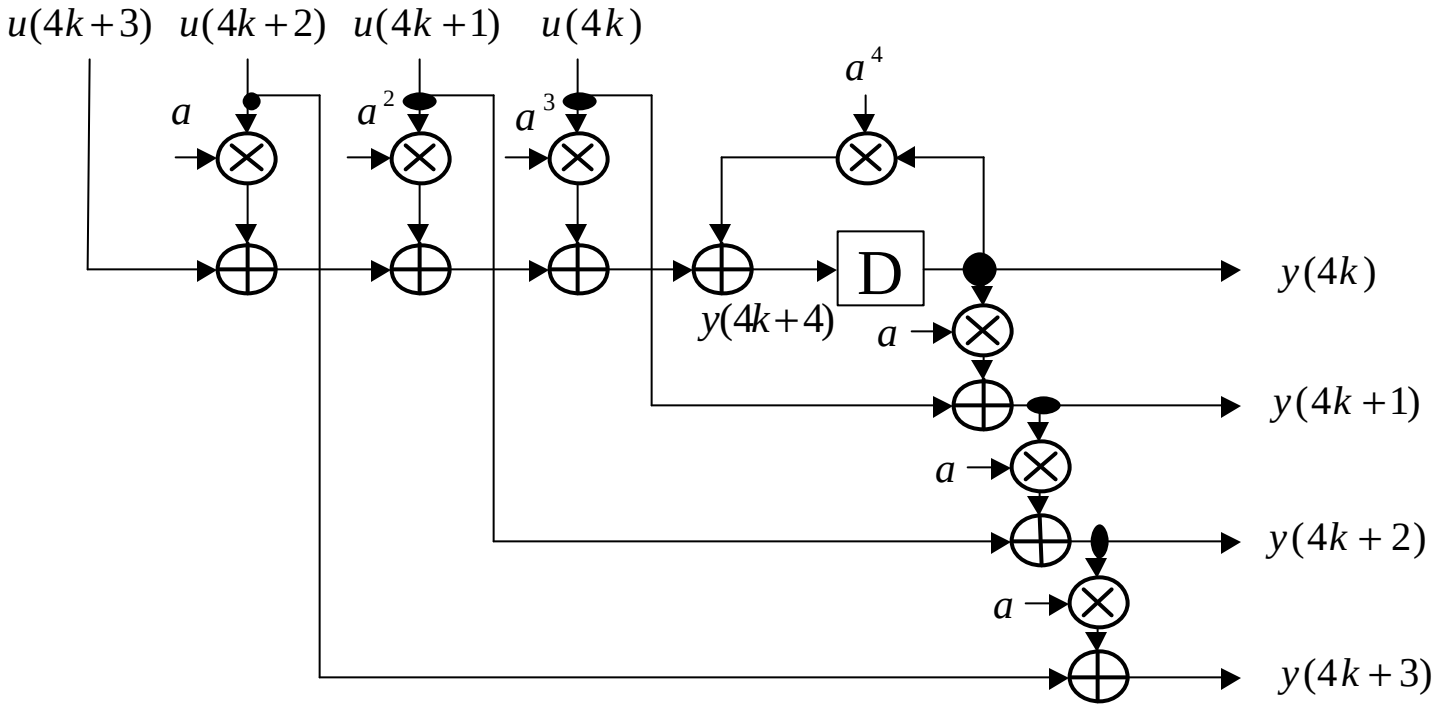


Fig.10: (also see Fig.10.16, p.342) Incremental block filter structure with L=4

- Example (Example 10.5.3, p.342) Consider a 2nd-order IIR filter described by the transfer function (10.21). Its pole-zero locations are shown in Fig.11. Derive a 3-parallel IIR filter where in every clock cycle 3 inputs are processed and 3 outputs are generated

$$H(z) = \frac{(1+z^{-1})^2}{1-\frac{5}{4}z^{-1}+\frac{3}{8}z^{-2}} \qquad (10.21)$$

  - Since the filter order is 2, 2 outputs need to be updated independently and the 3rd output can be computed incrementally outside the feedback loop using the 2 updated outputs. Assume that y(3k) and y(3k+1) are computed using loop update operations and y(3k+2) is computed incrementally. From the transfer function, we have:

$$\begin{aligned} y(n) &= \tfrac{5}{4}y(n-1) - \tfrac{3}{8}y(n-2) + f(n); \\ f(n) &= u(n) + 2u(n-1) + u(n-2) \end{aligned} \qquad (10.22)$$

  - The loop update process for the 3-parallel system is shown in Fig.12 where *y(3k+3)* and *y(3k+4)* are computed using *y(3k)* and *y(3k+1)*

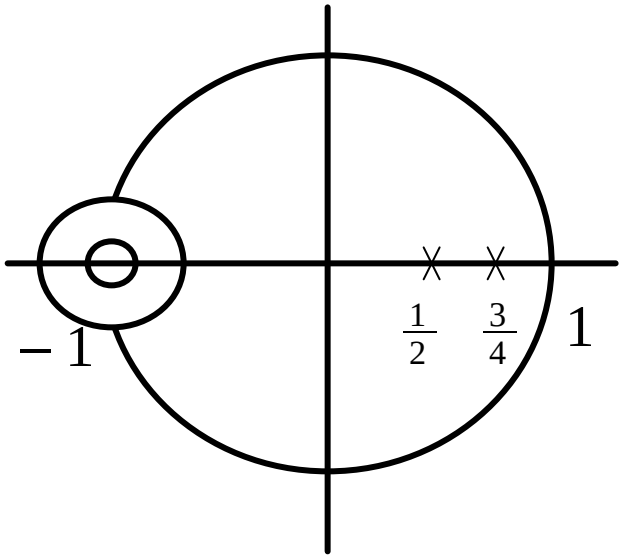


Fig.11: Pole-zero plots for the transfer function

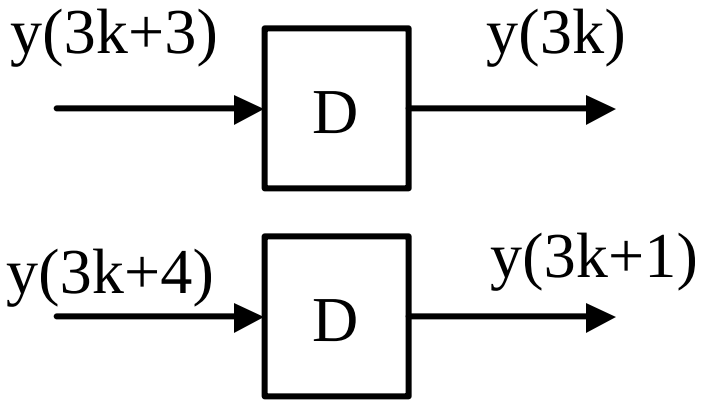


Fig.12: Loop update for block size=3

– The computation of *y(3k+3)* using *y(3k) & y(3k+1)* can be carried out if *y(n+3)* can be computed using *y(n) & y(n+1).* Similarly *y(3k+4)* can be computed using *y(3k) & y(3k+1)* if *y(n+4)* can be expressed in terms of *y(n) & y(n+1)* (see Fig.13). These state update operations correspond to clustered look-ahead operation for M=2 and 3 cases. The 2-stage and 3-stage clustered look-ahead equations are derived as:

$$
\begin{aligned}
y(n) &= \tfrac{5}{4}\, y(n-1) - \tfrac{3}{8}\, y(n-2) + f(n) \\
&= \tfrac{5}{4}\left[\tfrac{5}{4}\, y(n-2) - \tfrac{3}{8}\, y(n-3) + f(n-1)\right] - \tfrac{3}{8}\, y(n-2) \\
&\quad + f(n) \\
&= \tfrac{19}{16}\left[\tfrac{5}{4}\, y(n-3) - \tfrac{3}{8}\, y(n-4) + f(n-2)\right] - \tfrac{15}{32}\, y(n-3) \\
&\quad + \tfrac{5}{4}\, f(n-1) + f(n)
\end{aligned}
\qquad (10.23)
$$

– Substituting *n=3k+3 & n=3k+4* into (10.23), we have the following 2 loop update equations:

$$\begin{cases} y(3k+3) = \frac{19}{16}\, y(3k+1) - \frac{15}{32}\, y(3k) + \frac{5}{4}\, f(3k+2) \\ \qquad\qquad + f(3k+2) \\ y(3k+4) = \frac{65}{64}\, y(3k+1) - \frac{57}{128}\, y(3k) + \frac{19}{16}\, f(3k+2) \\ \qquad\qquad + \frac{5}{4}\, f(3k+3) + f(3k+4) \end{cases} \tag{10.24}$$

– The output y(3k+2) can be obtained incrementally as follows:

$$y(3k+2) = \frac{5}{4}\, y(3k+1) - \frac{8}{3}\, y(3k) + f(3k+2)$$

– The block structure is shown in Fig. 14

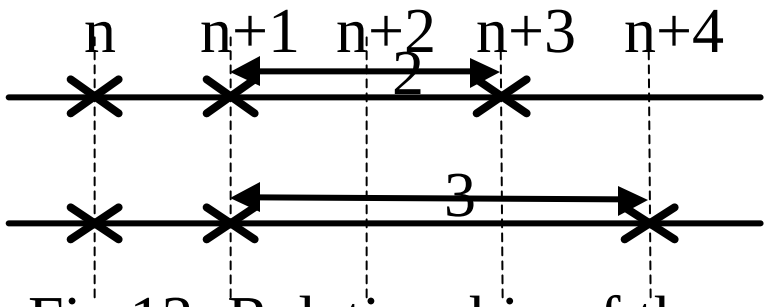


Fig.13: Relationship of the recursive outputs

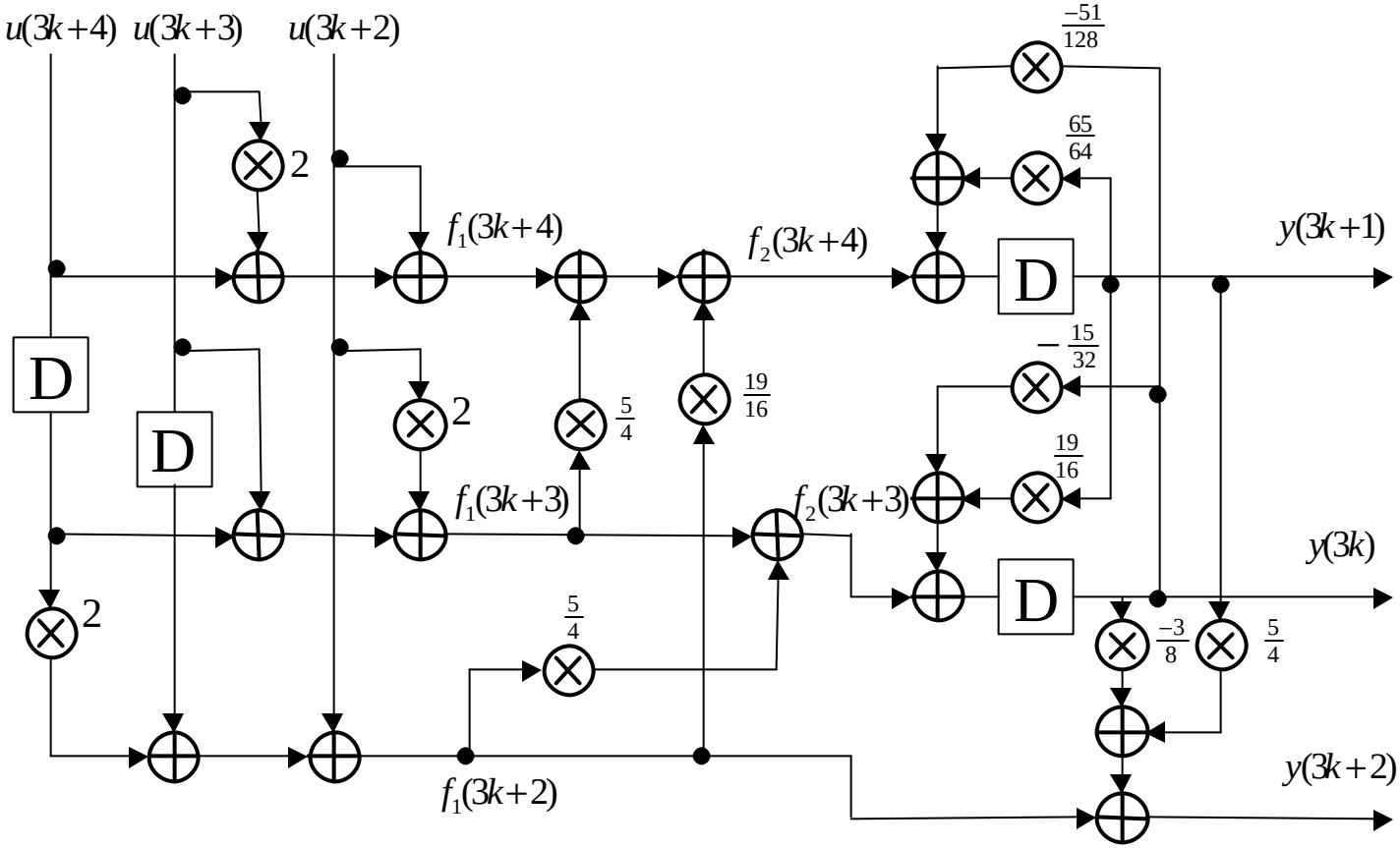


Fig. 14: Block structure of the 2nd-order IIR filter (L=3)
(also see Fig.10.20, p.344)

- Comments
  - The original sequential system has 2 poles at {1/2, 3/4}. Now consider the pole locations of the new parallel system. Rewrite the 2 state update equations in matrix form: ***Y(3k+3)=AY(3k)+F, i.e.***

$$\begin{bmatrix} y(3k+3) \\ y(3k+4) \end{bmatrix} = \begin{bmatrix} \frac{-15}{32} & \frac{19}{16} \\ \frac{-57}{128} & \frac{65}{64} \end{bmatrix} \cdot \begin{bmatrix} y(3k) \\ y(3k+1) \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \qquad (10.25)$$

  - The eigenvalues of system matrix A are $\left(\frac{1}{2}\right)^3$, $\left(\frac{3}{4}\right)^3$, which are the poles of the new parallel system. Thus, the parallel system is more stable. **Note**: the parallel system has the same number of poles as the original system
  - For a 2nd-order IIR filter (N=2), there are total *3L+[(L-2)+(L-1)]+4+2(L-2)=7L-3* multiplications, **(**the numerator part — *3L*; the overhead of loop update — *[(L-2)+(L-1)];* the loop multiplications — *4*; the incremental computation — *2(L-2)***)**. The multiplication complexity is linear function of block size L. This multiplication complexity can be further reduced by using fast parallel filter structures and substructure sharing for the incrementally-computed outputs

Combined Pipelining and Parallel Processing For IIR Filters

- Pipelining and parallel processing can also be combined for IIR filters to achieve a speedup in sample rate by a factor L×M, where L denotes the levels of block processing and M denotes stages of pipelining, or to achieve power reduction at the same speed
- Example (Example 10.6.1, p.345) Consider the 1st-order IIR with transfer function (10.26). Derive the filter structure with 4-level pipelining and 3-level block processing (i.e., M=4, L=3)

$$H(z) = \frac{1}{1 - a \cdot z^{-1}} \qquad (10.26)$$

  - Because the filter order is 1, only 1 loop update operation is required.The other 3 outputs can be computed incrementally.

– Since pipelining level *M=4*, the loop must contain 4 delay elements (shown in Fig.15). Since the block size *L=3*, each delay element represents a block delay (corresponds to 3 sample delays). Therefore, *y(3k+12)* needs to be expressed in terms of *y(3k)* and inputs (see Fig. 15).

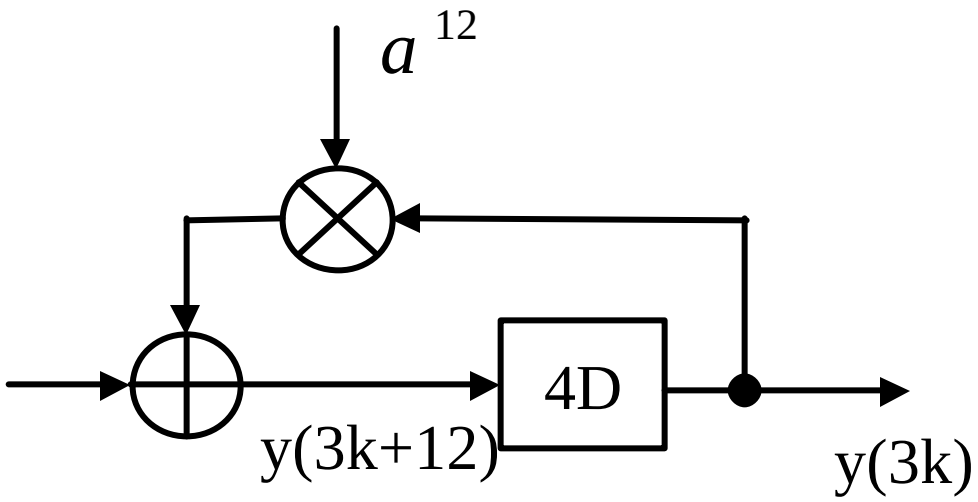


Fig.15: Loop update for the pipelined block system
(also see Fig.10.21, p. 346)

$$y(n) = ay(n-1) + u(n)$$
$$= \cdots\cdots$$
$$= a^{12} y(n-12) + a^{11} u(n-11) + \cdots\cdots + u(n)$$

– Substituting *n=3k+12*, we get:

$$y(3k+12) = a^{12} y(3k) + a^{11} u(3k+1) + \cdots\cdots + u(3k+12)$$

where
$$= a^{12} y(3k) + a^{6} f_2(3k+6) + a^{3} f_1(3k+9) + f_1(3k+12)$$

$$\begin{cases} f_1(3k+12) = a^2 u(3k+10) + au(3k+11) + u(3k+12) \\ f_2(3k+12) = a^3 f_1(3k+9) + f_1(3k+12) \end{cases}$$

– Finally, we have:

$$\begin{cases} y(3k+12) = a^{12} y(3k) + a^{6} f_2(3k+6) + a^{3} f_1(3k+9) + f_1(3k+12) \\ y(3k+1) = ay(3k) + u(3k+1) \\ y(3k+2) = ay(3k+1) + u(3k+2) \end{cases} \quad (10.27)$$

– The parallel-pipelined filter structure is shown in Fig. 16

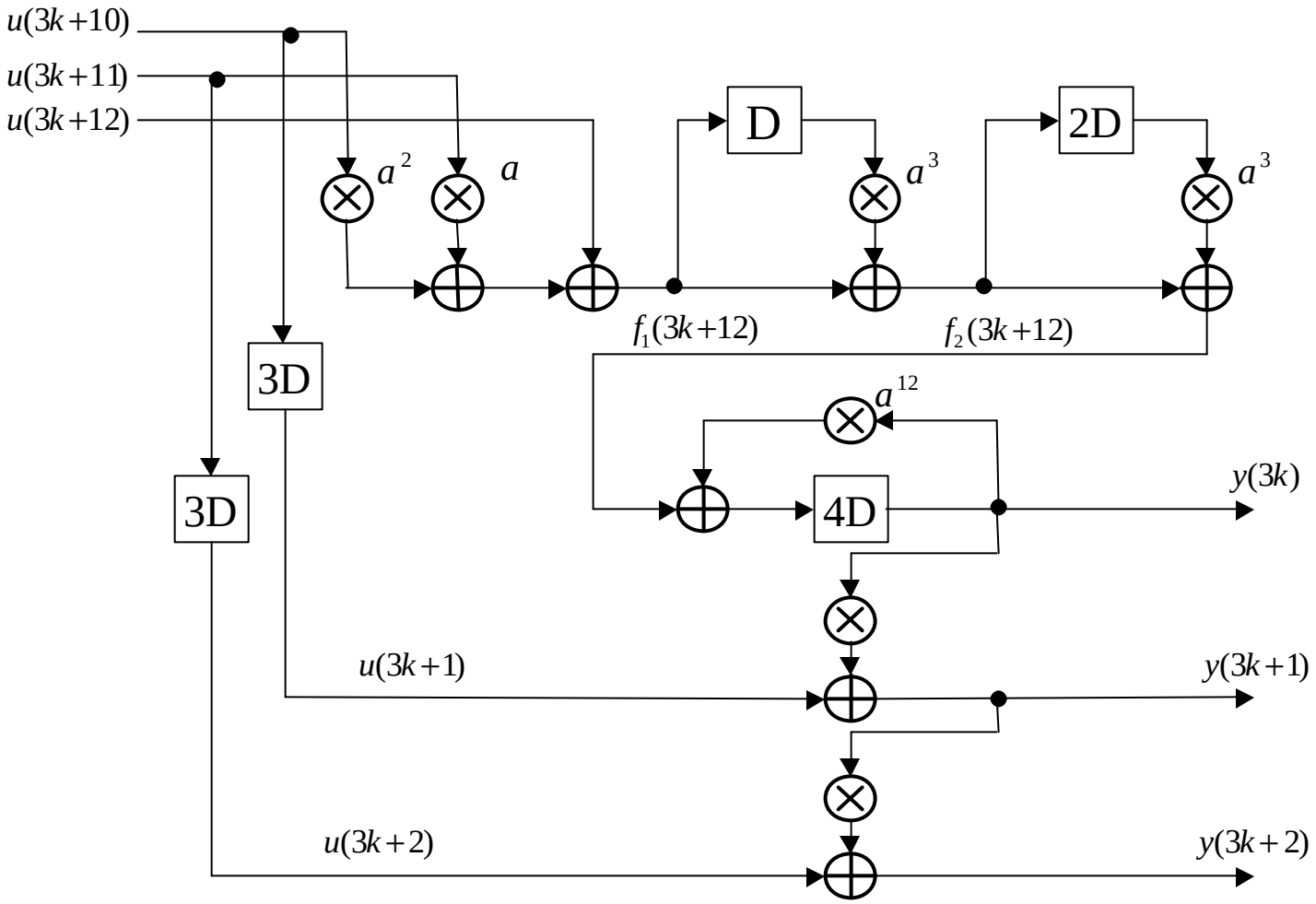


Fig. 16: The filter structure of the pipelined block system with L=3 & M=4 (also see Fig.10.22, p.347)

- Comments
  - The parallel-pipelined filter has 4 poles: $\left(a^3, -a^3, ja^3, -ja^3\right)$. Since the pipelining level is 4 and the filter order is 1, there are total 4 poles in the new system, which are separated by the same angular distance. Since the block size is 3, the distance of the poles from the origin is $\left|a^3\right|$.
  - Note: The decomposition method is used here in the pipelining phase.
  - The multiplication complexity (assuming the pipelining level M to be power of 2) can be calculated as (10.28), which is linear with respect to L, and logarithmic with respect to M:

$$(L-1)+\log_2 M + 1 + (L-1) = 2L - 1 + \log_2 M \qquad (10.28)$$

- Example (Example 10.6.2, p. 347) Consider the 2nd-order filter in Example 10.5.3 again, design a pipelined-block system for *L=3 and M=2*

$$\begin{aligned} y(n) &= \tfrac{5}{4} y(n-1) - \tfrac{3}{8} y(n-2) + f(n); \\ f(n) &= u(n) + 2u(n-1) + u(n-2) \end{aligned} \qquad (10.29)$$

- A method similar to clustered look-ahead can be used to update y(3k+6) and y(3k+7) using y(3k) and y(3k+1). Then by index substitution, the final system of equations can be derived.
- Suppose the system update matrix is A. Since the poles of the original system are $\left(\frac{1}{2}, \frac{3}{4}\right)$, the eigenvalues of A can be verified to be $\left(\frac{1}{2}\right)^6, \left(\frac{3}{4}\right)^6$
- The poles of the new parallel-pipelined second-order filter are the square roots of eigenvalues of A, i.e., $\left(\frac{1}{2}\right)^3, -\left(\frac{1}{2}\right)^3, \left(\frac{3}{4}\right)^3, -\left(\frac{3}{4}\right)^3$

• Comments: In general, the systematic approach below can be used to compute the pole location of the new parallel pipelined system:

- 1. Write the loop update equations using LM-level look-ahead, where M and L denote the level of pipelining and parallel processing, respectively.
- 2. Write the state space representation of the parallel pipelined filter, where state matrix A has dimension N×N and N is the filter order
- 3. Compute the eigenvalues λ_i of matrix A,
- 4. The NM poles of the new parallel-pipelined system correspond to the M-th roots of the eigenvalues of A, i.e.,

$$\left(\lambda_i\right)^{\frac{1}{M}} \quad 1 \le i \le N$$

Chapter 11: Scaling and Round-off Noise

Keshab K. Parhi

Outline

- Introduction
- Scaling and Round-off Noise
- State Variable Description of Digital Filters
- Scaling and Round-off Noise Computation
- Round-off Noise Computation Using State Variable Description
- Slow-Down, Retiming, and Pipelining

Introduction

- **In a fixed-point digital filter implementation, the overall input-output behavior is non-ideal. The quantization of signals and coefficients using finite word-lengths and propagation of roundoff noises to the output are the sources of noise.**
- **Other undesirable behavior include limit-cycle oscillations where undesirable periodic components are present at filter output even in the absence of any input. These may be caused due to internal rounding or overflow.**
- **Scaling is often used to constrain the dynamic range of the variables to a certain word-length**
- **State variable description of a linear filter: provides a mathematical formulation for studying various structures. These are most useful to compute quantities that depend on the internal structure of the filter. Power at each internal node and the output round-off noise of a digital FIR/IIR filter can be easily computed once the digital filter is described in state variable form**

Scaling and Round-off Noise

Scaling Operation

- Scaling: A process of readjusting certain internal gain parameters in order to constrain internal signals to a range appropriate to the hardware with the constraint that the transfer function from input to output should not be changed
- Illustration:
  - The filter in Fig.11.1(a) with unscaled node x has the transfer function
$$H(z) = D(z) + F(z)G(z) \qquad (11.1)$$
  - To scale the node x, we divide F(z) by some number β and multiply G(z) by the same number as in Fig.11.1(b). Although the transfer function does not change by this operation, the signal level at node x has been changed

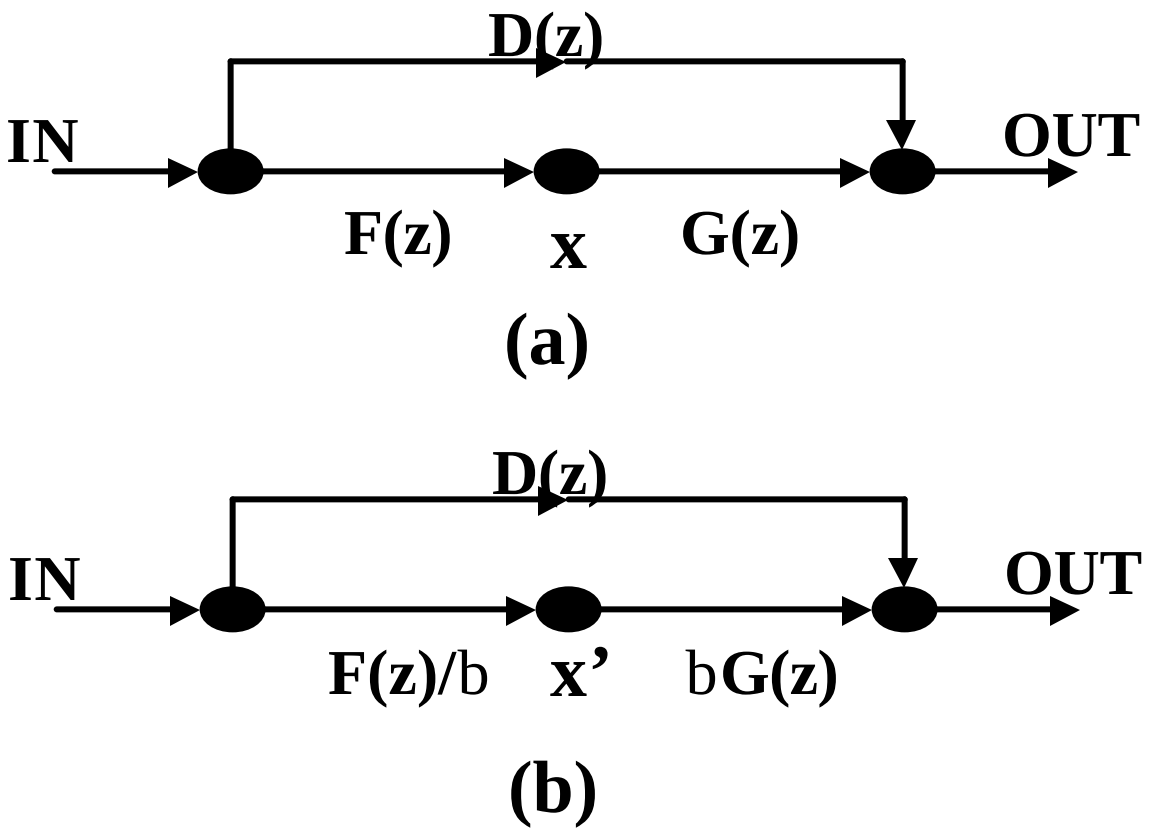


Fig.11.1 (a) A filter with unscaled node x, (b) A filter with scaled node x’

– The scaling parameter β can be chosen to meet any specific scaling rule such as

$$\begin{cases} l_1 - scaling: & b = \sum_{i=0}^{\infty} |f(i)|, \quad (11.2) \\ l_2 - scaling: & b = d\sqrt{\sum_{i=0}^{\infty} |f^2(i)|}, \quad (11.3) \end{cases}$$

  - where f(i) is the unit-sample response from input to the node x and the parameter δ can be interpreted to represent the number of standard deviations representable in the register at node x if input is unit-variance white noise

– If the input is bounded by $|u(n)| \le 1$, then

$$|x(n)| = \left|\sum_{i=0}^{\infty} f(i)u(n-i)\right| \le \sum_{i=0}^{\infty} |f(i)| \quad (11.4)$$

  - Equation (11.4) represents the true bound on the range of x and overflow is completely avoided by l_1 scaling in (11.2), which is the most stringent scaling policy

– Input can be generally assumed to be white noise.For unit-variance white noise input, variance at node ***x*** is given by:

$$E\left[x^2(n)\right] = \sum_{i=0}^{\infty} f^2(i) \qquad (11.5)$$

- l_2-scaling is commonly used because most input signals can be assumed to be white noise
- (11.5) is a variance (not a strict bound). So, we can increase δ in (11.3) to prevent possible overflow. But increasing δ will decrease SNR (signal-to-noise ratio). Thus, there is a trade-off between overflow and round-off noise

Scaling and Round-off Noise(cont'd)

Round-off Noise

- Round-off Noise: Product of two *W*-bit fixed-point fractions is a (2W-1) bit number. This product must eventually be quantized to W-bits by rounding or truncation, which results in round-off noise.
- Example:
  - Consider the 1st-order IIR filter shown in Fig. 11.2. Assume that the input wordlength W=8 bits, and the multiplier coefficient wordlength is also 8 bits. To maintain full precision in the output, we need to increase the output wordlength by 8 bits per iteration. This is clearly infeasible. Thus, the result needs to be rounded or truncated to its nearest 8-bit representation. This introduces a round-off noise e(n) (see Fig. 11.3).

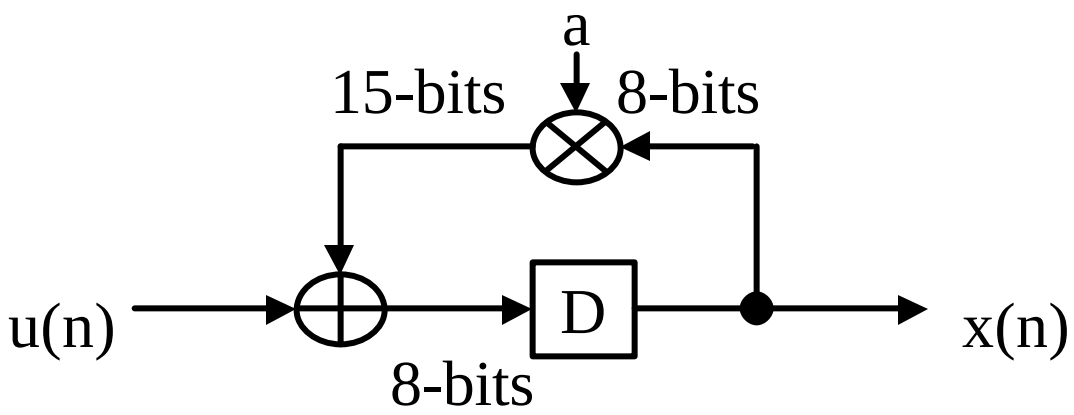


Fig.11.2 A 1ST-order IIR filter (W=8)

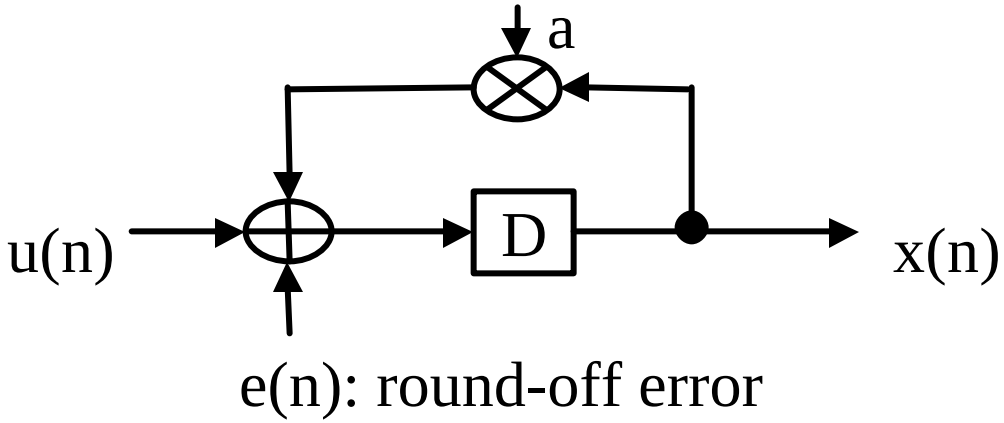


Fig.11.3 Model of Round-off Error

- Round-off Noise Mathematical Model: usually modeled as an infinite precision system with an external error input (see Fig.11.3)
- Rounding is a nonlinear operation, but its effect at the output can be analyzed using linear system theory with the following assumptions about e(n)
  - 1.e(n) is uniformly distributed white noise
  - 2. e(n) is a wide-sense stationary random process (mean & covariance of e(n) are independent of the time index n)
  - 3. e(n) is uncorrelated to all other signals such as input and other noise signals
- Let the wordlength of the output be W-bits, then the round-off error e(n) can be given by

$$-\frac{2^{-(W-1)}}{2} \le e(n) \le \frac{2^{-(W-1)}}{2} \qquad (11.6)$$

  - The error is assumed to be uniformly distributed over the interval in (11.6), the corresponding probability distribution is shown in Fig.11.4, where Δ is the length of the interval and $\Delta = 2^{-(W-1)}$

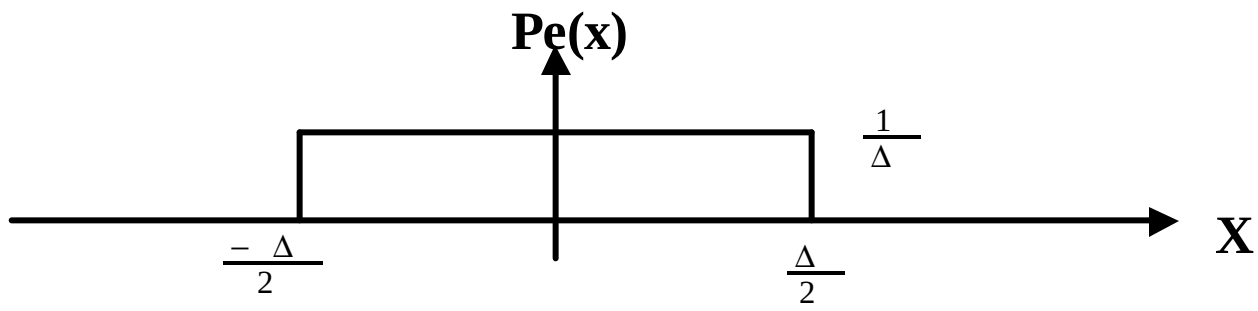


Fig.11.4 Error probability distribution

- The mean $E[e(n)]$ and variance $E[e^2(n)]$ of this error function:

$$\begin{cases} E[e(n)] = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} x P_e(x)dx = \frac{1}{\Delta}\frac{x^2}{2}\Big|_{\frac{-\Delta}{2}}^{\frac{\Delta}{2}} = 0 & (11.7) \\ E[e^2(n)] = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} x^2 P_e(x)dx = \frac{1}{\Delta}\frac{x^3}{3}\Big|_{\frac{-\Delta}{2}}^{\frac{\Delta}{2}} = \frac{\Delta^2}{12} = \frac{2^{-2W}}{3} & (11.8) \end{cases}$$

  - (11.8) can be rewritten as (11.9), where s_e^2 is the variance of the round-off error in a finite precision W-bit wordlength system

$$s_e^2 = 2^{-2W}/3 \qquad (11.9)$$

  - The variance is proportional to 2^{-2W}, so, increase in wordlength by 1 bit decreases the error by a factor of 4.
- Purpose of analyzing round-off noise: determine its effect at the output
  - If the noise variance at output is not negligible in comparison to the output signal level, the wordlength should be increased or some low-noise structure should be used.
  - We need to compute the SNR at the output, not just the noise gain to the output
  - In noise analysis, we use a double-length accumulator model: rounding is performed after two (2W-1)-bit products are added. Notice: multipliers are the sources for round-off noise

State Variable Description of Digital Filters

- Consider the signal flow graph (SFG) of an N-th order digital filter in Fig.11.5. We can represent it in the following recursive matrix form:

$$\begin{cases} \underline{x}(n+1) = \underline{\underline{A}} \cdot \underline{x}(n) + \underline{b} \cdot u(n), & (11.10) \\ y(n) = \underline{c}^{T} \cdot \underline{x}(n) + d \cdot u(n) & (11.11) \end{cases}$$

  - where $\underline{\boldsymbol{x}}$ is the state vector, $\boldsymbol{u}$ is the input, and $\boldsymbol{y}$ is the output of the filter; $\underline{\boldsymbol{x}}$, $\underline{\boldsymbol{b}}$ and $\underline{\boldsymbol{c}}$ are $\boldsymbol{N \times 1}$ column vectors; $\underline{\underline{A}}$ is $\boldsymbol{N \times N}$ matrix; $\boldsymbol{d}$, $\boldsymbol{u}$ and $\boldsymbol{y}$ are scalars.
- Let $\{f_i(n)\}$ be the unit-sample response from the input *u(n)* to the state $\underline{x}_i(n)$ and let $y_i(n)$ be the unit-sample response from the state $\underline{x}_i(n)$ to the output $\{g_i(n)\}$. It is necessary to scale the inputs to multipliers in order to avoid internal overflow

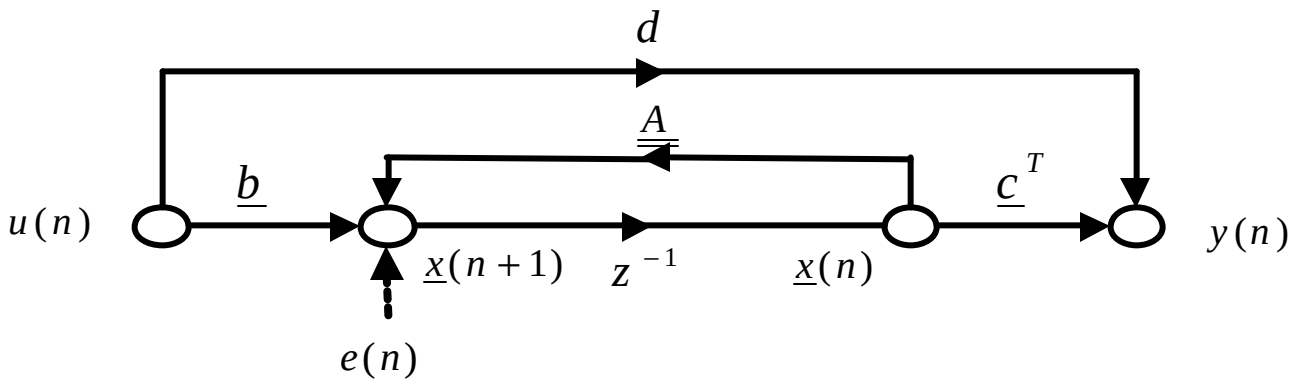


Fig.11.5 Signal flow graph of IIR filter

- Signals $\underline{x}(n)$ are input to the multipliers in Fig11.5. We need to compute $f(n)$ for scaling. Conversely, to find the noise variance at the output, it is necessary to find the unit-sample response from the location of the noise source $e(n)$ to $y(n)$. Thus g(n) represents the unit-sample response of the noise transfer function
- From the SFG of Fig.11.15, we can write:

$$\frac{\underline{X}(z)}{U(z)} = \frac{\underline{b} \cdot z^{-1}}{\underline{\underline{I}} - z^{-1} \cdot \underline{\underline{A}}} \qquad (11.12)$$

- Then, we can write the z-transform of ***f***(n), ***F***(z) as,

$$\underline{F}(z) = \underline{X}(z)/U(z) = (\underline{\underline{I}} + \underline{\underline{A}}z^{-1} + \underline{\underline{A}}^2 z^{-2} + \cdots)\underline{b}z^{-1}, \qquad (11.13)$$

$$\Rightarrow \underline{f}(n) = \underline{\underline{A}}^{n-1} \cdot \underline{b}, \quad n \geq 1. \qquad (11.14)$$

  - We can compute f(n) by substituting u(n) by δ(n) and using the recursion (11.15) and initial condition *f(0)=0:*

$$\underline{f}(n+1) = \underline{\underline{A}} \cdot \underline{f}(n) + \underline{b} \cdot d(n) \qquad (11.15)$$

  - The unit-sample response *g(n)* from the state ***x****(n)* to the output *y(n)* can be computed similarly with *u(n)=0.* The corresponding SFG is shown in Fig.11.6, which represents the following transfer function **G(z)**,

$$\underline{G}(z) = \frac{\underline{c}^T}{\underline{\underline{I}} - \underline{\underline{A}} \cdot z^{-1}}, \qquad (11.16)$$

$$\Rightarrow \underline{g}(n) = \underline{c}^T \cdot \underline{\underline{A}}^n, \quad n \geq 0 \qquad (11.17)$$

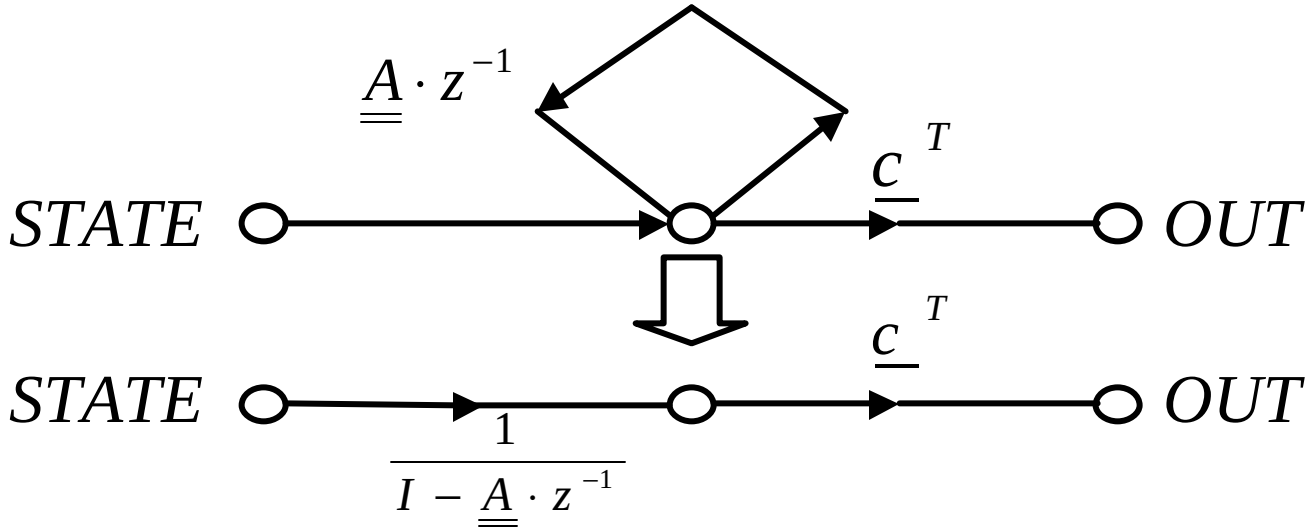


Fig.11.6 Signal flow graph of g(n)

- State covariance matrix **K**:

$$\underline{\underline{K}} \equiv E\left\{\underline{x}(n) \cdot \underline{x}^T(n)\right\} \qquad (11.18)$$

  - Because $\underline{X}$ is an N×1 vector, $\underline{K}$ is an N×N matrix
  - $\underline{K}$ is a measure of error power at various states (the diagonal element K_{ii} is the energy of the error signal at state x_i due to the input white noise)

- Express $\underline{K}$ in a form that reflects the error properties of the filter:
  - State vector $\underline{X}(n)$ can be obtained by the convolution of $u(n)$ and $f(n)$, by using (11.14) for $f(n)$, we get:

$$\underline{x}(n) = [x_1(n), x_2(n), \cdots, x_N(n)]^T \tag{11.19}$$

$$= \underline{f}(n) * u(n) = \sum_{l=0}^{\infty} \underline{\underline{A}}^l \underline{b} \cdot u(n-l-1) \tag{11.20}$$

  - Therefore

$$\underline{\underline{K}} = E\left\{ \sum_{l=0}^{\infty} (\underline{\underline{A}}^l \underline{b}) u(n-l-1) \sum_{m=0}^{\infty} u(n-m-1)(\underline{\underline{A}}^m \underline{b})^T \right\}$$

$$= E\left\{ \sum_{l=0}^{\infty} \sum_{m=0}^{\infty} \underline{\underline{A}}^l \underline{b} u(n-l-1) u(n-m-1)(\underline{\underline{A}}^m \underline{b})^T \right\}$$

$$= \sum_{l=0}^{\infty} \sum_{m=0}^{\infty} \underline{\underline{A}}^l \underline{b} E\left[u(n-l-1)u(n-m-1)\right](\underline{\underline{A}}^m \underline{b})^T \tag{11.21}$$

  - Assume u(n) is zero-mean unit-variance white noise, so we have:

$$\begin{cases} E[u^2(n)] = 1 & (11.22) \\ E[u(n)u(n-k)] = 0, \quad k \neq 0 & (11.23) \end{cases}$$

– Substituting (11.22) & (11.23) into (11.21), we obtain:

$$\underline{\underline{K}} = \sum_{l=0}^{\infty}\sum_{m=0}^{\infty} \underline{\underline{A}}^{l}\underline{b}\cdot d_{lm}\cdot(\underline{\underline{A}}^{m}\underline{b})^{T} = \sum_{l=0}^{\infty} f(l)f^{T}(l) = \sum_{l=0}^{\infty} \underline{\underline{A}}^{l}\underline{b}\cdot(\underline{\underline{A}}^{l}\underline{b})^{T}$$

$$= \underline{b}\underline{b}^{T} + \sum_{l=1}^{\infty} \underline{\underline{A}}^{l}\underline{b}\cdot(\underline{\underline{A}}^{l}\underline{b})^{T} = \underline{b}\underline{b}^{T} + \sum_{K=0}^{\infty} \underline{\underline{A}}^{K+1}\underline{b}\cdot(\underline{\underline{A}}^{K+1}\underline{b})^{T}$$

$$= \underline{b}\underline{b}^{T} + \sum_{K=0}^{\infty} \underline{\underline{A}}\left[\underline{\underline{A}}^{K}\underline{b}\cdot(\underline{\underline{A}}^{K}\underline{b})^{T}\right]\underline{\underline{A}}^{T} = \underline{b}\underline{b}^{T} + \underline{\underline{A}}\left[\sum_{K=0}^{\infty} \underline{\underline{A}}^{K}\underline{b}\cdot(\underline{\underline{A}}^{K}\underline{b})^{T}\right]\underline{\underline{A}}^{T} \quad (11.24)$$

– Finally, we get the Lyapunov equation:

$$\Rightarrow \underline{\underline{K}} = \underline{b}\cdot\underline{b}^{T} + \underline{\underline{A}}\cdot\underline{\underline{K}}\cdot\underline{\underline{A}}^{T} \quad (11.25)$$

- If for some state x_i, $E[x_i^2]$ has a higher value than other states, then x_i needs to be assigned more bits, which leads to extra hardware and irregular design.
  – By scaling, we can ensure that all nodes have equal power, and the same word-length can be assigned to all nodes.

- Orthogonal filter structure: All internal variables are uncorrelated and have unit variance assuming a white-noise input, it satisfies the following:

$$\underline{\underline{K}} = \underline{\underline{I}} = \underline{\underline{A}} \cdot \underline{\underline{A}}^T + \underline{b} \cdot \underline{b}^T \qquad (11.26)$$

- The advantages of orthogonal filter structure:
  - The scaling rule is automatically satisfied
  - The round-off noise gain is low and invariant under frequency transformations
  - Overflow oscillations are impossible
- Similarly, define the output covariance matrix $\underline{\underline{W}}$ as follows:

$$\underline{\underline{W}} = \sum_{n=0}^{\infty} \underline{g}^T(n)\underline{g}(n) = \sum_{n=0}^{\infty} (\underline{c}^T \underline{\underline{A}}^n)^T \underline{c}^T \underline{\underline{A}}^n \qquad (11.27)$$

- Proceeding in a similar manner as before, we can get

$$\underline{\underline{W}} = \underline{\underline{A}}^T \cdot \underline{\underline{W}} \cdot \underline{\underline{A}} + \underline{c} \cdot \underline{c}^T \qquad (11.28)$$

Scaling and Round-off Noise Computation

Scaling Operation

- The same word-length can be assigned to all the variables of the system only if all the states have equal power. This is achieved by scaling
- The state vector is pre-multiplied by inverse of the scaling matrix $\underline{\underline{T}}$.
  - If we denote the scaled states by $\underline{x}_S$, we can write,

$$\underline{x}_S(n) = \underline{\underline{T}}^{-1} \cdot \underline{x}(n) \quad \Rightarrow \quad \underline{x}(n) = \underline{\underline{T}} \cdot \underline{x}_S(n) \tag{11.29}$$

  - Substituting for $\underline{x}$ from (11.29) into (11.10) and solving for $\underline{x}_S$, we get

$$\underline{\underline{T}} \cdot \underline{x}_S(n+1) = \underline{\underline{A}} \cdot \underline{\underline{T}} \cdot \underline{x}_S(n) + \underline{b} \cdot u(n) \tag{11.30}$$

$$\Rightarrow \quad \underline{x}_S(n+1) = \underline{\underline{T}}^{-1} \cdot \underline{\underline{A}} \cdot \underline{\underline{T}} \cdot \underline{x}_S(n) + \underline{\underline{T}}^{-1} \cdot \underline{b} \cdot u(n) \tag{11.31}$$

$$\Rightarrow \quad \underline{x}_S(n+1) = \underline{\underline{A}}_S \cdot \underline{x}_S(n) + \underline{b}_S \cdot u(n) \tag{11.32}$$

– where $\left(\underline{\underline{A}}_S = \underline{\underline{T}}^{-1} \cdot \underline{\underline{A}} \cdot \underline{\underline{T}}, \quad \underline{b}_S = \underline{\underline{T}}^{-1} \cdot \underline{b}\right)$

- Similarly, the output equation (11.11) can be derived as follows

$$y(n) = \underline{c}^T \cdot \underline{\underline{T}} \cdot \underline{x}_S(n) + d \cdot u(n)$$

$$= \underline{c}_S^T \cdot \underline{x}_S(n) + d_S \cdot u(n)$$

$$\Rightarrow \left\{\underline{c}_S^T = \underline{c}^T \cdot \underline{\underline{T}}, \quad d_S = d\right\} \qquad (11.33)$$

- The scaled ***K*** matrix is given by

$$\underline{\underline{K}}_S = E[\underline{x}_S \cdot \underline{x}_S^T] = E[\underline{\underline{T}}^{-1} \underline{x} \cdot \underline{x}^T (\underline{\underline{T}}^{-1})^T] = \underline{\underline{T}}^{-1} E\left(\underline{x} \cdot \underline{x}^T\right) (\underline{\underline{T}}^{-1})^T$$

$$\Rightarrow \quad \underline{\underline{K}}_S = \underline{\underline{T}}^{-1} \cdot \underline{\underline{K}} \cdot (\underline{\underline{T}}^{-1})^T \qquad (11.34)$$

- It is desirable to have equal power at all states, so the transformation matrix ***T*** is chosen such that the ***Ks*** matrix of the scaled system has all diagonal entries as 1.

- Further assume T to be diagonal, i.e.,

$$\underline{\underline{T}} = diag\,[t_{11}, t_{22}, \cdots\cdots t_{NN}], \tag{11.35}$$

$$\Rightarrow \underline{\underline{T}}^{-1} = diag\,[\frac{1}{t_{11}}, \frac{1}{t_{22}}, \cdots\cdots \frac{1}{t_{NN}}] = (\underline{\underline{T}}^{-1})^{T} \tag{11.36}$$

  - From (11.34) and (11.35) and let $(K_S)_{ii} = 1$, we can obtain:

$$(K_S)_{ii} = \frac{K_{ii}}{t_{ii}^2} = 1, \tag{11.37}$$

$$\Rightarrow t_{ii} = \sqrt{K_{ii}} \tag{11.38}$$

- Conclusion: By choosing i-th diagonal entry in ***T*** to be equal to the square root of the i-th diagonal element of ***K*** matrix, all the states can be guaranteed to have equal unity power
- Example (Example 11.4.1,pp.387) Consider the unscaled 2nd-order filter shown in Fig.11.7, its state variable matrices are (see the next page):

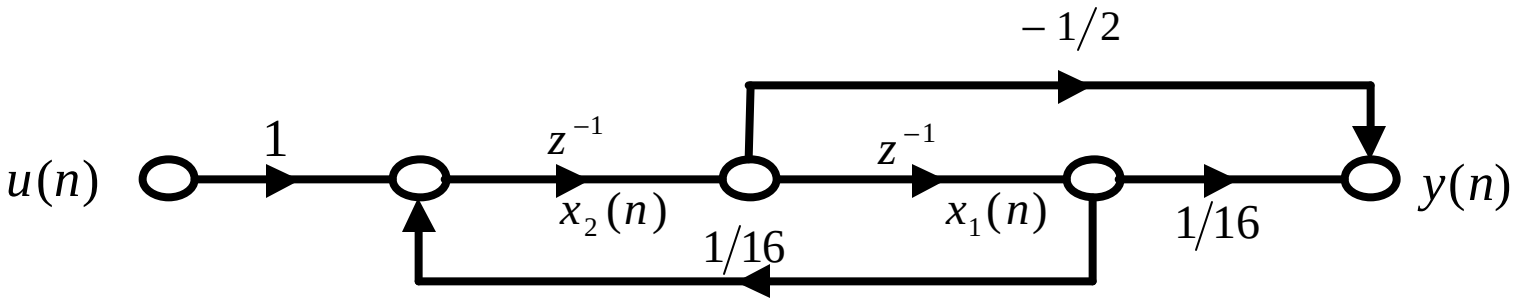


Fig.11.7 An SFG of an unscaled 2nd-order filter

– Example (cont'd)

$$\underline{\underline{A}} = \begin{bmatrix} 0 & 1 \\ \frac{1}{16} & 0 \end{bmatrix}, \quad \underline{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \underline{c} = \begin{bmatrix} \frac{1}{16} \\ -\frac{1}{2} \end{bmatrix}, \quad d = 0$$

– The state covariance matrix $\underline{\underline{\boldsymbol{K}}}$ can be computed using (11.25) as

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{1}{16} & 0 \end{bmatrix} \cdot \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \cdot \begin{bmatrix} 0 & \frac{1}{16} \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} K_{22} & \frac{1}{16} K_{21} \\ \frac{1}{16} K_{12} & \frac{1}{256} K_{11} + 1 \end{bmatrix}$$

– Thus, we get: $\left\{K_{11} = K_{22} = \frac{256}{255}, \quad K_{12} = K_{21} = 0\right\}$

– For l_2 scaling with δ=1, the transformation matrix is

$$\underline{\underline{T}} = \begin{bmatrix} \frac{16}{\sqrt{255}} & 0 \\ 0 & \frac{16}{\sqrt{255}} \end{bmatrix}$$

– Thus the scaled filter is described as below and is shown in Fig.11.8

$$\underline{\underline{A}}_S = \underline{\underline{T}}^{-1} \cdot \underline{\underline{A}} \cdot \underline{\underline{T}} = \begin{bmatrix} 0 & 1 \\ \frac{1}{16} & 0 \end{bmatrix}, \quad \underline{b}_S = \underline{\underline{T}}^{-1} \cdot \underline{b} = \begin{bmatrix} 0 \\ \frac{\sqrt{255}}{16} \end{bmatrix},$$

$$\underline{c}_S = \underline{\underline{T}}^{T} \cdot \underline{c} = \begin{bmatrix} \frac{1}{\sqrt{255}} \\ -\frac{8}{\sqrt{255}} \end{bmatrix}, \qquad d_S = 0$$

– Note: the state covariance matrix $\underline{\underline{\boldsymbol{K}}}s$ of the scaled filter is

$$\underline{\underline{K}}_S = \begin{bmatrix} \frac{\sqrt{255}}{16} & 0 \\ 0 & \frac{\sqrt{255}}{16} \end{bmatrix} \begin{bmatrix} \frac{256}{255} & 0 \\ 0 & \frac{256}{255} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{255}}{16} & 0 \\ 0 & \frac{\sqrt{255}}{16} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

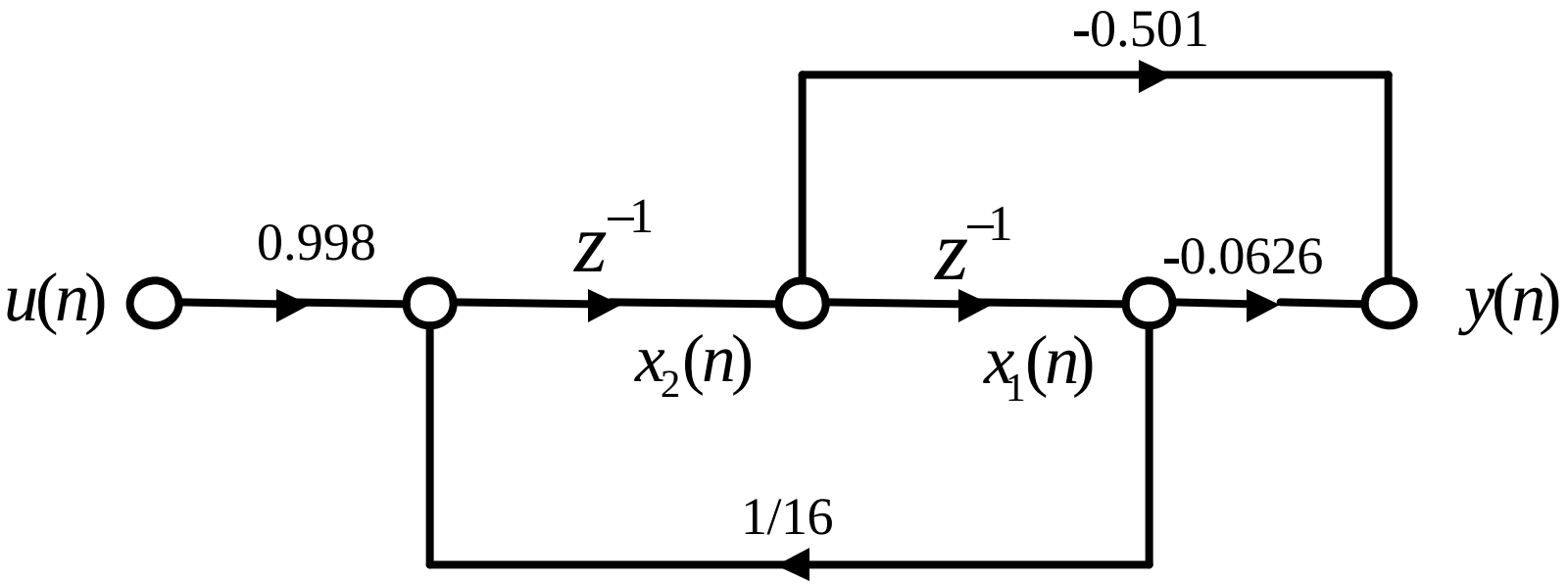


Fig.11.8 A SFG of a scaled 2nd-order filter

Scaling and Round-off Noise Computation (cont'd)

Round-off Noise Computation

- Computation: Let $e_i(n)$ be the error due to round-off at state x_i. Then the output round-off noise $y_i(n)$, due to this error, can be written as the convolution of the error input $e_i(n)$ with the state-to-output unit-sample response $g_i(n)$:

$$y_i(n) = e_i(n) * g_i(n) = \sum_{l=0}^{\infty} e_i(l) g_i(n-l) \qquad (11.39)$$

  - Consider the mean and the variance of $y_i(n)$. Since $e_i(n)$ is white noise with zero mean, so we have:

$$E[y_i(n)] = 0, \qquad (11.40)$$

$$E[y_i^2(n)] = E\left[\sum_l e_i(l) g_i(n-l) \sum_m e_i(m) g_i(n-m)\right]$$

$$= \sum_l \sum_m g_i(n-l) E[e_i(l) e_i(m)] g_i(n-m)$$

$$let \quad s_e^2 = E[e_i^2(n)] = \text{var iance}[e_i(n)]$$

– (cont'd) $$E\left[y_i^2(n)\right] = \sum_l \sum_m g_i(n-l)\, s_e^2 \cdot d_{lm}\, g_i(n-m)$$

$$= s_e^2 \sum_l g_i^2(n-l) = s_e^2 \sum_n g_i^2(n) \qquad (11.41)$$

– Expand $\underline{\underline{W}}$ in its explicit matrix form, we can observe that all its diagonal entries are of the form $\sum_n g_i^2(n)$:

$$\underline{\underline{W}} = \sum_n \underline{g}^T(n)\underline{g}(n) = \sum_n \begin{bmatrix} g_1(n), \\ \cdots \\ g_N(n) \end{bmatrix} \cdot \left[g_1(n), \quad \cdots, \quad g_N(n) \right] \qquad (11.42)$$

$$= \begin{bmatrix} \sum_n g_1^2(n) & \sum_n g_1(n)g_2(n) & \cdots & \sum_n g_1(n)g_N(n) \\ \sum_n g_2(n)g_1(n) & \sum_n g_2^2(n) & \cdots & \sum_n g_2(n)g_N(n) \\ \cdots & \cdots & \cdots & \cdots \\ \sum_n g_N(n)g_1(n) & \sum_n g_N(n)g_2(n) & \cdots & \sum_n g_N^2(n) \end{bmatrix} \qquad (11.43)$$

– Using (11.41), we can write the expression for the total output round-off noise in terms of trace of $\underline{\underline{W}}$:

$$total\_roundoff\_noise = s_e^2 \sum_{i=1}^{N} \sum_n g_i^2(n) = s_e^2 \sum_{i=1}^{N} W_{ii} = s_e^2 Trace(\underline{\underline{W}}) \quad (11.44)$$

– Note: (11.44) is valid for all cases. But when there is no round-off operation at any node, then the W_{ii} corresponding to that node should not be included while computing noise power

– (11.44) can be extended to compute the total round-off noise for the scaled system, which will simply be the trace of the scaled $\underline{\underline{W}}$ matrix:

$$\text{total round-off noise (scaled system)} = s_e^2 Trace(\underline{\underline{W}}_S) \quad (11.45)$$

– Replacing the filter parameters with the scaled parameters in (11.27), we can show:

$$\underline{\underline{W}}_S = \underline{\underline{T}}^T \cdot \underline{\underline{W}} \cdot \underline{\underline{T}} \quad (11.46)$$

– Also, for a diagonal $\underline{\underline{T}}$ we can write:

$$Trace\left(\underline{\underline{W}}_S\right) = \sum_{i=1}^{N} \left(W_S\right)_{ii} = \sum_{i=1}^{N} \left(t_{ii}^2 \cdot W_{ii}\right) \quad (11.47)$$

– (11.47) can be rewritten as follows because $t_{ii} \equiv \sqrt{K_{ii}}$

$$Trace\left(\underline{\underline{W}}_S\right) = \sum_{i=1}^{N}\left(K_{ii}\cdot W_{ii}\right) \quad \Rightarrow$$

total round-off noise (scaled system) $= s_e^2 \sum_{i=1}^{N}\left(K_{ii}\cdot W_{ii}\right)$ (11.48)

– **<u>Conclusion:</u>** The round-off noise of the scaled system can be computed using (11.48), i.e., using $\left\{K_{ii},\quad W_{ii}\right\}$

• Example (Example 11.4.2, p.390) To find the output round-off noise for the scaled filter in Fig.11.8, $\underline{\underline{\boldsymbol{W}}}$ can be calculated using (11.28) as

$$\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{16} \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ \frac{1}{16} & 0 \end{bmatrix} + \begin{bmatrix} \frac{1}{255} & \frac{-8}{255} \\ \frac{-8}{255} & \frac{64}{255} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{256}W_{22} + \frac{1}{255} & \frac{1}{16}W_{21} - \frac{8}{255} \\ \frac{1}{16}W_{12} - \frac{8}{255} & W_{11} + \frac{64}{255} \end{bmatrix}$$

– Thus $\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} = \begin{bmatrix} 0.0049 & -0.0332 \\ -0.0332 & 0.2559 \end{bmatrix}$

– The total output round-off noise for the scaled filter is

$$\left(W_{11}+W_{22}\right)\cdot s_e^2 = 0.2608 s_e^2$$

– For the unscaled filter in Fig.11.7:

$$\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{16} \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ \frac{1}{16} & 0 \end{bmatrix} + \begin{bmatrix} \frac{1}{256} & \frac{-1}{32} \\ \frac{-1}{32} & \frac{1}{4} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{256}W_{22} + \frac{1}{256} & \frac{1}{16}W_{21} - \frac{1}{32} \\ \frac{1}{16}W_{12} - \frac{1}{32} & W_{11} + \frac{1}{4} \end{bmatrix}$$

– Thus $\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} = \begin{bmatrix} 0.0049 & -0.0333 \\ -0.0333 & 0.2549 \end{bmatrix}$

– The total output round-off noise for the unscaled filter is

$$(W_{11}+W_{22})\cdot s_e^2 = 0.2598 s_e^2$$

– Notice: The scaled filter suffers from larger round-off noise, which can also be observed by comparing the unscaled and scaled filter structure:

$$\frac{roundoff\_noise(unscaled)}{roundoff\_noise(scaled)} = \frac{0.2598}{0.2608} < 1$$

- In the scaled filter, the input is scaled down by multiplying 0.998 to the input to avoid overflow (See Fig.11.8). Therefore, to keep the transfer functions the same in both filters, the output path of the scaled filter should have a gain which is 1/0.998 times the gain of the output path of the unscaled filter. Thus the round-off noise of the scaled filter is $1/0.998^2$ times that of the unscaled filter
- The above observation represents the tradeoff between overflow and round-off noise: More stringent scaling reduces the possibility of overflow but increases the effect of round-off noise

– Notice: (11.48) can be confirmed by:

$$(K_{11}W_{11}+K_{22}W_{22})_{unscaled} = \frac{256}{255}(0.0049+0.2549) = 0.2608 = (W_{11}+W_{22})_{scaled}$$

Round-off Noise Computation Using State Variable Description

Algorithms for Computing $\underline{K}$ and $\underline{W}$

- Parseval's relation and Cauchy's residue theorem are useful for finding signal power or round-off noise of digital filters. But, they are not useful for complex structures.
- The power at each internal node and the output round-off noise of a complex digital filter can be easily computed once the digital filter is described in state variable form
- Algorithm for computing $\underline{K}$
  - Using (11.24), K can be computed efficiently by the following algorithm: (see it on next page)

- Algorithm for computing $\underline{\underline{K}}$ (cont'd)
  - 1. Initialize: $\underline{\underline{F}} \leftarrow \underline{\underline{A}}, \quad \underline{\underline{K}} \leftarrow \underline{b} \cdot \underline{b}^T$
  - 2. Loop: $\underline{\underline{K}} \leftarrow \underline{\underline{F}} \cdot \underline{\underline{A}} \cdot \underline{\underline{F}}^T, \quad \underline{\underline{F}} \leftarrow \underline{\underline{F}}^2$
  - 3. Computation continues until $\underline{\underline{F}} = 0$
- Algorithm analysis:
  - After the 1st-loop iteration:

$$\begin{cases} \underline{\underline{K}} = \underline{\underline{A}} \cdot (\underline{b}\underline{b}^T) \cdot \underline{\underline{A}}^T + \underline{b}\underline{b}^T \\ \underline{\underline{F}} = \underline{\underline{A}}^2 \end{cases} \qquad (11.49)$$

  - After the 2nd-loop iteration:

$$\begin{cases} \underline{\underline{K}} = \underline{\underline{A}}^3 \underline{b}\underline{b}^T (\underline{\underline{A}}^3)^T + \underline{\underline{A}}^2 \underline{b}\underline{b}^T (\underline{\underline{A}}^2)^T + \underline{\underline{A}}\underline{b}\underline{b}^T \underline{\underline{A}}^T + \underline{b}\underline{b}^T, \\ \underline{\underline{F}} = \underline{\underline{A}}^4 \end{cases} \qquad (11.50)$$

- Thus, each iteration doubles the number of terms in the sum of (11.24). The above algorithm converges as long as the filter is stable (because the eigen-values of the matrix $\underline{\underline{\boldsymbol{A}}}$ are the poles of the transfer function)
- This algorithm can be used to compute $\underline{\underline{\boldsymbol{W}}}$ after some changes

- Algorithm for Computing $\underline{\underline{W}}$
  - 1. Initialize: $\underline{\underline{F}} \leftarrow \underline{\underline{A}}^T, \quad \underline{\underline{W}} \leftarrow \underline{c} \cdot \underline{c}^T$
  - 2. Loop: $\underline{\underline{W}} \leftarrow \underline{\underline{F}} \cdot \underline{\underline{W}} \cdot \underline{\underline{F}}^T, \quad \underline{\underline{F}} \leftarrow \underline{\underline{F}}^2$
  - 3. Computation continues until $\underline{\underline{F}} = 0$
- Example (Example 11.6.1, p.404) Consider the scaled-normalized lattice filter in Fig.11.9. We need to compute the signal powers at node 1, 2 and 3:
  - Because there are 3 states (1—3), the dimensions of the matrix $\underline{\underline{\boldsymbol{A}}}$, $\underline{\boldsymbol{b}}$, $\underline{\boldsymbol{c}}$ and $\boldsymbol{d}$ are 3×3, 3×1, 3×1, and 1×1, respectively. From Fig.11.9, the state equations can be written as (see next page)

$$\begin{cases} x_1(n+1) = 0.4944x_1(n) - 0.1915x_2(n) + 0.0443x_3(n) + 0.8467u(n), \\ x_2(n+1) = 0.3695x_1(n) + 0.9054x_2(n) - 0.2093x_3(n) \\ x_3(n+1) = 0.2252x_1(n) + 0.9743x_3(n) \\ y(n) = 0.0184x_1(n) + 0.1035x_2(n) + 0.3054x_3(n) + 0.0029u(n) \end{cases}$$

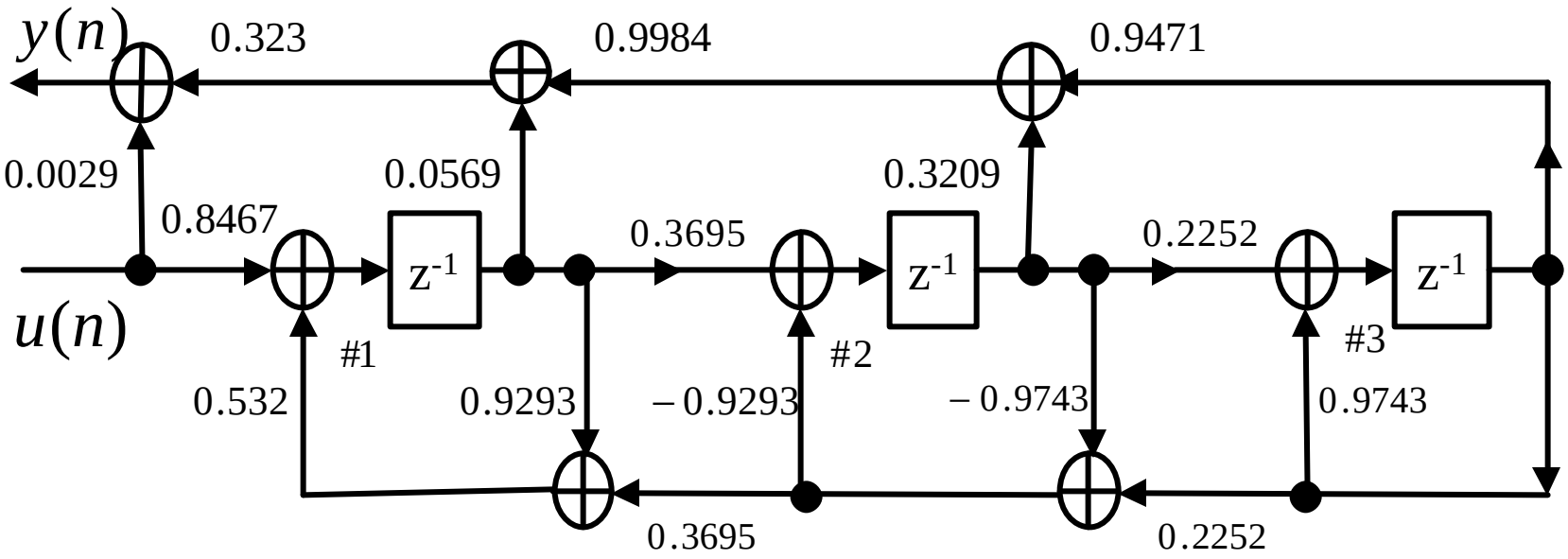


Fig.11.9 A 3rd-order scaled-normalized lattice filter
(also see Fig.11.18, p.403, Textbook)

- From these equations, matrices ***A***, ***b***, ***c*** and ***d*** can be obtained directly. By substituting them into the K-computing algorithm, we get

$$\underline{\underline{K}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Since $\{K_{11} = K_{22} = K_{33} = 1\}$, so no scaling is needed for nodes 1—3. In addition, the K matrix shows that the signals at nodes 1—3 are orthogonal to each other since all off-diagonal elements are zeros
- By the W-computing algorithm, we obtain:

$$\{W_{11} = 0.1455, W_{22} = 0.2952, W_{33} = 0.3096\}$$

- Conclusion:
  - Using state variable description method, we can compute signal power or round-off noise of a digital filter easily and directly. However, it can not be used on the nodes that are not connected to unit-delay branches because these nodes do not appear in the state variable description

Slow-Down, Retiming, and Pipelining

Introduction

- Many useful realizations contains roundoff nodes that are not connected to unit-delay branches. Thus these nodes (variables) do not appear in a state variable description and the scaling and roundoff noise computation methods can not be applied directly.
- The SRP (slow-down and retiming/pipelining) transformation technique can be used as a preprocessing step to overcome this difficulty
  - Slow-down: every delay element ($\mathbf{Z}$) in the original filter is changed into M delay element (Z^M)
  - Retiming and Pipelining (Please see Chapters 4 and 3 for details)

- Slow-down: Consider the filter in Fig.11.10(b) which is obtained by applying slow-down transformation (M=3) to the filter in Fig.11.10(a). By 3 slow down transformation, every *Z*-variable in Fig.11.10(a) is changed into Z^3. Thus the transfer function of the transformed filter H'(Z) is related to the original transfer function H(Z) as (11.51):

$$H'(z) = F'(z)G'(z) = F(z^3)G(z^3) = H(z^3) \qquad (11.51)$$

  - Thus, if the unit-sample response from the input to the internal node x in Fig.11.10(a) is defined by:

$$f(n) = \{f(0),\ f(1),\ f(2),\cdots\}, \qquad (11.52)$$

  - Then, the unit-sample response from the input to the internal node x' in Fig.11.10(b) is:

$$f'(n) = \{f(0),0,0,\ f(1),0,0,\ f(2),0,0,\cdots\}, \qquad (11.53)$$

  - We can get:

$$K'_{xx} = \sum_n [f'(n)]^2 = \sum_n f(n)^2 = K_{xx} \qquad (11.54)$$

  - Similarly it can be shown that:

$$W'_{xx} = W_{xx} \qquad (11.55)$$

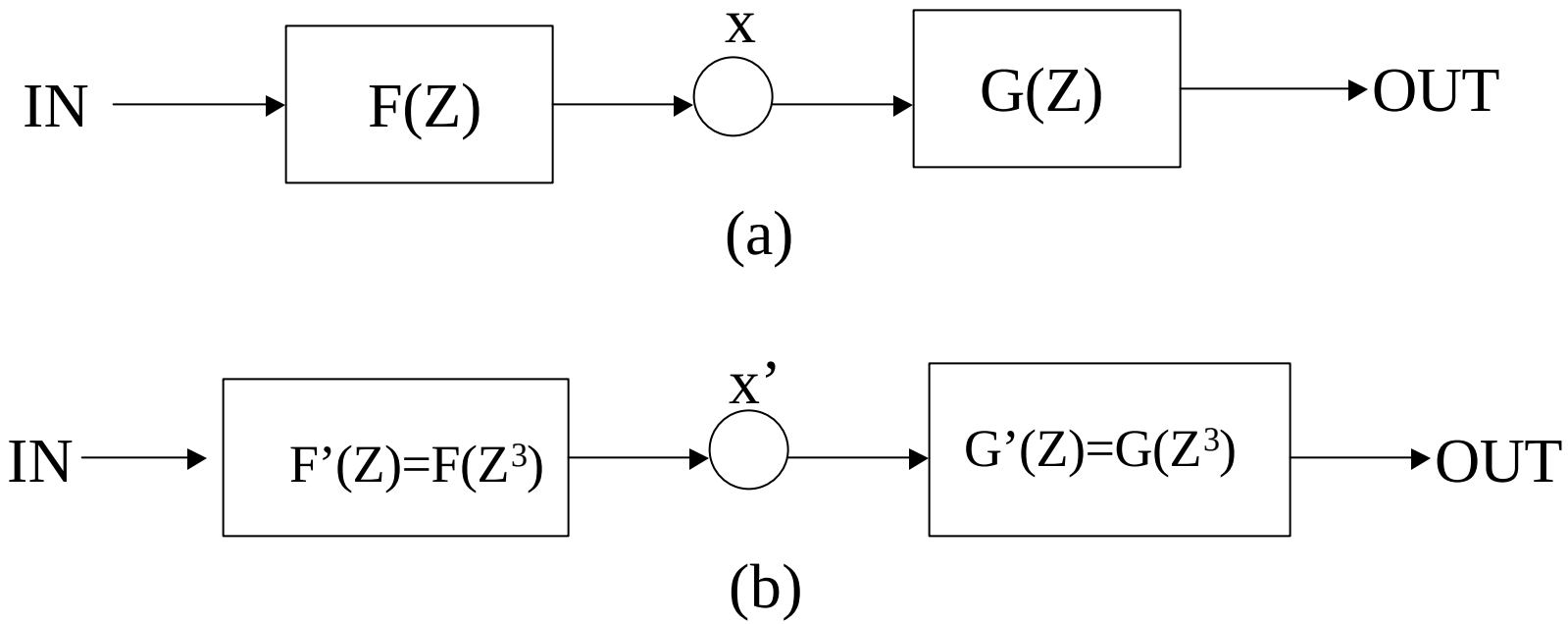


Figure 11.10 (a) A filter with transfer function H(z)=F(Z)G(Z). (b) Transformed filter obtained by 3 slow-down transformation H'(Z)=F(Z^3)G(Z^3).

- The foregoing analysis shows that slow-down transformation does not change the finite word-length behavior

- Pipelining:
  - Consider the filter in Fig.11.11(a), which has a non-state variable node x on the feed-forward path. It is obvious that the non-state variable node cannot be converted into the state variable node by slow-down transformation.
  - However, since x is on the feed-forward path, a delay can be placed on a proper cut-set location as shown in Fig.11.11(b). This pipelining operation converts the non-state variable node x into state variable node. The output sequence of the pipelined filter is equal to that of the original filter except one clock cycle delay.
  - So, the pipelined filter undergoes the same possibility of overflow and the same effect of round-off noise as in the original filter. Thus it is clear that pipelining does not change the filter finite word-length behavior

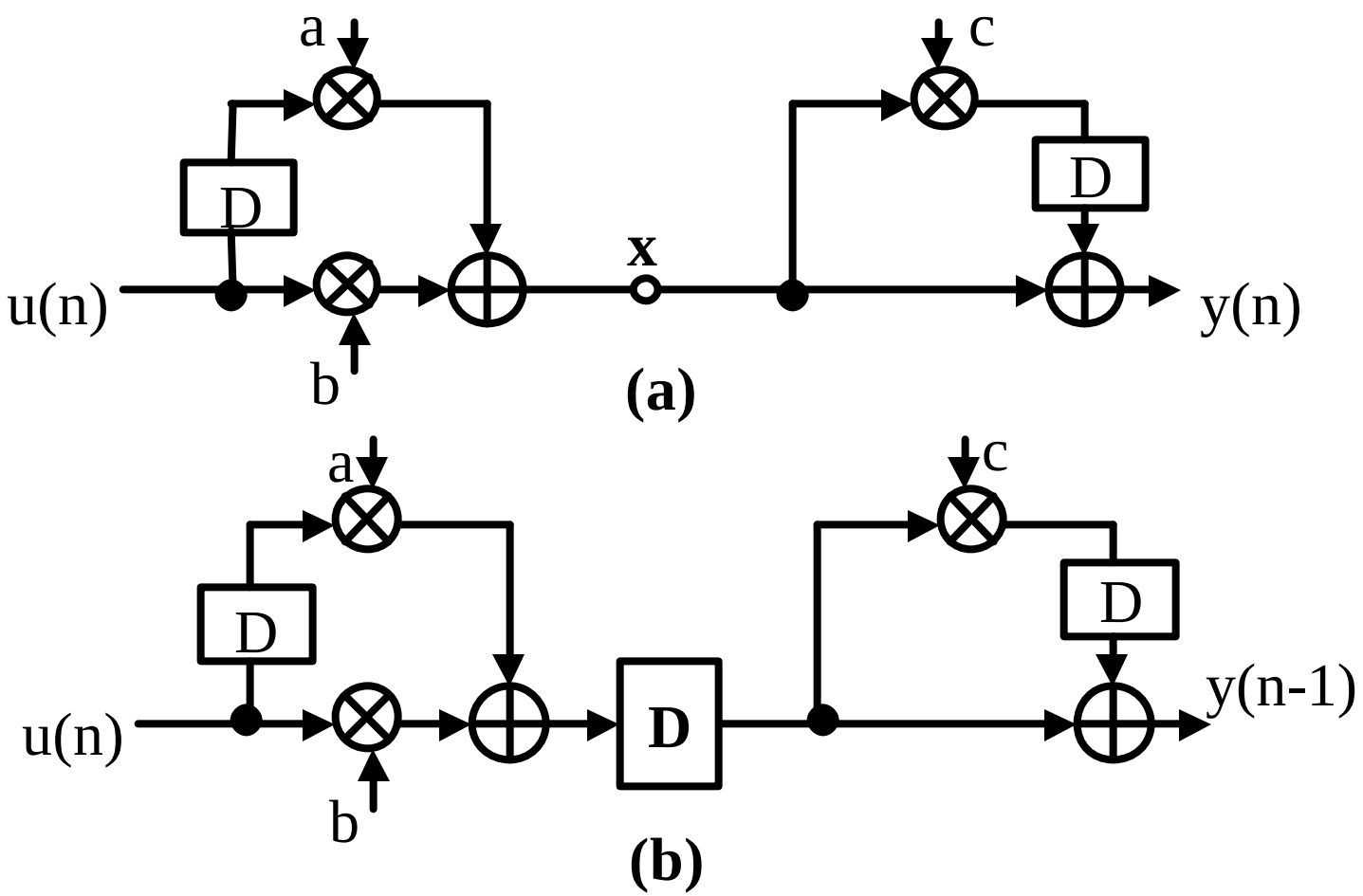


Fig.11.11 (a) A filter with a non-state variable node on a feed-forward path (b) Non-state variable node is converted into state variable node by pipelining

- Retiming:
  - In a linear array, if either all the left-directed or all the right-directed edges between modules carry at least 1 delay on each edge, the cut-set localization procedure can be applied to transfer some delays or a fraction of a delay to the opposite directed edges (see Chapter 4) — This is called **retiming**
- SRP transformation technique is summarized as follows:
  - 1. Apply slow-down transformation by a factor of M to a linear array, i.e., replace Z by Z^M. Also, apply pipelining technique to appropriate locations.
  - 2. Distribute the additional delays to proper locations such that non-state variable nodes are converted to state variable nodes
  - 3. Apply the scaling and noise computation method using state variable description
- Example (Example 11.7.1, p.407) Consider the filter shown in Fig.11.12, same as the 3rd-order scaled-normalized lattice filter in Fig.11.9 except that it has five more delays. The SFG in Fig.11.12 is obtained by using a 2-slow transformation and followed by retiming or cut-set transformation. (cont'd)

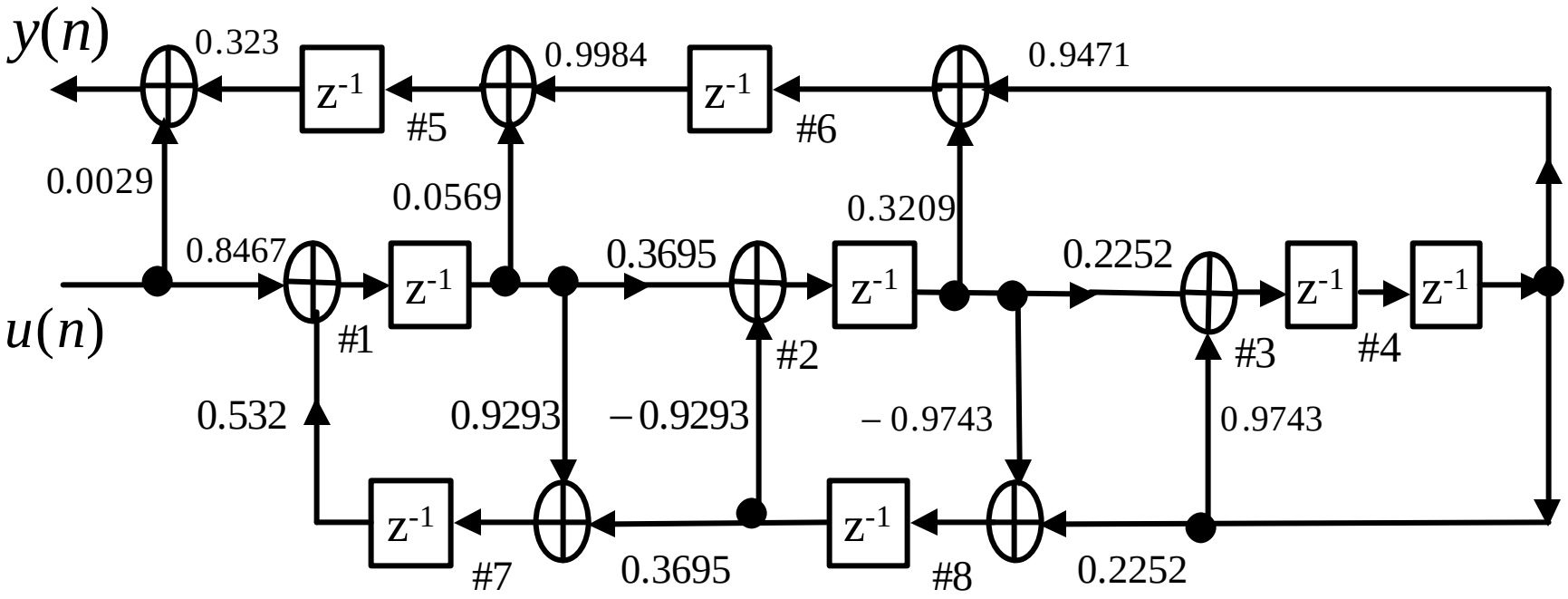


Fig.11.12 A transformed filter of the 3rd-order scaled-normalized lattice filter in Fig.11.9 (also see Fig.11.21,pp.407)

– (cont'd) Notice that signal power or round-off noise at every internal node in this filter can be computed using state variable description since each node is connected to a unit delay branch. Since there are 8 states, the dimensions of the matrices $\underline{\boldsymbol{A}}$, $\underline{\boldsymbol{b}}$, $\underline{\boldsymbol{c}}$, and d are 8×8, 8×1, 8×1, and 1×1, respectively. From Fig.11.12, state equations can be written as follows:

$$\begin{cases} x_1(n+1) = 0.532\,x_7(n) + 0.8467\,u(n), \\ x_2(n+1) = 0.3695\,x_1(n) - 0.9293\,x_8(n), \\ x_3(n+1) = 0.2252\,x_2(n) + 0.9743\,x_4(n), \\ x_4(n+1) = x_3(n), \\ x_5(n+1) = 0.0569\,x_1(n) + 0.9984\,x_6(n), \\ x_6(n+1) = 0.3209\,x_2(n) + 0.9471\,x_4(n), \\ x_7(n+1) = 0.9293\,x_1(n) + 0.3695\,x_8(n), \\ x_8(n+1) = -0.9743\,x_2(n) + 0.2252\,x_4(n), \\ y(n) = 0.323\,x_5(n) + 0.0029\,u(n) \end{cases}$$

- From the above equations, matrices ***A***, ***b***, ***c***, and d can be obtained directly. Using the K-computing algorithm, we obtain $\{K_{ii} = 1, \quad i = 1,2,\cdots,8\}$, which means that every internal node is perfectly scaled. Similarly, we get $W_{11},\cdots,W_{88} = \{0.1455, 0.2952, 0.3096, 0.3096, 0.1043, 0.104, 0.0412, 0.1912\}$
- Thus, the total output round-off noise is: $= s_e^2 \sum_i K_{ii} W_{ii} = 1.191\, s_e^2$
- Note: no round-off operation is associated with node 4 or state x_4. Therefore, W_{44} is not included in Trace(***W***) for round-off noise computation

- Example (omitted, study at home)
  - For details, please see Example 11.7.2, p.408 of textbook

Chapter 13: Bit Level Arithmetic Architectures

Keshab K. Parhi

- A W-bit fixed point two's complement number A is represented as :

$$A=a_{w-1}.a_{w-2}...a_1.a_0$$

where the bits ai, $0 \leq i \leq W-1$, are either 0 or 1, and the msb is the sign bit.

- The value of this number is in the range of $[-1, 1 - 2^{-W+1}]$ and is given by :

$$A = - a_{w-1} + \Sigma a_{w-1-i}2^{-i}$$

- For bit-serial implementations, constant word length multipliers are considered. For a W×W bit multiplication the W most-significant bits of the (2W-1)-bit product are retained.

- Parallel Multipliers :

$$A = a_{w-1}.a_{w-2}\ldots a_1.a_0 = -a_{w-1} + \sum_{i=1}^{W-1} a_{w-1-i}2^{-i}$$

$$B = b_{w-1}.b_{w-2}\ldots b_1.b_0 = -b_{w-1} + \sum_{i=1}^{W-1} b_{w-1-i}2^{-i}$$

Their product is given by :

$$P = -p_{2W-2} + \sum_{i=1}^{2W-2} p_{2W-2-i}2^{-i}$$

In constant word length multiplication, W – 1 lower order bits in the product P are ignored and the Product is denoted as $X \Leftarrow P = A \times B$, where

$$X = -x_{w-1} + \sum_{i=1}^{W-1} x_{w-1-i}2^{-i}$$

- Parallel Multiplication with Sign Extension :

  Using Horner's rule, multiplication of A and B can be written as

$$P = A \times (-b_{W-1} + \Sigma\, b_{W-1-i} 2^{-i})$$

$$= -A.\, b_{W-1} + [A.\, b_{W-2} + [A.\, b_{W-3} + [\ldots +$$

$$[A.\ b_1 + A\, b_0\, 2^{-1}]\, 2^{-1}]\ldots]2^{-1}]\, 2^{-1}$$

  where 2^{-1} denotes scaling operation.

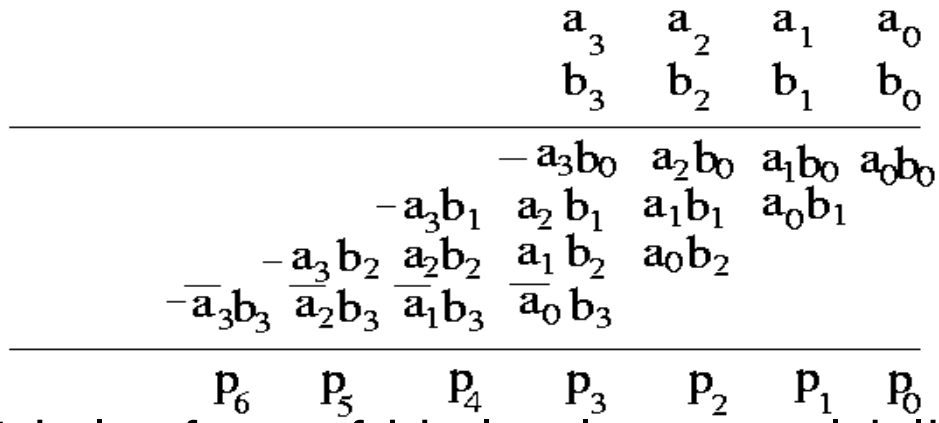
- In 2's complement, negating a number is equivalent to taking its 1's complement and adding 1 to lsb as shown below:

$$-A = a_{w-1} - \sum_{i=1}^{W-1} a_{w-1-i} 2^{-i}$$

$$= a_{w-1} + \sum_{i=1}^{W-1} (1 - a_{w-1-i}) 2^{-i} - \sum_{i=1}^{W-1} 2^{-i}$$

$$= a_{w-1} + \sum_{i=1}^{W-1} (1 - a_{w-1-i}) 2^{-i} - 1 + 2^{-W+1}$$

$$= -(1 - a_{w-1}) + \sum_{i=1}^{W-1} (1 - a_{w-1-i}) 2^{-i} + 2^{-W+1}$$



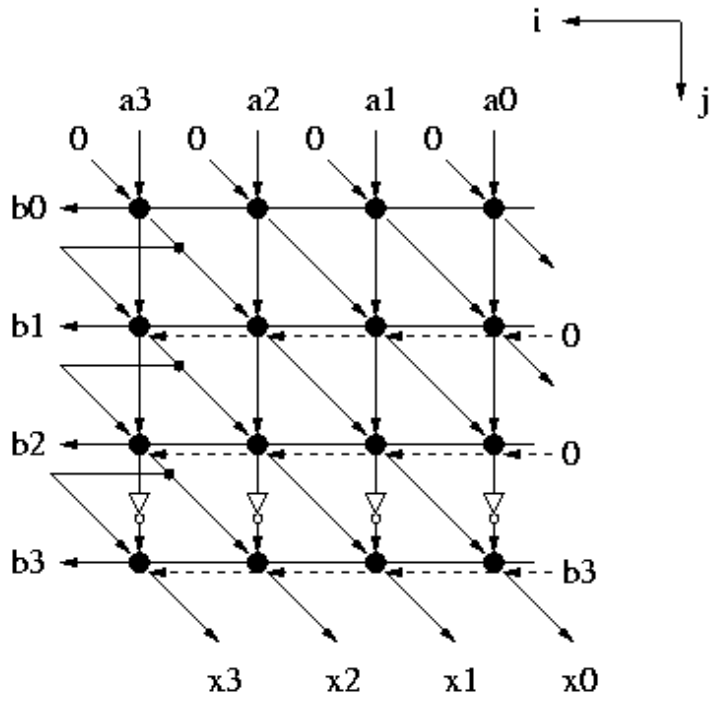
Tabular form of bit-level array multiplication

- The additions cannot be carried out directly due to terms having negative weight. Sign extension is used to solve this problem. For example,

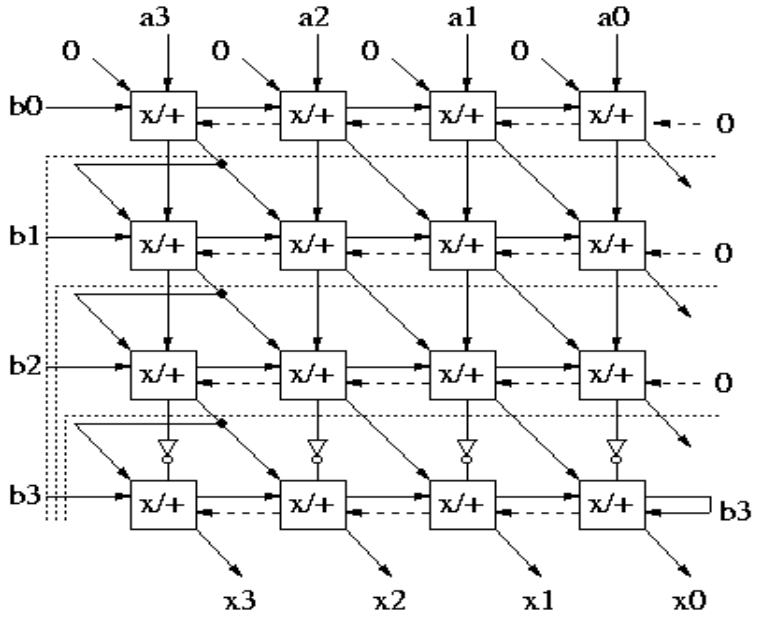
$$A = a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$
$$= -a_3 2 + a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$
$$= -a_3 2^2 + a_3 2 + a_3 + a_2 2^{-1} + a_1 2^{-2} + a_0 2^{-3}$$

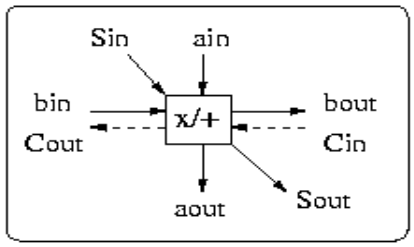
describes sign extension of A by 1 and 2 bits.

- Parallel Carry-Ripple Array Multipliers :



Bit level dependence Graph



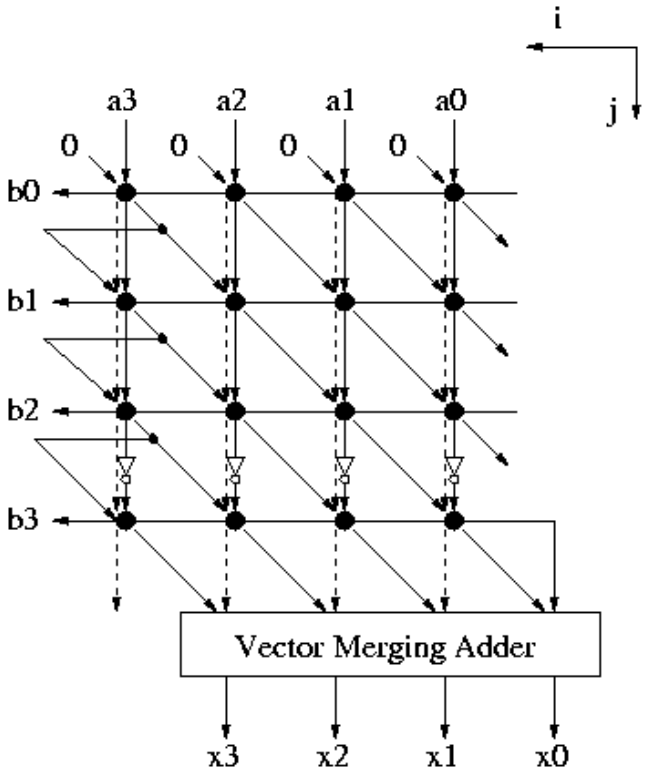


→ Broadcast signals:
bout=bin; aout=ain

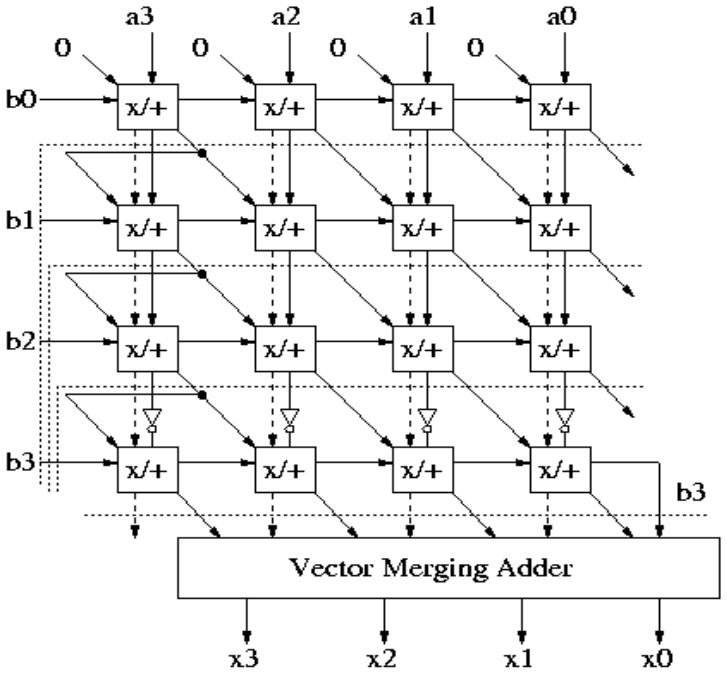
→ Single-bit Full-Adder:

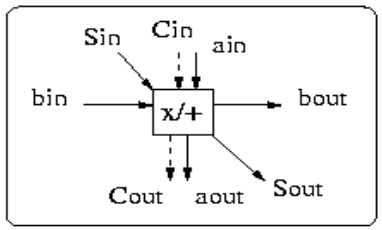
$2\,Cout + Sout = ain*bin + Sin + Cin$

Parallel Carry Ripple Multiplier



DG for 4×4-bit carry save array multiplication





→ Broadcast signals:
bout=bin; aout=ain

→ Single-bit Full-Adder:
2 Cout + Sout = ain*bin + Sin + Cin

Parallel carry-save array multiplier

- Baugh-Wooley Multipliers:
  - Handles the sign bits of the multiplicand and multiplier efficiently.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				$\overline{a_3b_0}$	a_2b_0	a_1b_0	a_0b_0
			$\overline{a_3b_1}$	a_2b_1	a_1b_1	a_0b_1	
		$\overline{a_3b_2}$	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	x_3	x_2	x_1	x_0			

Tabular form of bit-level Baugh-Wooley multiplication

- Parallel Multipliers with Modified Booth Recoding :
  - Reduces the number of partial products to accelerate the multiplication process.
  - The algorithm is based on the fact that fewer partial products need to be generated for groups of consecutive zeros and ones. For a group of "m" consecutive ones in the multiplier, i.e.,

    $\ldots 0\{11\ldots 1\}0\ldots = \ldots 1\{00\ldots \bar{0}\}0\ldots - \ldots 0\{00\ldots 1\}0\ldots$

    $= \ldots 1\{00\ldots 1\}0\ldots$

    instead of "m" partial products, only 2 partial products need to be generated is signed digit representation is used.
  - Hence, in this multiplication scheme, the multiplier bits are first recoded into signed-digit representation with fewer number of nonzero digits; the partial products are then generated using the recoded multiplier digits and accumulated.

b_{2i+1}	b_{2i}	b_{2i-1}	b'_i	Operation	Comments
0	0	0	0	+0	string of 0's
0	0	1	1	+A	end of 1's
0	1	0	1	+A	a single 1
0	1	1	2	+2A	end of 1's
1	0	0	-2	-2A	beginning of 1's
1	0	1	-1	-A	A single 0
1	1	0	-1	-A	beginning of 1's
1	1	1	0	-0	string of 1's

Radix-4 Modified Booth Recoding Algorithm

Recoding operation can be described as:

$$b'_i = -2b_{2i+1} + b_{2i} + b_{2i-1}$$

Interleaved Floor-Plan and Bit-Plane-Based Digital Filters

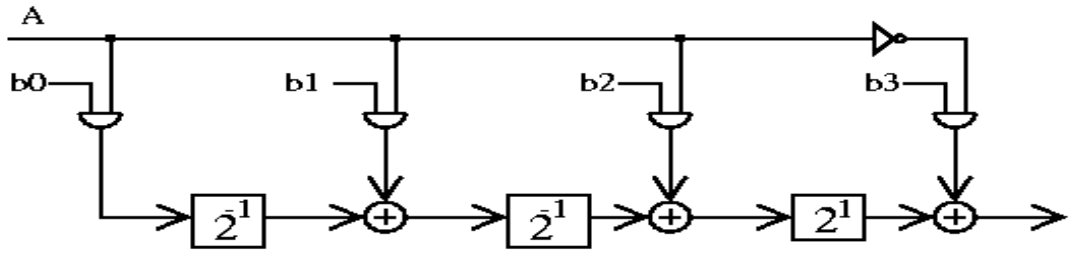
- A constant coefficient FIR filter is given by:

$$y(n) = x(n) + f \bullet x(n-1) + g \bullet x(n-2)$$

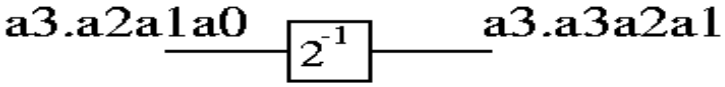
  where, x(n) is the input signal, and f and g are filter coefficients.
- The main idea behind the interleaved approach is to perform the computation and accumulation of partial products associated with f and g simultaneously thus increasing the speed.
- This increases the accuracy as truncation is done at the final step.
- If the coefficients are interleaved in such a way that their partial products are computed in different rows, the resulting architecture is called bit-plane architecture.

Bit-Serial Multipliers

- Lyon's Bit-Serial Multiplier using Horner's Rule :

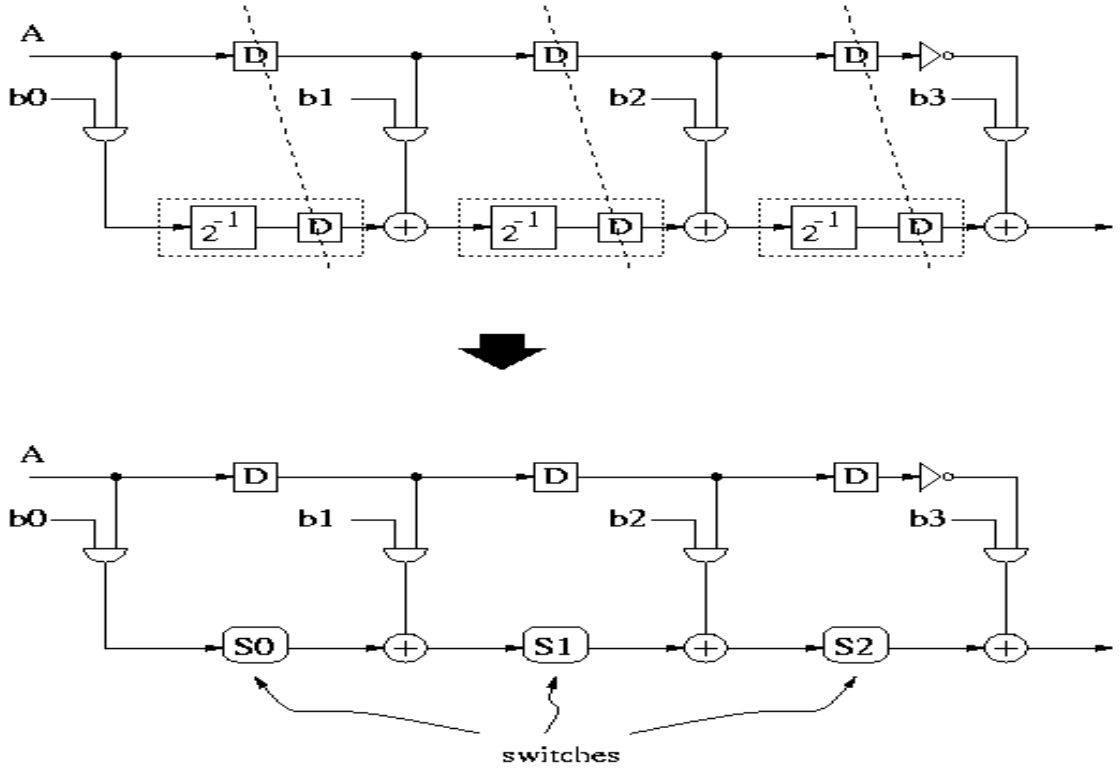


(a)

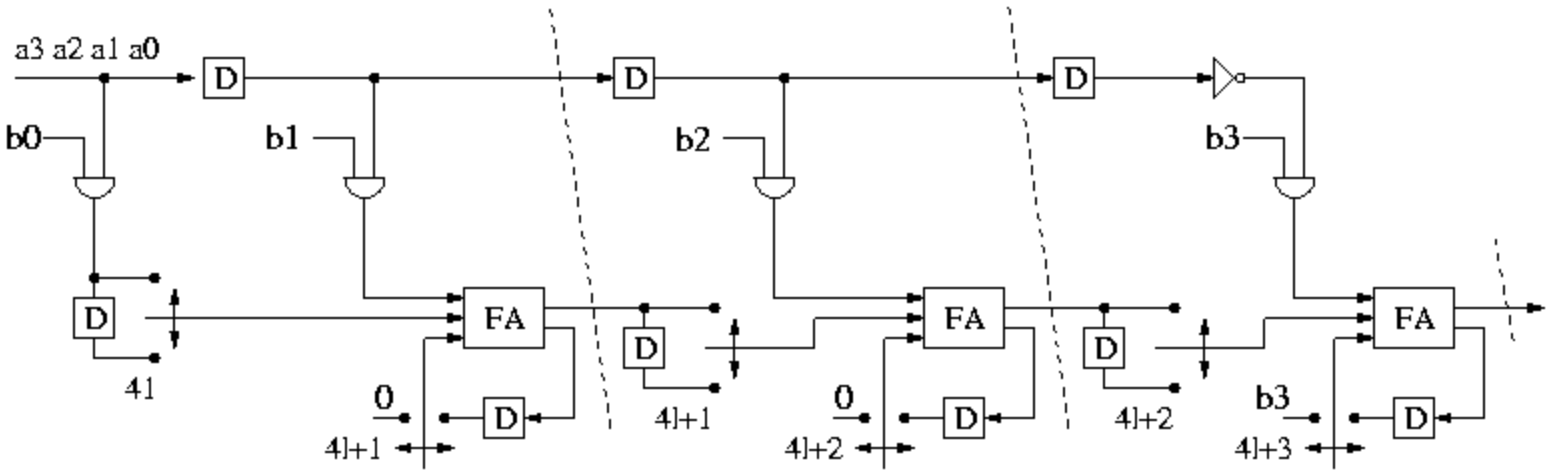


(b)

- For the scaling operator, the first output bit a_1 should be generated at the same time instance when the first input a_1 enters the operator. Since input a_1 has not entered the system yet, the scaling operator is non-causal and cannot be implemented in hardware.



Derivation of implementable bit-serial 2's complement multiplier



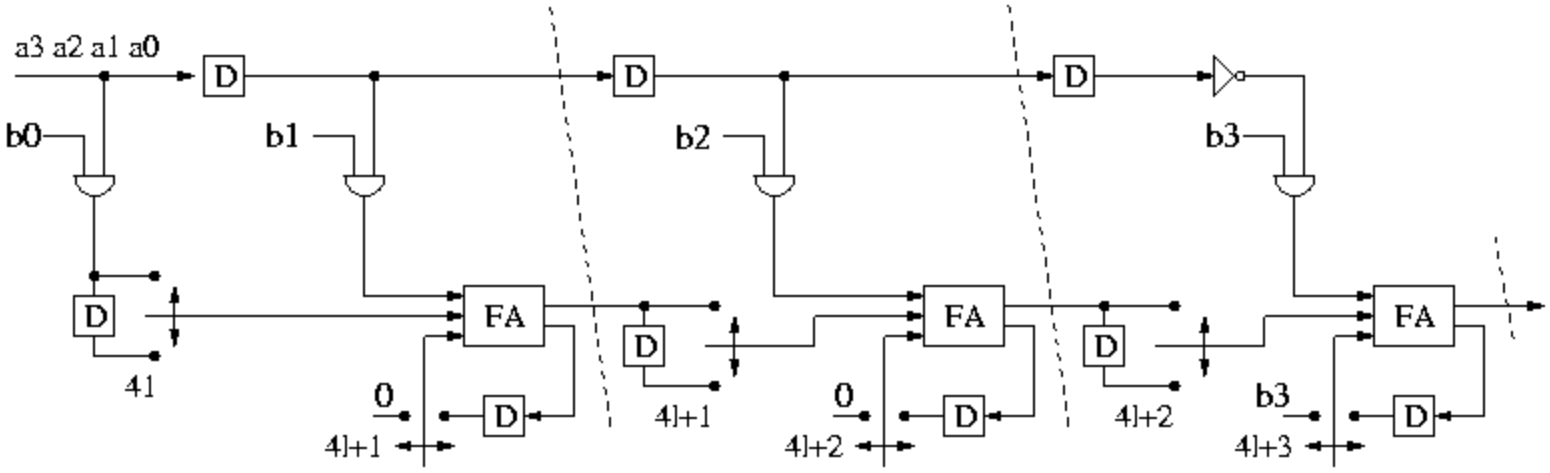
Lyon's bit-serial 2's complement multiplier

Design of Bit-Serial Multipliers Using Systolic Mappings

•Design of Lyon's bit-serial multiplier by systolic mapping Using DG of ripple carry multiplication.

e	$p^T e$	$s^T e$
a(0,1)	1	1
b(1,0)	0	1
carry(1,0)	0	1

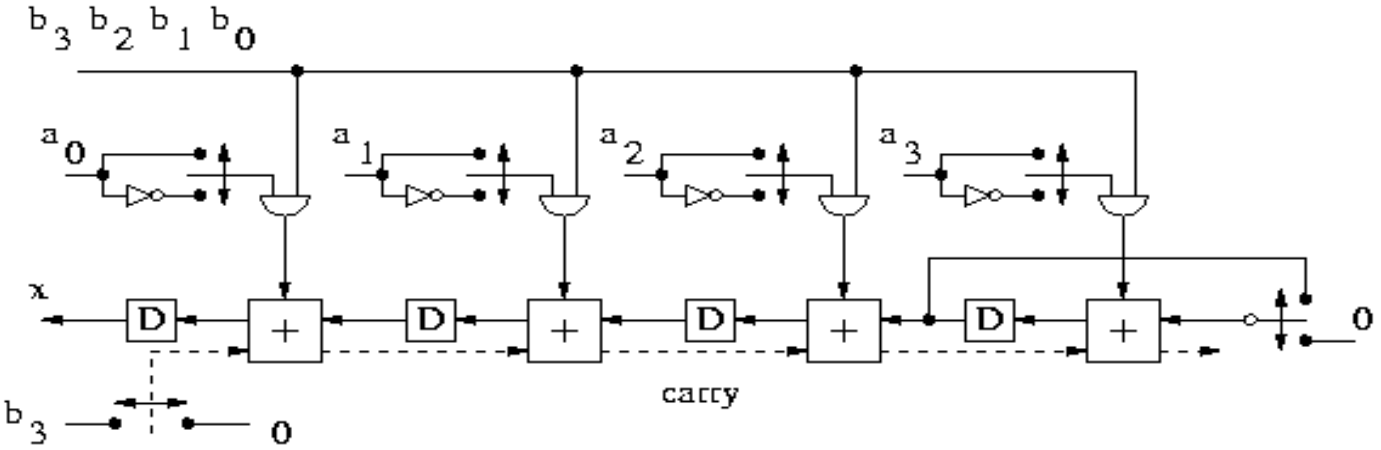
Here, $d^T = [1\ 0]$, $s^T = [1\ 1]$ and $p^T = [0\ 1]$



•Design of bit-serial multiplier by systolic mapping using DG of ripple carry multiplication and the following :

$d^T = [0\ 1]$, $s^T = [0\ 1]$ and $p^T = [1\ 0]$

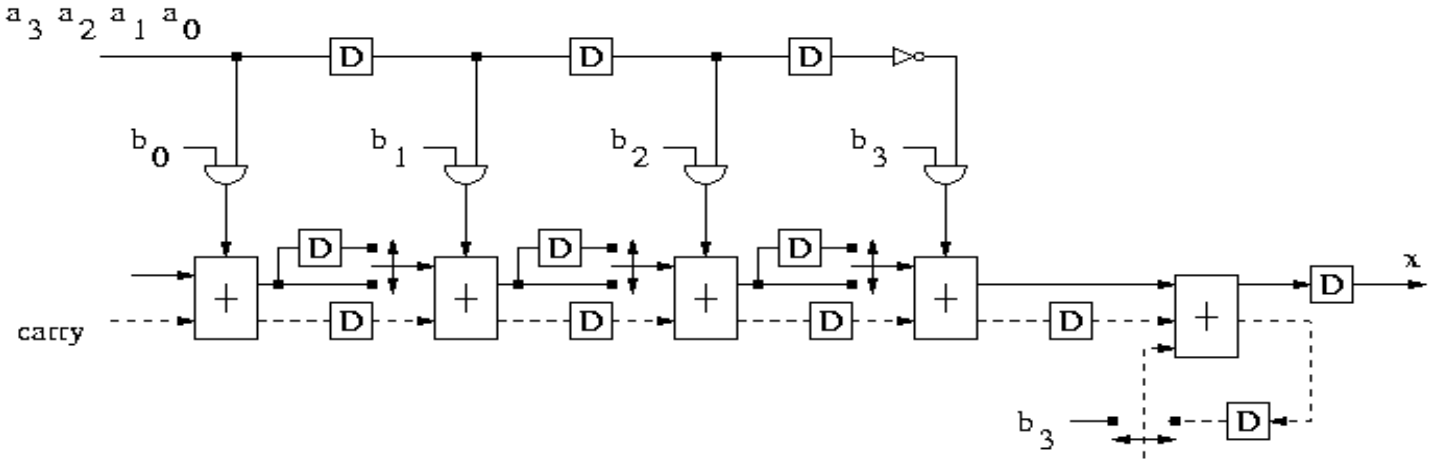
e	$p^T e$	$s^T e$
a(0,1)	0	1
b(1,0)	1	0
carry(1,0)	1	0
x(-1,1)	-1	1

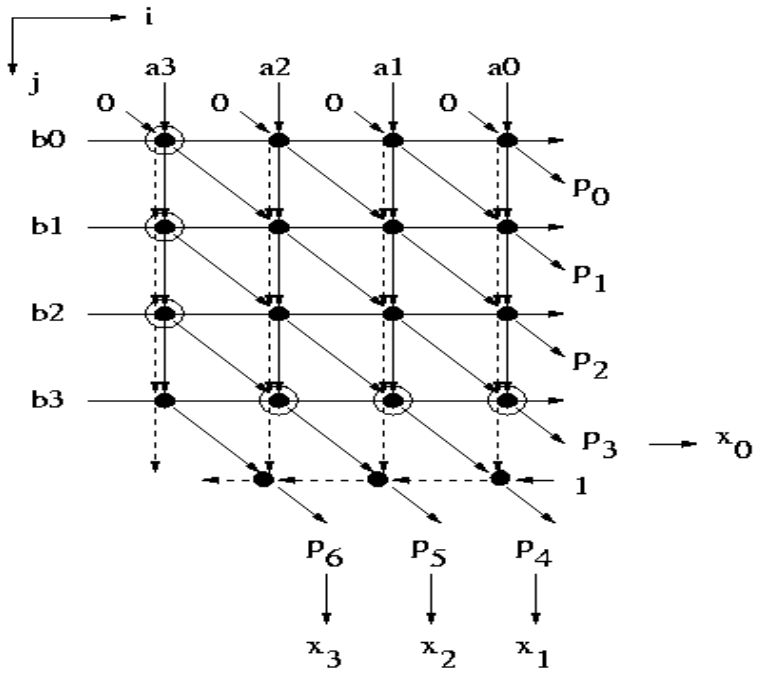


•Design of bit-serial multiplier by systolic mapping
using DG for carry-save array multiplication and the following :

$d^T = [1\ 0]$, $s^T = [1\ 1]$ and $p^T = [0\ 1]$

e	p^Te	s^Te
a(0,1)	1	1
b(1,0)	1	1
carry(1,0)	0	1
x(-1,1)	1	0





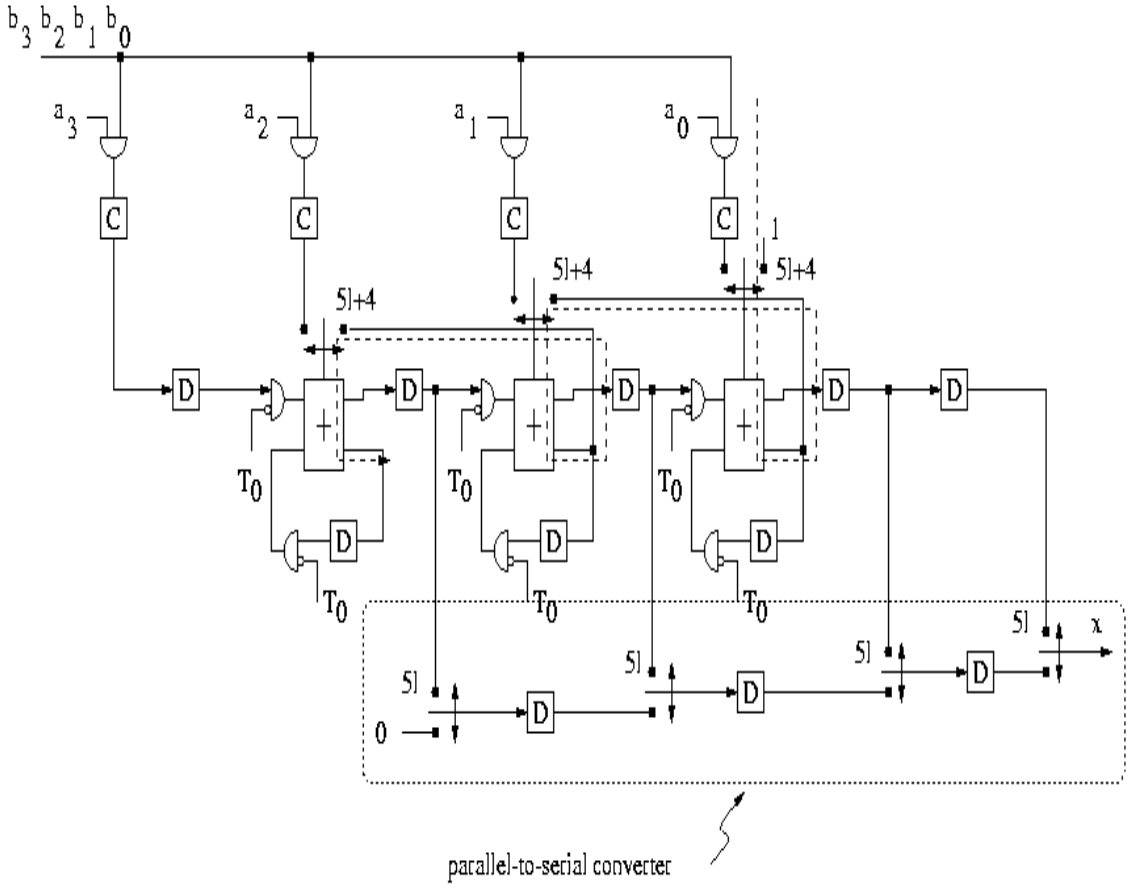
Dependence graph for carry save Baugh-Wooley multiplication with carry ripple vector merging

•Design of bit-serial Baugh-Wooley multiplier by systolic mapping using DG for Baugh-Wooley multiplication and the following :

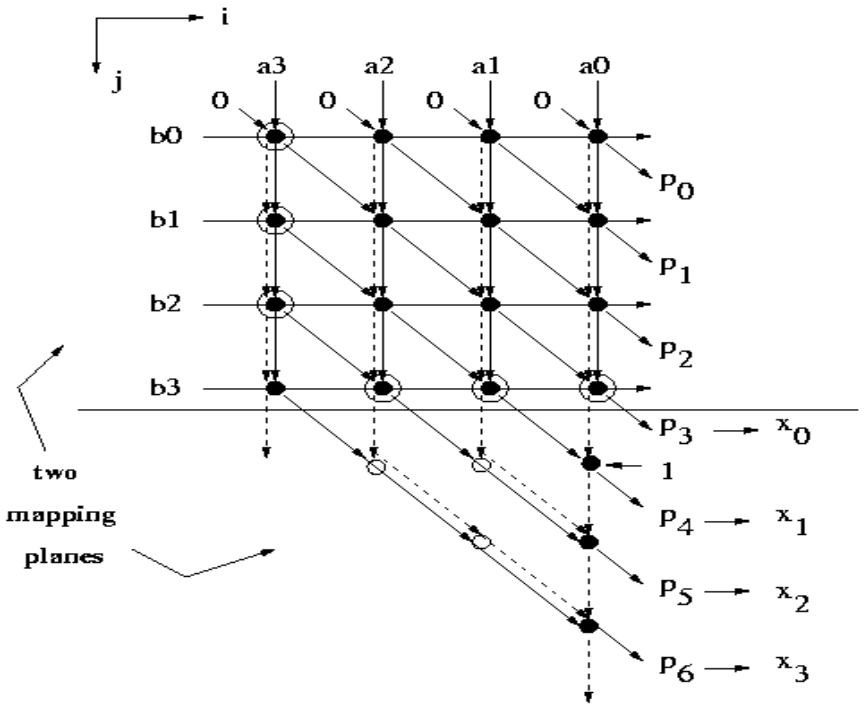
$d^T = [0\ 1]$, $s^T = [0\ 1]$ and $p^T = [1\ 0]$

e	$p^T e$	$s^T e$
a(0,1)	0	1
carry(0,1)	0	1
b(1,0)	1	0
x(1,1)	1	1
carry-vm(-1,0)	-1	0

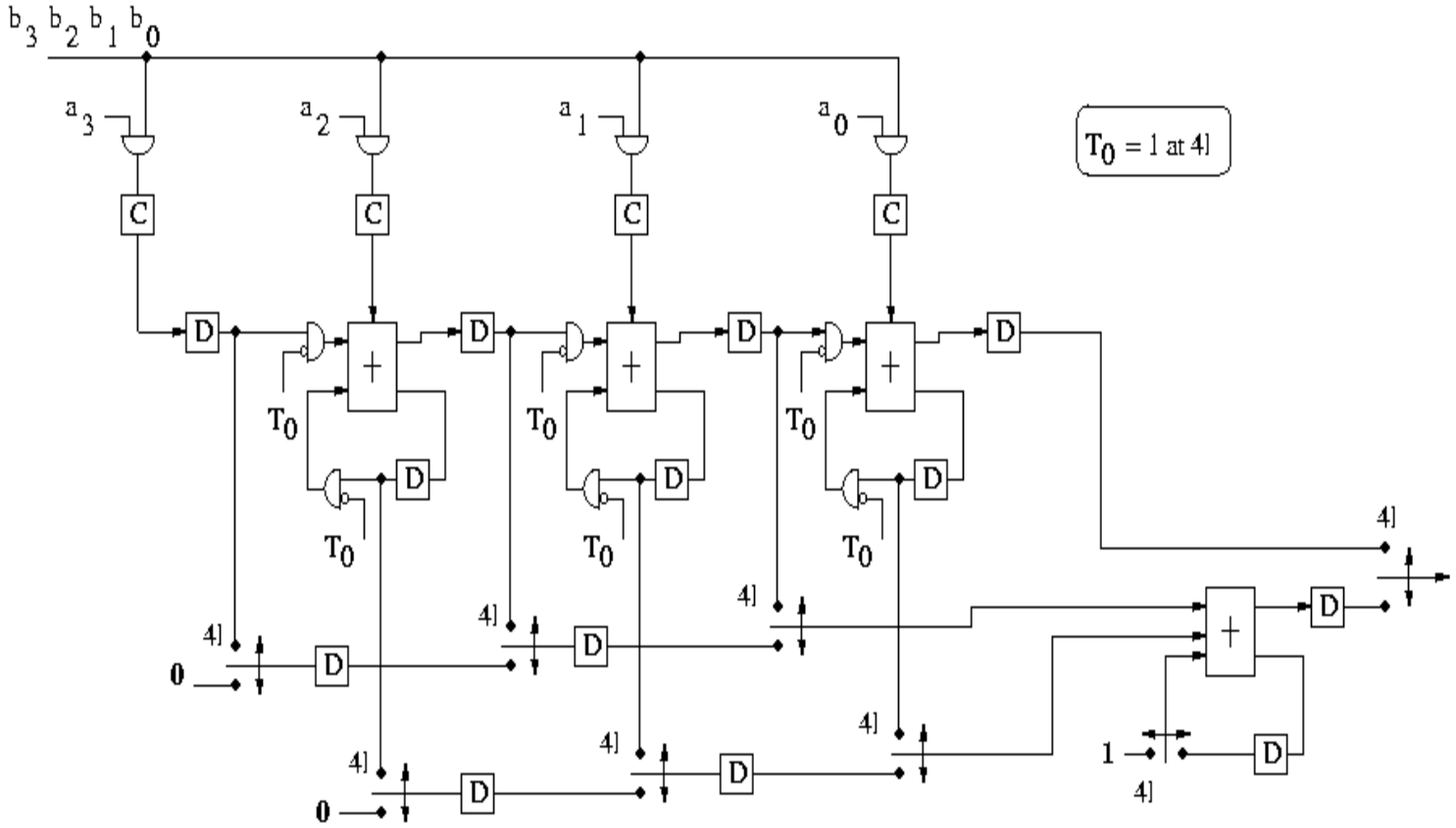
Here, carry-vm denotes the carry outputs in the vector merging portion.



Bit-Serial Baugh-Wooley Multiplier

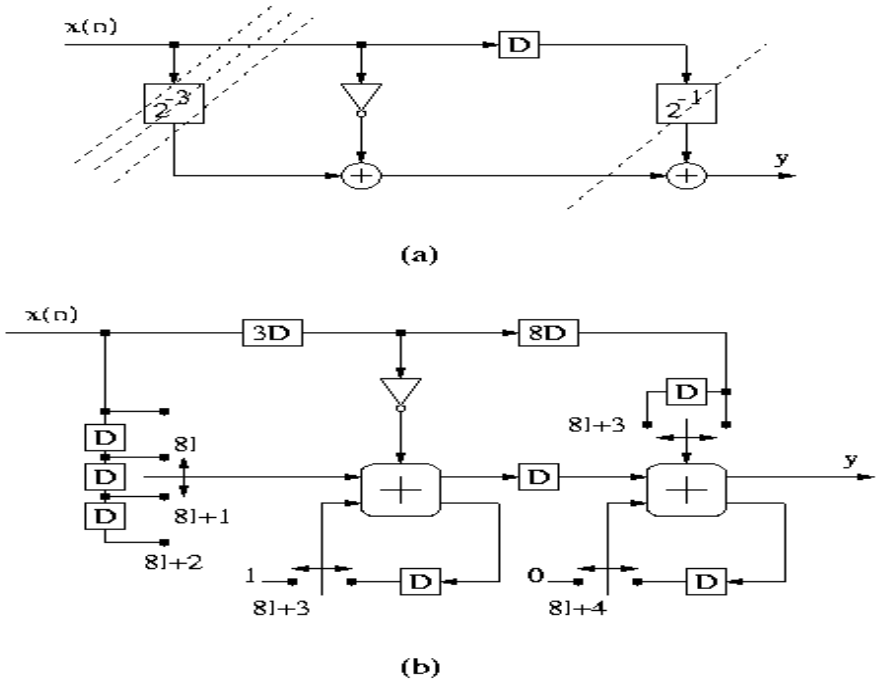


DG bit-serial Baugh-Wooley multiplier with carry-save array and vector merging portion treated as two separate planes



Bit-serial Baugh-Wooley multiplier
using the DG having two separate planes
for carry-save array and the vector merging portion

Bit-Serial FIR Filter



Bit-level pipelined bit-serial FIR filter, $y(n) = (-7/8)x(n) + (1/2)x(n-1)$, where constant coefficient multiplications are implemented as shifts and adds as $y(n) = -x(n) + x(n)2^{-3} + x(n-1)2^{-1}$.
(a)Filter architecture with scaling operators;
(b) feasible bit-level pipelined architecture

Bit-Serial IIR Filter

- Consider implementation of the IIR filter

$$Y(n) = (-7/8)y(n-1) + (1/2)y(n-2) + x(n)$$

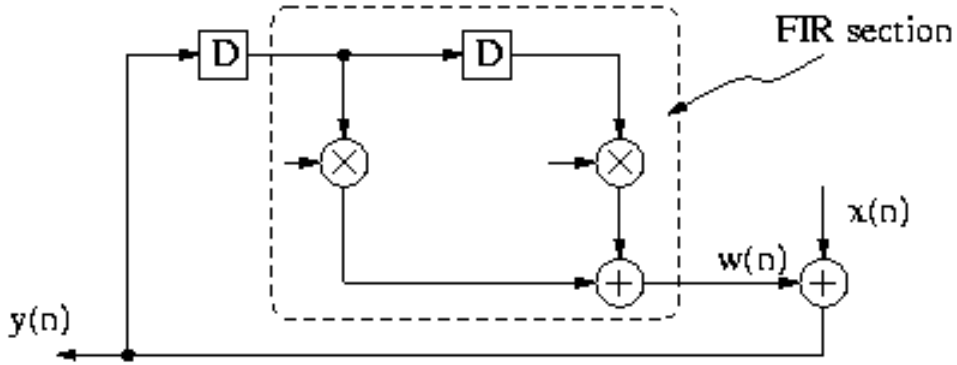
where, signal word-length is assumed to be 8.

- The filter equation can be re-written as follows:

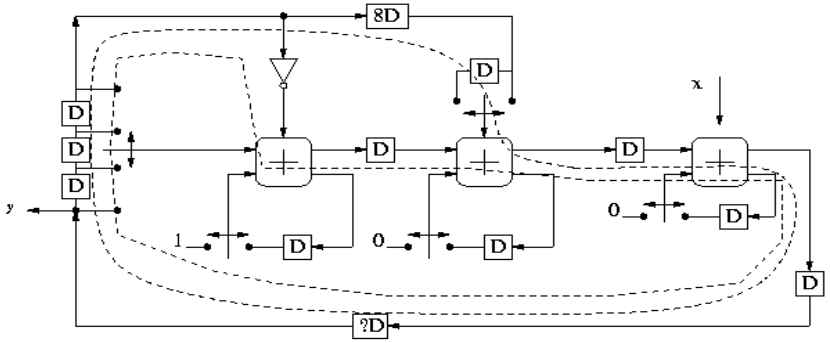
$$w(n) = (-7/8)y(n-1) + (1/2)y(n-2)$$

$$Y(n) = w(n) + x(n)$$

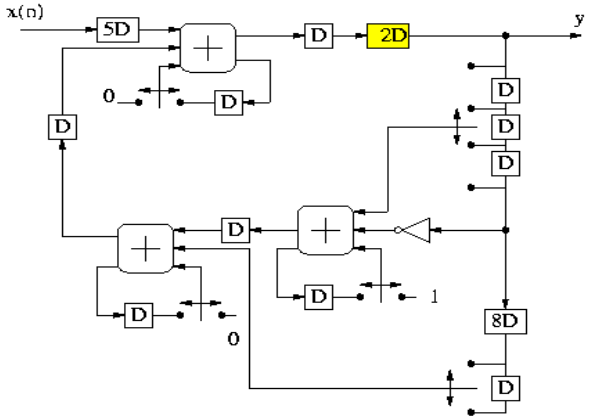
which can be implemented as an FIR section from y(n-1) with an addition and a feedback loop as shown below:



- Steps for deriving a bit-serial IIR filter architecture:
  - A bit-level pipelined bit-serial implementation of the FIR section needs to be derived.
  - The input signal x(n) is added to the output of the bit-serial FIR section w(n).
  - The resulting signal y(n) is connected to the signal y(n-1).
  - The number of delay elements in the edge marked ?D needs to be determined.(see figure in next page)
- For, systems containing loop, the total number of delay elements in the loops should be consistent with the original SFG, in order to maintain synchronization and correct functionality.
- **Loop delay synchronization** involves matching the number of word-level loop delay elements and that in the bit-serial architecture. The number of bit-level delay elements in the bit-serial loops should be $W \times N_D$, where W is signal word-length and N_D denotes the number of delay elements in the word-level SFG.



(a)



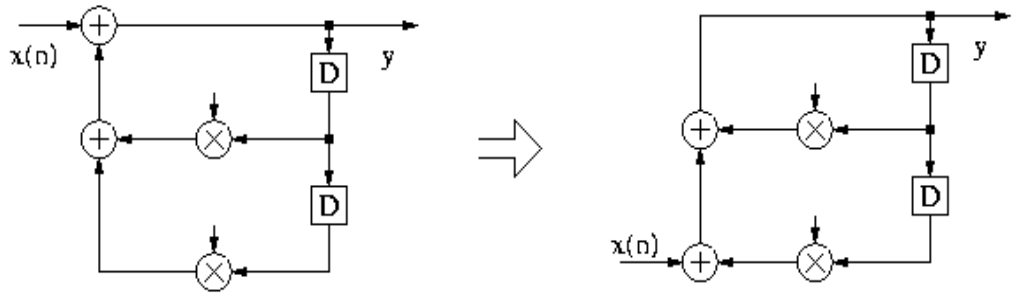
(b)

- Bit-level pipelined bit-serial architecture, without synchronization delay elements. (b) Bit-serial IIR filter. Note that this implementation requires a minimum feasible word-length of 6.

Note:

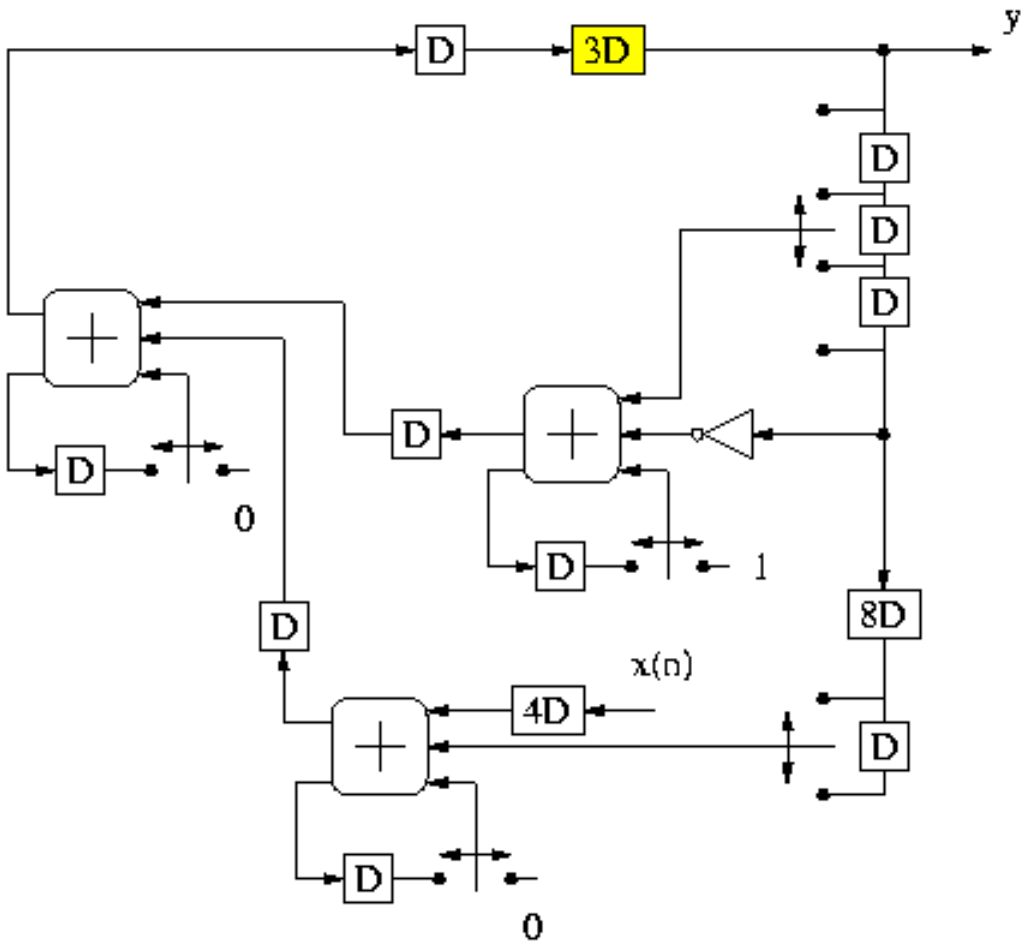
- To compute the total number of delays in the bit-level architecture, the paths with the **largest number of delay elements** in the switching elements should be counted.
- Input synchronizing delays (also referred as **shimming** delays or **skewing** delays).
- It is also possible that the loops in the intermediate bit-level pipelined architecture may contain more than $W \times N_D$ number of bit-level delay elements, in which case the word-length needs to be increased.
- The architecture without the two loop synchronizing delays can function correctly with a signal word-length of 6, which is the minimum word-length for the bit-level pipelined bit-serial architecture.

- **Associativity transformation** :



(a) (b)

Loop iteration bound of IIR filter can be reduced from one-multiply-two-add to one-multiply-add by associative transformation



Bit-serial IIR filter after associative transformation. This implementation requires a minimum feasible word-length of 5.

Canonic Signed Digit Arithmetic

- Encoding a binary number such that it contains the fewest number of non-zero bits is called **canonic signed digit**(CSD).
- The following are the properties of CSD numbers:
  - No 2 consecutive bits in a CSD number are non-zero.
  - The CSD representation of a number contains the minimum possible number of non-zero bits, thus the name canonic.
  - The CSD representation of a number is unique.
  - CSD numbers cover the range (-4/3,4/3), out of which the values in the range [-1,1) are of greatest interest.
  - Among the W-bit CSD numbers in the range [-1,1), the average number of non-zero bits is $W/3 + 1/9 + O(2^{-W})$. Hence, on average, CSD numbers contains about 33% fewer non-zero bits than two's complement numbers.

- Conversion of W-bit number to CSD format:
  - $A = a'_{W-1}.\ a'_{W-2} \ldots a'_1.\ a'_0$ = 2's complement number
  - Its CSD representation is $a_{W-1}.\ a_{W-2} \ldots a_1.\ a_0$
- Algorithm to obtain CSD representation:
  - $a'_{-1} = 0;$
  - $\gamma_{-1} = 0;$
  - $a'_W = a'_{W-1};$
  - for (i = 0 to W-1)

    {

    $\theta_i = a'_i \oplus a'_{i-1};$

    $\gamma_i = \overline{\gamma}_{i-1}\theta_i;$

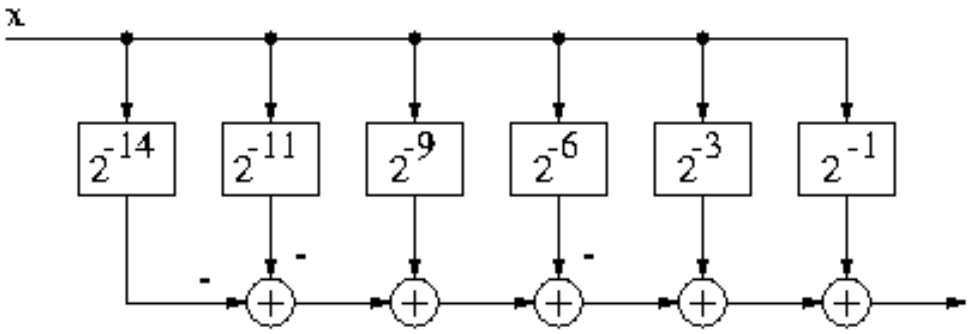
    $a_i = (1 - 2a'_{i+1})\gamma_i;$

    }

i	W	W-1								0	-1
a'_i	1	1	0	1	1	1	0	0	1	1	
θ_i		1	1	0	0	1	0	1	0	1	
γ_i		0	1	0	0	1	0	1	0	1	
$1 - 2a'_{i+1}$		-1	-1	1	-1	-1	-1	1	1	-1	
a_i		0	-1	0	0	-1	0	1	0	-1	

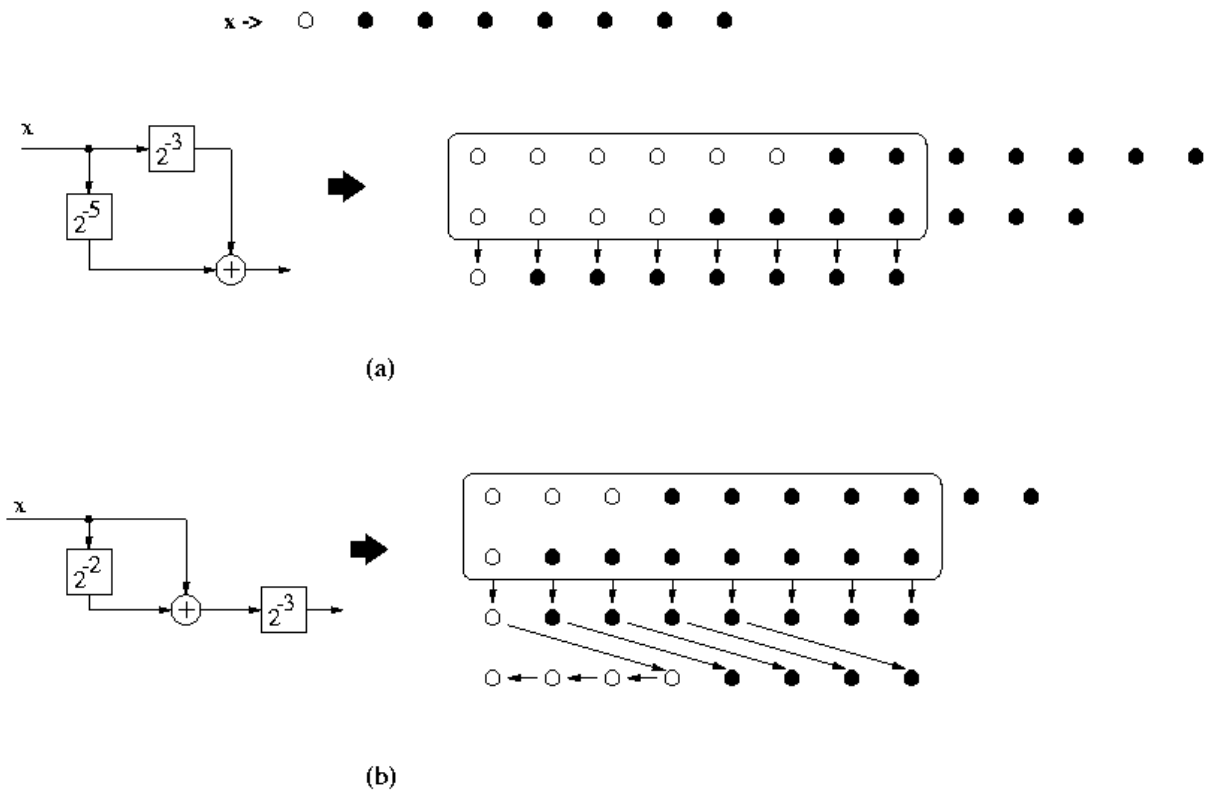
Table showing the computation of the CSD representation for the number 1.01110011.

CSD Multiplication

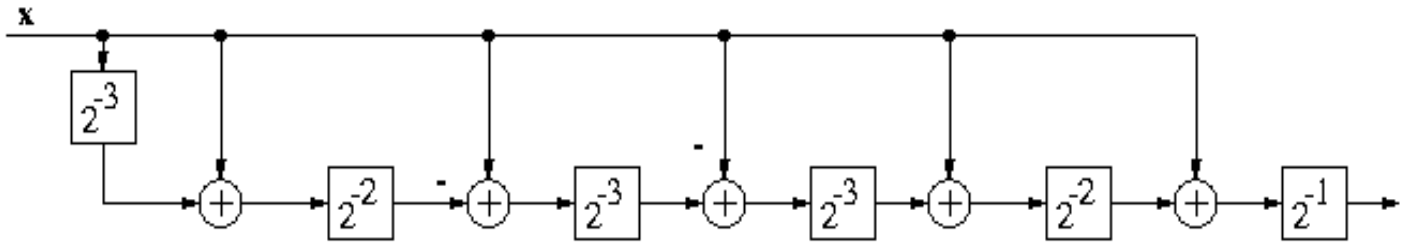


A CSD multiplier using linear arrangement of adders to compute $x \times 0.10100\bar{1}00\bar{1}0\bar{1}00\bar{1}$

- Horner's rule for precision improvement : This involves delaying the scaling operations common to the 2 partial products thus increasing accuracy.
- For example, $x \bullet 2^{-5} + x \bullet 2^{-3}$ can be implemented as $(x \bullet 2^{-2} + x)2^{-3}$ to increase the accuracy.

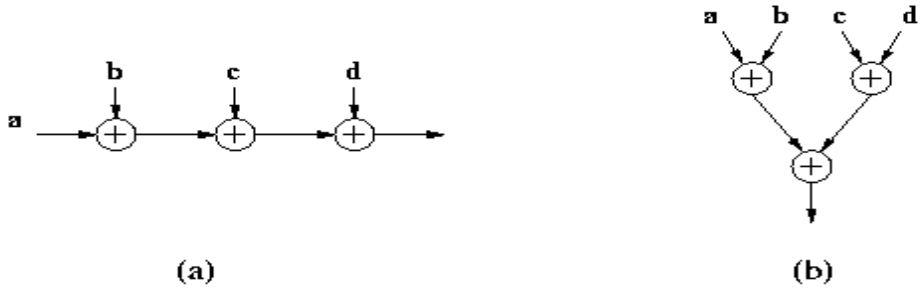


Using Horner's rule for partial product accumulation to reduce the truncation error.

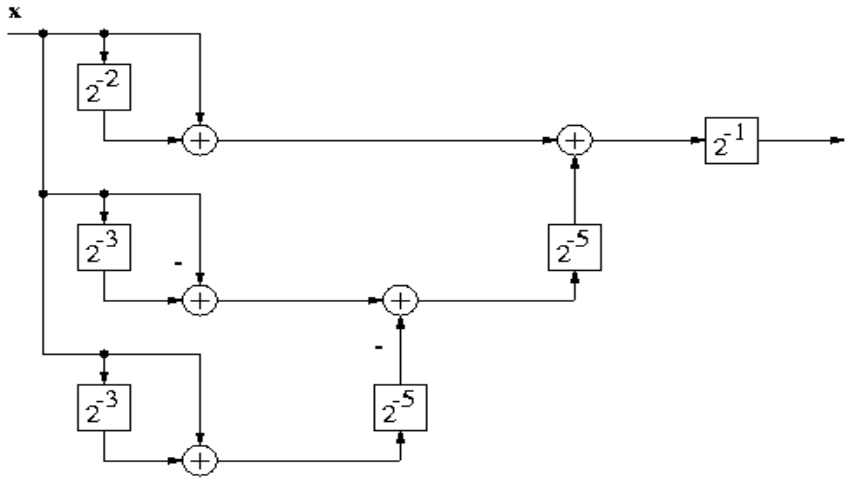


Rearrangement of the CSD multiplication of $x \times 0.1010010010101001$ using Horner's rule for partial product accumulation to reduce the truncation error.

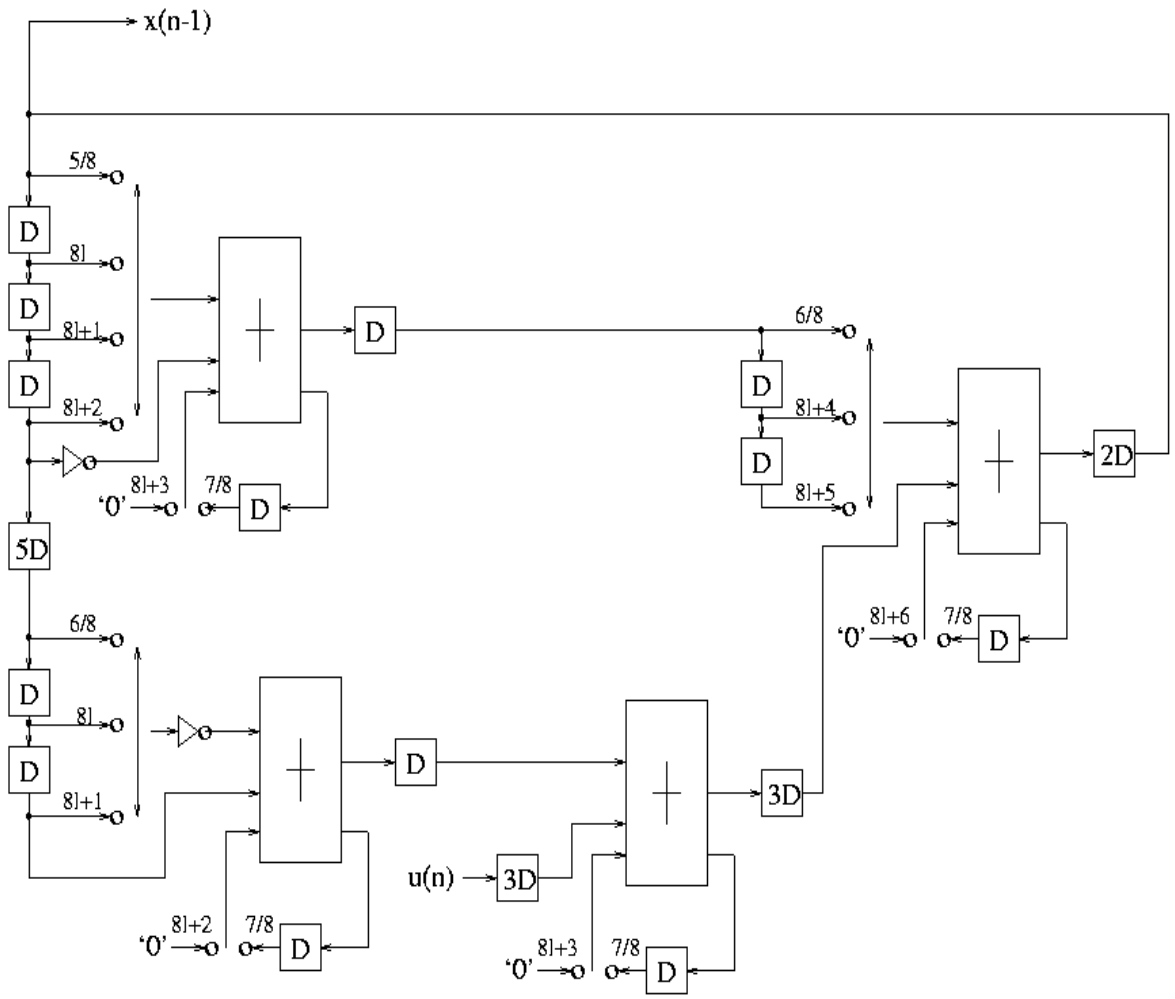
Use of Tree-Height Reduction for Latency Reduction



(a) linear arrangement (b) tree arrangement



Combination of tree-type arrangement and Horner's rule for the accumulation of partial products in CSD multiplication



Bit serial architecture using CSD. In this case the coefficients -7/32 = -1/4 + 1/32 is encoded as $0.0\bar{1}001$ and ¾ = 1 – ¼ is encoded as $1.0\bar{1}$.

Chapter 14: Redundant Arithmetic

Keshab K. Parhi

- A non-redundant radix-r number has digits from the set{0, 1, … , r - 1} and all numbers can be represented in a unique way.
- A radix-r redundant signed-digit number system is based on digit set $S \equiv \{-\beta, -(\beta - 1), \ldots, -1, 0, 1, \ldots, \alpha\}$, where, $1 \leq \beta, \alpha \leq r - 1$.
- The digit set S contains more than r values $\Rightarrow$ multiple representations for any number in signed digit format. Hence, the name redundant.
- A symmetric signed digit has $\alpha = \beta$.
- Carry-free addition is an attractive property of redundant signed-digit numbers. This allows most significant digit (msd) first redundant arithmetic, also called on-line arithmetic.

Redundant Number Representations

- A symmetric signed-digit representation uses the digit set $D_{<r.\alpha>} = \{-\alpha, \ldots, -1, 0, 1, \ldots, \alpha\}$, where r is the radix and α the largest digit in the set. A number in this representation is written as :

$$X_{<r.\,\alpha>} = x_{W-1}.x_{W-2}.x_{W-3}\ldots x_0 = \sum x_{W-1-i} r^i$$

The sign of the number is given by the sign of the most significant non-zero digit.

Digit Set $D_{<r.\alpha>}$	α	Redundancy Factor ρ
Incomplete	$< (r-1)/2$	$< \frac{1}{2}$
Complete but non-redundant	$= (r-1)/2$	$= \frac{1}{2}$
Redundant	$\geq \lceil r/2 \rceil$	$> \frac{1}{2}$
Minimally redundant	$= \lceil r/2 \rceil$	$> \frac{1}{2}$ and < 1
Maximally redundant	$= r - 1$	$= 1$
Over-redundant	$> r - 1$	> 1

Hybrid Radix-2 Addition

$$S_{<2.1>} = X_{<2.1>} + Y$$

where, $X_{<r.\alpha>} = x_{W-1}.x_{W-2}x_{W-3}...x_0$, $Y = y_{W-1}.y_{W-2}y_{W-3}...y_0$. The addition is carried out in two steps :

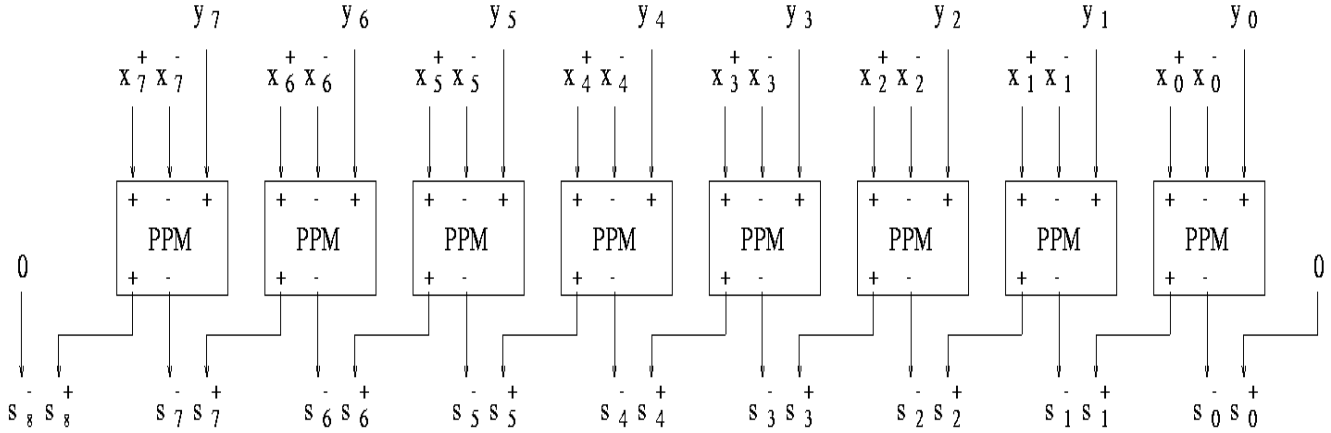
1. The 1st step is carried out in parallel for all the bit positions. An intermediate sum $p_i = x_i + y_i$ is computed, which lies in the range $\{\bar{1}, 0, 1, 2\}$. The addition is expressed as:

   $$x_i + y_i = 2t_i + u_i,$$

   where t_i is the transfer digit and has value 0 or 1, and is denoted as t_i^+; u_i is the interim sum and has value either 1 or 0 and is denoted as $-u_i^-$. t_{-1} is assigned the value of 0.
2. The sum digits s_i are formed as follows:

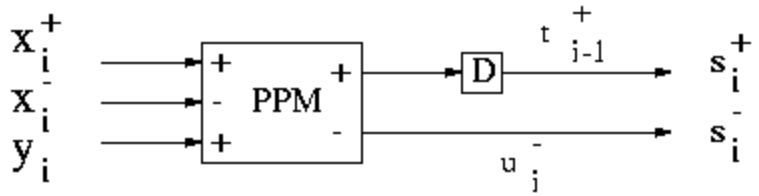
   $$s_i = t_{i-1}^+ - u_i^-$$

Digit	Radix 2 Digit Set	Binary Code
x_i	$\{\bar{1}, 0, 1\}$	$x_i^+ - x_i^-$
y_i	$\{0, 1\}$	y_i^+
$p_i = x_i + y_i$	$\{\bar{1}, 0, 1, 2\}$	$2t_i + u_i$
u_i	$\{\bar{1}, 0\}$	$-u_i^-$
t_i	$\{0, 1\}$	t_i^+
$s_i = u_i + t_{i-1}$	$\{\bar{1}, 0, 1\}$	$s_i^+ - s_i^-$

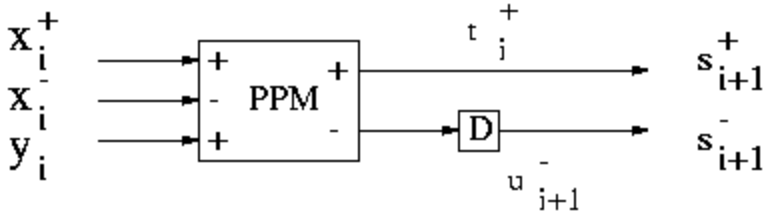


Eight-digit hybrid radix-2 adder

Digit-serial adder formed by folding



LSD-first adder



MSD-first adder

Hybrid Radix-2 Subtraction

$$S_{<2.1>} = X_{<2.1>} - Y$$

where, $X_{<r.\alpha>} = x_{W-1}.x_{W-2}x_{W-3}...x_0$, $Y = y_{W-1}.y_{W-2}y_{W-3}...y_0$. The addition is carried out in two steps :

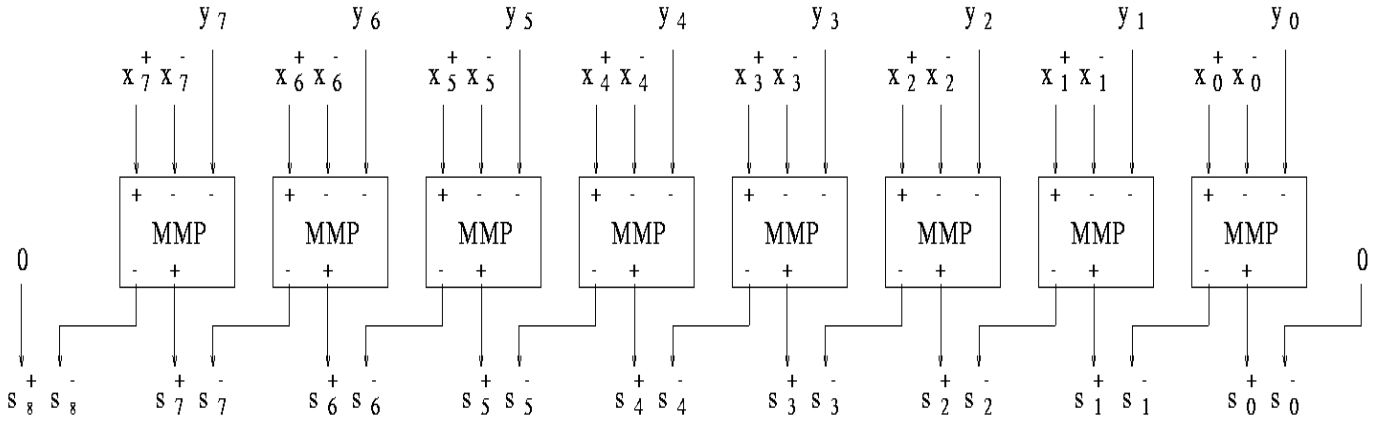
1. The 1st step is carried out in parallel for all the bit positions. An intermediate difference $p_i = x_i - y_i$ is computed, which lies in the range $\{\bar{2}, \bar{1}, 0, 1\}$. The addition is expressed as:

   $$x_i - y_i = 2t_i + u_i,$$

   where t_i is the transfer digit and has value 1 or 0, and is denoted as $-t_i^-$; u_i is the interim sum and has value either 0 or 1 and is denoted as u_i^+. t_{-1} is assigned the value of 0.
2. The sum digits s_i are formed as follows:

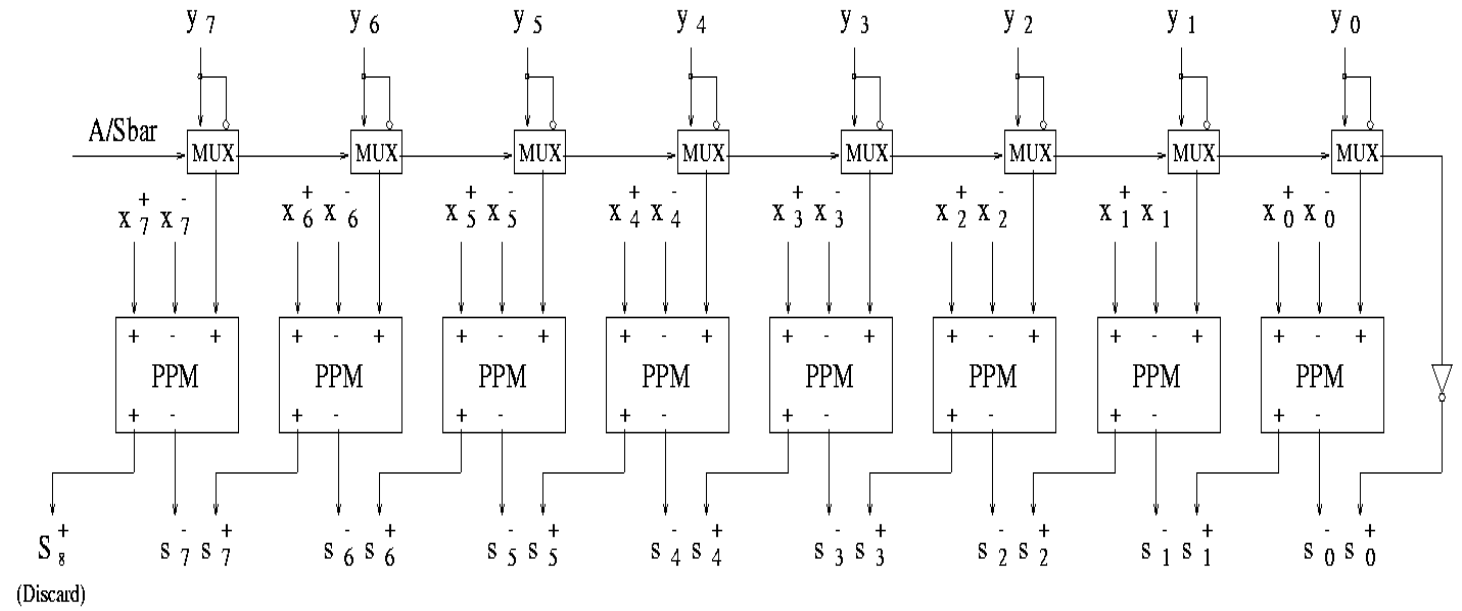
   $$s_i = -t_{i-1}^- + u_i^+$$

Digit	Radix 2 Digit Set	Binary Code
x_i	$\{\overline{1}, 0, 1\}$	$x_i^+ - x_i^-$
y_i	$\{0, 1\}$	y_i^-
$p_i = x_i - y_i$	$\{\overline{2}, \overline{1}, 0, 1\}$	$2t_i + u_i$
u_i	$\{0, 1\}$	u_i^+
t_i	$\{\overline{1}, 0\}$	$-t_i^-$
$s_i = u_i + t_{i-1}$	$\{\overline{1}, 0, 1\}$	$s_i^+ - s_i^-$



Eight-digit hybrid radix-2 subtractor

Hybrid Radix-2 Addition/Subtraction



Hybrid radix-2 adder/subtractor ($A/\overline{S} = 1$ for addition and $A/\overline{S} = 0$ for subtraction)

- This is possible if one of the operands is in radix-r complement representation. Hybrid subtraction is carried out by hybrid addition where the 2's complement of the subtrahend is added to the minuend and the carry-out from the most significant position is discarded.

Signed Binary Digit (SBD) Addition/Subtraction

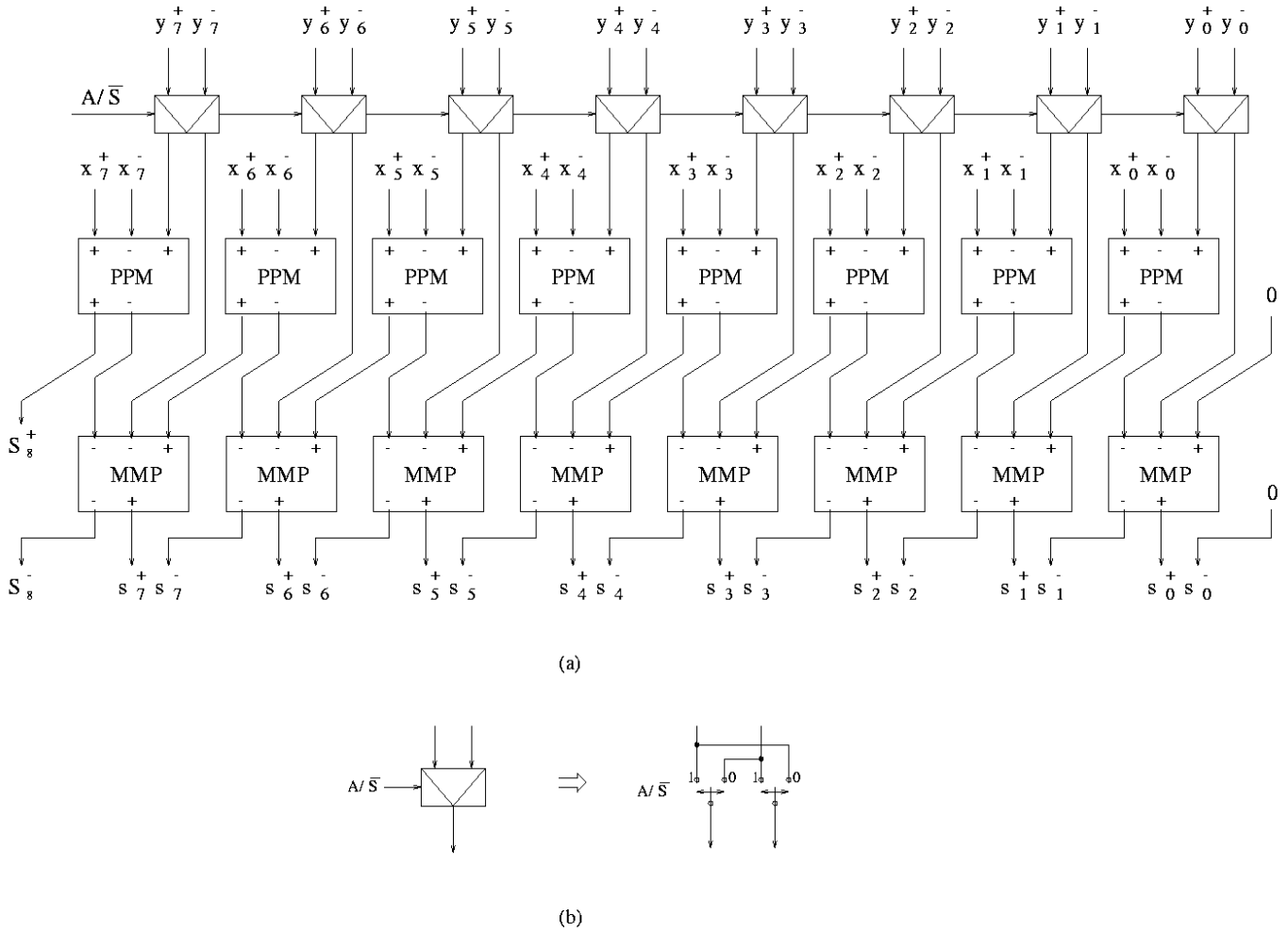
- $Y_{<r.\alpha>} = Y^+ - Y^-$, is a signed digit number, where Y^+ and Y^- are from the digit set $\{0, 1, \ldots, \alpha\}$.
- A signed digit number is thus subtraction of 2 unsigned conventional numbers.
- Signed addition is given by:

$$S_{<r.\alpha>} = X_{<r.\alpha>} + Y_{<r.\alpha>} = X_{<r.\alpha>} + Y^+ - Y^-,$$

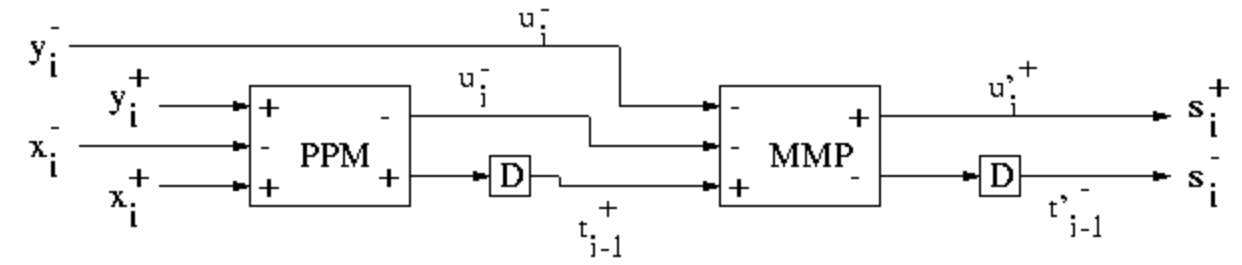
$$\Rightarrow S1_{<r.\alpha>} = X_{<r.\alpha>} + Y^+,$$

$$S_{<r.\alpha>} = S1_{<r.\alpha>} - Y^-$$

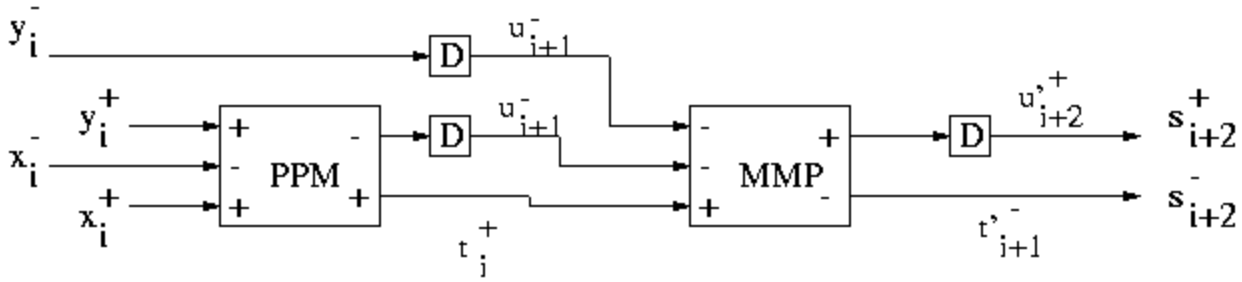
- Digit serial SBD adders can be derived by folding the digit parallel adders in both lsd-first and msd-first modes.
- LSD-first adders have zero latency and msd-first adders have latency of 2 clock cycles.



(a) Signed binary digit adder/subtractor
(b) Definition of the switching box



(a)



(b)

Digit serial SBD redundant adders. (a) LSD-first adder (b) msd-first adder

Maximally Redundant Hybrid Radix-4 Addition (MRHY4A)

- Maximally redundant numbers are based on digit set $D_{<4.3>}$.

$$S_{<4.3>} = X_{<4.3>} - Y_4$$

- The first step computes:

$$x_i + y_i = 4t_i + u_i$$

Replacing the respective binary codes from the table the following is obtained :

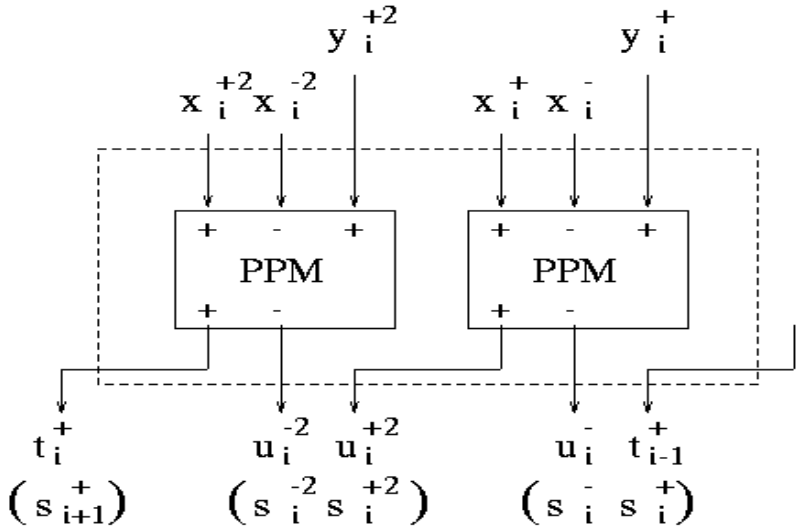
$$(2x_i^{+2} - 2x_i^{-2} + 2y_i^{+2}) + x_i^{+} - x_i^{-} + y_i^{+} = 4t_i^{+} + 2u_i^{+2} - 2u_i^{-2} - u_i^{-}$$

A MRHY4A cell consisting of two PPM adders is used to compute the above.

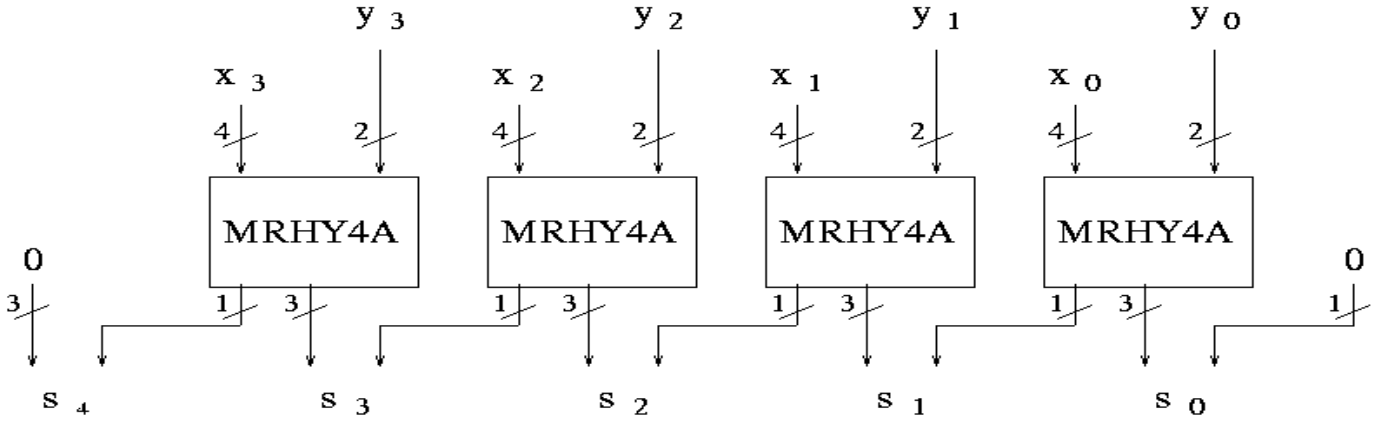
- Step 2 computes computes $s_i = t_{i-1} + u_i$. Replacing s_i, u_i, and t_{i-1} by corresponding binary codes leads to $s_i^{+2} = u_i^{+2}$, $s_i^{-2} = u_i^{-2}$, $s_i^{+}=t_{i-1}^{+}$ and $s_i^{-} = u_i^{-}$.

Digit	Radix 4 Digit Set	Binary Code
x_i	$\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$	$2x_i^{+2} - 2x_i^{-2} + x_i^{+} - x_i^{-}$
y_i	$\{0, 1, 2, 3\}$	$2y_i^{+2} + y_i^{+}$
$p_i = x_i + y_i$	$\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5, 6\}$	$4t_i + u_i$
u_i	$\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2\}$	$2u_i^{+2} - 2u_i^{-2} - u_i^{-}$
t_i	$\{0, 1\}$	t_i^{+}
$s_i = u_i + t_{i-1}$	$\{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$	$2s_i^{+2} - 2s_i^{-2} + s_i^{+} - s_i^{-}$

Digit sets involved in Maximally Redundant Hybrid Radix-4 Addition



MRHY4A adder cell



Four-digit MRHY4A

Minimally Redundant Hybrid Radix-4 Addition (mrHY4A)

- Minimally redundant numbers are based on digit set $D_{<4.2>}$.

$$S_{<4.2>} = X_{<4.2>} - Y_4$$

- The first step computes:

$$x_i + y_i = 4t_i + u_i$$

Replacing the respective binary codes from the table the following is obtained :

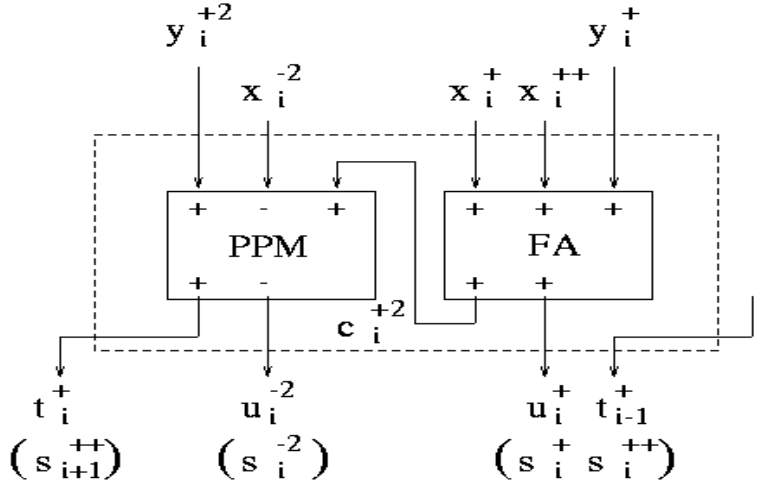
$$(-2x_i^{-2} + 2y_i^{+2}) + (x_i^{+} + x_i^{++} + y_i^{+}) = 4t_i^{+} - 2u_i^{-2} + u_i^{+}$$

A mrHY4A cell consisting of one PPM adder and a full adder is used to compute the above.

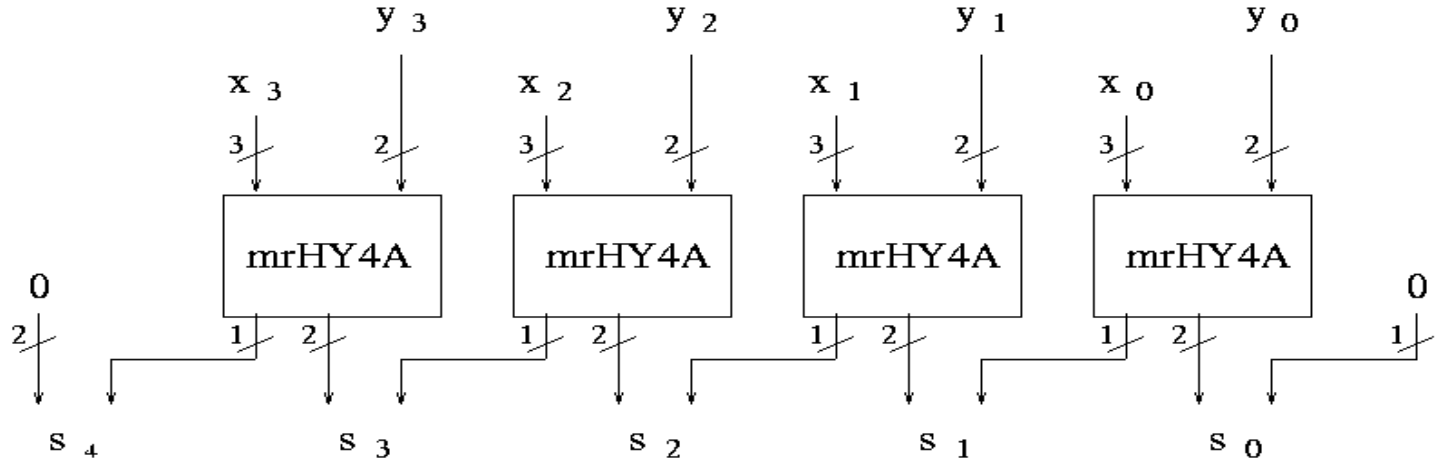
- Step 2 computes computes $s_i = t_{i-1} + u_i$. Replacing s_i, u_i, and t_{i-1} by corresponding binary codes leads to $s_i^{-2} = u_i^{-2}$, $s_i^{++} = t_{i-1}^{+}$ and $s_i^{+} = u_i^{+}$.

Digit	Radix 4 Digit Set	Binary Code
x_i	$\{\bar{2}, \bar{1}, 0, 1, 2\}$	$-2x_i^{-2} + x_i^{+} + x_i^{++}$
y_i	$\{0, 1, 2, 3\}$	$2y_i^{+2} + y_i^{+}$
$p_i = x_i + y_i$	$\{\bar{2}, \bar{1}, 0, 1, 2, 3, 4, 5\}$	$4t_i + u_i$
u_i	$\{\bar{2}, \bar{1}, 0, 1\}$	$2u_i^{+2} - 2u_i^{-2} - u_i^{-}$
t_i	$\{0, 1\}$	t_i^{+}
$s_i = u_i + t_{i-1}$	$\{\bar{2}, \bar{1}, 0, 1, 2\}$	$2s_i^{-2} + s_i^{+} + s_i^{++}$

Digit sets involved in Minimally Redundant Hybrid Radix-4 Addition



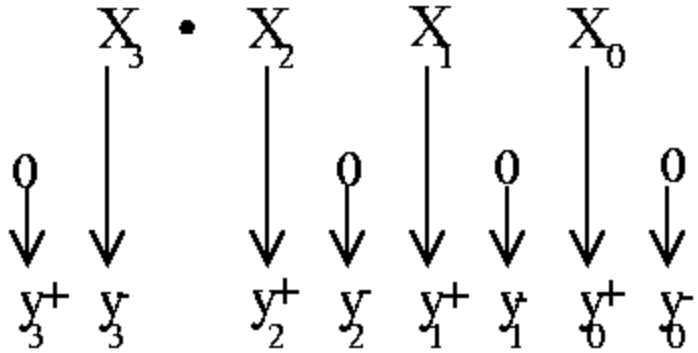
mrHY4A adder cell



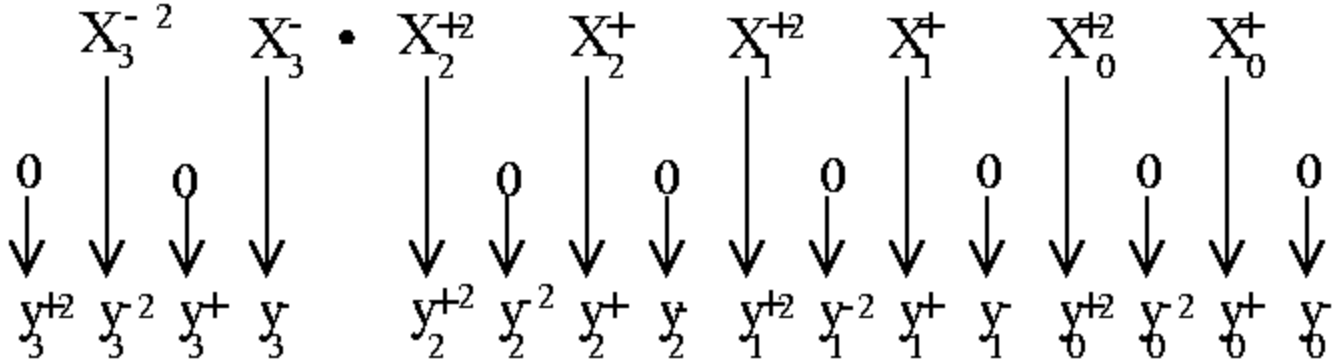
Four-digit mrHY4A

Non-redundant to Redundant Conversion

- Radix-2 Representation : A non-redundant number $X = x_3.x_2.x_1.x_0$ can be converted to a redundant number $Y = y_3.y_2.y_1.y_0$, where each digit y_i is encoded as y_i^+ and y_i^- as shown below:

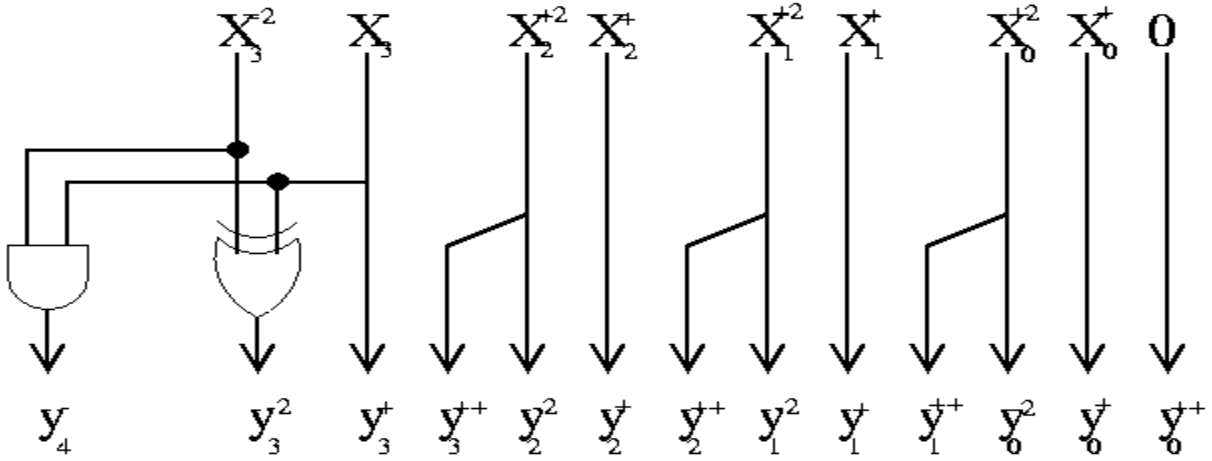


- Radix-4 representation :
  - <u>radix-4 maximally redundant number</u>: X is a radix-4 complement number, whose digits x_i are encoded using 2 wires as $x_i = 2x_i^{+2} + x_i^{+}$. Its corresponding maximally redundant number Y is encoded using $y_i = 2y_i^{+2} - 2y_i^{-2} + y_i^{+} - y_i^{-}$. The sign digit x_3 can take values -3, -2, -1 or 0, and is encoded using $x_3 = -2x_3^{-2} - x_3^{-}$.



– <u>radix-4 minimally redundant number</u>: X is a radix-4 complement number, whose digits x_i are encoded using 2 wires as $x_i = 2x_i^{+2} + x_i^{+}$. Its corresponding minimally redundant number Y is encoded using $y_i = -2y_i^{-2} + y_i^{+} + y_i^{++}$. To convert radix-r number x to redundant number $y_{<r,\alpha>}$, the digits in the range $[\alpha, r - 1]$ are encoded using a transfer digit 1 and a corresponding digit $x_i - r$ where x_i is the i^{th} digit of x. Thus,

$$2x_i^{+2} + x_i^{+} = 4x_i^{+2} - 2x_i^{+2} + x_i^{+}$$
$$= y_{i+1}^{++} - 2y_i^{-2} + y_i^{+}$$



Chapter 15: Numerical Strength Reduction

Keshab K. Parhi

- Sub-expression elimination is a numerical transformation of the constant multiplications that can lead to efficient hardware in terms of area, power and speed.
- Sub-expression can only be performed on constant multiplications that operate on a common variable.
- It is essentially the process of examining the shift and add implementations of the constant multiplications and finding redundant operations.
- Example: $a \times x$ and $b \times x$, where a = 001101 and b = 011011 can be performed as follows:
  - $a \times x = 000100 \times x + 001001 \times x$
  - $b \times x = 010010 \times x + 001001 \times x = (001001 \times x) << 1 + (001001 \times x)$.
  - The term $001001 \times x$ needs to be computed only once.
  - So, multiplications were implemented using 3 shifts and 3 adds as opposed to 5 shifts and 5 adds.

Multiple Constant Multiplication(MCM)

The algorithm for MCM uses an iterative matching process that consists of the following steps:

- Express each constant in the set using a binary format (such as signed, unsigned, 2's complement representation).
- Determine the number of bit-wise matches (non-zero bits) between all of the constants in the set.
- Choose the best match.
- Eliminate the redundancy from the best match. Return the remainders and the redundancy to the set of coefficients.
- Repeat Steps 2-4 until no improvement is achieved.

Example:

Constant	Value	Unsigned
a	237	11101101
b	182	10110110
c	93	01011101

Binary representation of constants

Constant	Unsigned
Rem. of a	10100000
b	10110110
Rem. of c	00010000
Red. of a,c	01001101

Updated set of constants
1st iteration

Constant	Unsigned
Rem. of a	00000000
Rem. of b	00010110
Rem. of c	00010000
Red. of a,c	01001101
Red. of Rem a,b	10100000

Updated set of constants
2nd iteration

Linear Transformations

- A general form of linear transformation is given as:

$$\mathbf{y = T*x}$$

where, T is an m by n matrix, y is length-m vector and x is a length-n vector. It can also be written as:

$$y_i = \sum_{j=1}^{n} t_{ij} x_j, i = 1, \ldots, m$$

- The following steps are followed:
  - Minimize the number of shifts and adds required to compute the products $t_{ij} x_j$ by using the iterative matching algorithm.
  - Formation of unique products using the sub-expression found in the 1st step.
  - Final step involves the sharing of additions, which is common among the y_i's. This step is very similar to the MCM problem.

Example:

$$T = \begin{bmatrix} 7 & 8 & 2 & 13 \\ 12 & 11 & 7 & 13 \\ 5 & 8 & 2 & 15 \\ 7 & 11 & 7 & 11 \end{bmatrix}$$

- The constants in each column multiply to a common variable. For Example x_1 is multiplied to the set of constants [7, 12, 5, 7].
- Applying iterative matching algorithm the following table is obtained.

Column 1	Column 2	Column 3	Column 4
0101	1000	0010	1001
0010	1011	0111	0100
1100			0010

- Next, the unique products are formed as shown below:

$$p_1 = 0101*x_1,\ p_2 = 0010*x_1,\ p_3 = 1100*x_1$$

$$p_4 = 1000*x_2,\ p_5 = 1011*x_2,$$

$$p_6 = 0010*x_3,\ p_7 = 0111*x_3$$

$$p_8 = 1001*x_4,\ p_9 = 0100*x_4,\ p_{10} = 0010*x_4$$

- Using these products the y_i's are as follows:

$$y_1 = p_1 + p_2 + p_4 + p_6 + p_8 + p_9;$$

$$y_2 = p_3 + p_5 + p_7 + p_8 + p_9;$$

$$y_3 = p_1 + p_4 + p_6 + p_8 + p_9 + p_{10};$$

$$y_4 = p_1 + p_2 + p_5 + p_7 + p_8 + p_{10};$$

- This step involves sharing of additions which are common to all y_i's. For this each y_i is represented as k bit word ($1 \leq k \leq 10$), where each of the k products formed after the 2^{nd} step represents a particular bit position. Thus,

$$y_1 = 1101010110,\ y_2 = 0010101110,$$

$$y_3 = 1001010111,\ y_4 = 1100101101.$$

- Applying iterative matching algorithm to reduce the number of additions required for y_i's we get:

$$y_1 = p_2 + (p_1 + p_4 + p_6 + p_8 + p_9);$$

$$y_2 = p_3 + p_9 + (p_5 + p_7 + p_8);$$

$$y_3 = p_{10} + (p_1 + p_4 + p_6 + p_8 + p_9);$$

$$y_4 = p_1 + p_2 + p_{10} + (p_5 + p_7 + p_8);$$

- The total number of additions are reduced from 35 to 20.

Polynomial Evaluation

Evaluating the polynomial:

$$x^{13} + x^7 + x^4 + x^2 + x$$

- Without considering the redundancies this polynomial evaluation requires 22 multiplications.
- Examining the exponents and considering their binary representations:

  1 = 0001, 2 = 0010, 4 = 0100, 7 = 0111, 13 = 1101.

- x^7 can be considered as $x^4 \times x^2 \times x^1$. Applying sub-expression sharing to the exponents the polynomial can be evaluated as follows:

$$x^8 \times (x^4 \times x) + x^2 \times (x^4 \times x) + x^4 + x^2 + x$$

- The terms x^2, x^4 and x^8 each require one multiplication as shown below:

$$x^2 = x \times x, \quad x^4 = x^2 \times x^2, \quad x^8 = x^4 \times x^4$$

- Thus, we require 6 instead of 22 multiplications.

Sub-expression Sharing in Digital Filters

- Example of common sub-expression elimination within a single multiplication :

$$y = 0.10\bar{1}000101\bar{1} * x.$$

This may be implemented as:

$$y = (x \gg 1) - (x \gg 3) + (x \gg 7) - (x \gg 9).$$

Alternatively, this can be implemented as,

$$x2 = x - (x \gg 2)$$

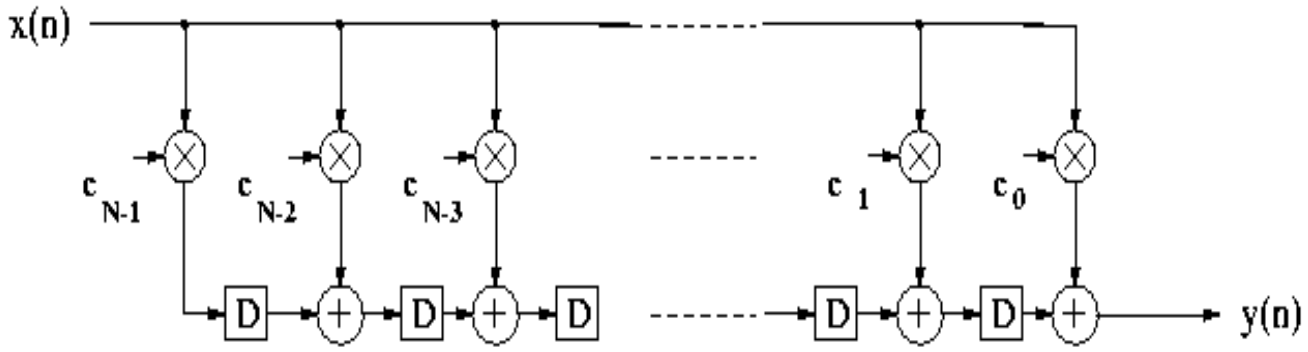
$$Y = (x2 \gg 1) + (x2 \gg 7)$$

which requires one less addition.

- In order to realize the sub-expression elimination transformation, the N-tap FIR filter:

$$y(n) = c_0x(n) + c_1x(n-1) + \ldots + c_0x(n-N+1)$$

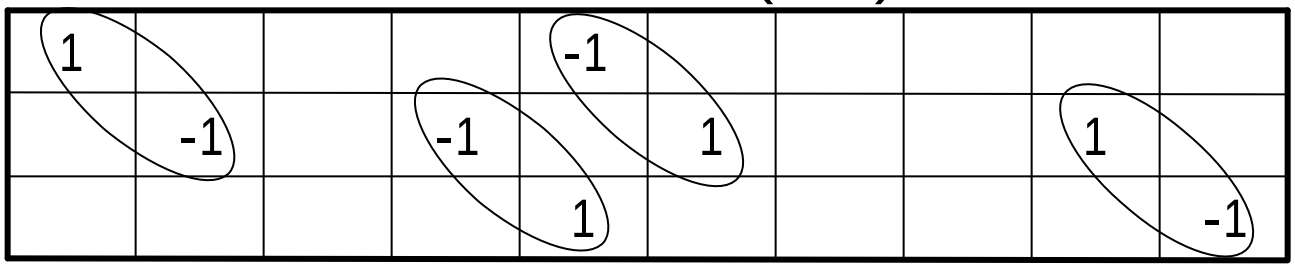
must be implemented using transposed direct-form structure also called <u>data-broadcast filter structure</u> as shown below:



- Represent a filter operation by a table (matrix) $\{x_{ij}\}$, where the rows are indexed by delay i and the columns by shift j, i.e., the row i is the coefficient c_i for the term x(n-i), and the column 0 in row i is the msb of c_i and column W-1 in row i is the lsb of c_i , where W is the word length.
- The row and column indexing starts at 0.
- The entries are 0 or 1 if 2's complement representation is used and $\{1, 0, \bar{1}\}$ if CSD is used.
- A non-zero entry in row i and column j represents x(n-i) >> j. It is to be added or subtracted according to whether the entry is +1 or –1.

Example:

$$y(n) = 1.000\overline{1}00000*x(n) + \overline{0.1}0\overline{1}010010*x(n-1) + 0.000100001*x(n-2)$$



This filter has 8 non-zero terms and thus requires 7 additions. But, the sub-expressions x1 + x1[-1] >> 1 occurs 4 times in shifted and delayed forms by various amounts as circled. So, the filter requires 4 adds.

$$x2 = x1 - x1[-1] >> 1$$

$$y = x2 - (x2 >> 4) - (x2[-1] >> 3) + (x2[-1] >> 8)$$

An alternative realization is :

$$x2 = x1 - (x1 >> 4) - (x1[-1] >> 3) + (x1[-1] >> 8)$$

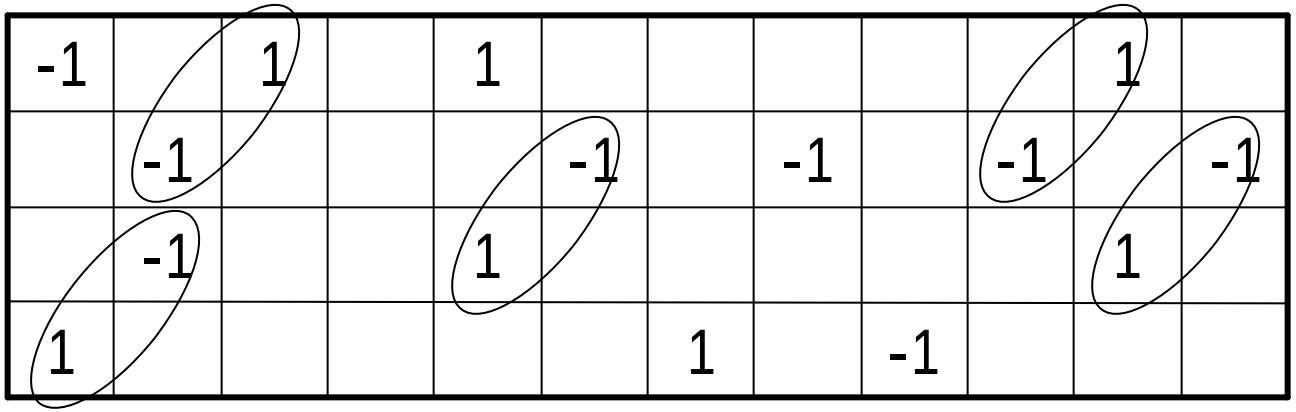
$$y = x2 - (x2[-1] >> 1).$$

Example:

$$y(n) = \bar{1}.01010000010*x(n) + 0.\bar{1}000\bar{1}0\bar{1}0\bar{1}01*x(n-1)$$
$$+ 0.\bar{1}0010000010*x(n-2) + 1.00000\bar{1}0\bar{1}000*x(n-4)$$

The substructure matching procedure for this design is as follows:

- Start with the table containing the coefficients of the FIR filter. An entry with absolute value of 1 in this table denotes add or subtract of x1. Identify the best sub-expression of size 2.



- Remove each occurrence of each sub-expression and replace it by a value of 2 or –2 in place of the first (row major) of the 2 terms making up the sub-expression.

-1		2		1						2	
					-2		-1		-1		-2
	-2										
						1		-1			

- Record the definition of the sub-expression. This may require a negative value of shift which will be taken care of later.

$$x3 = x1 - x1[-1] >> (-1)$$

- Continue by finding more sub-expressions until done.

-1		3								2	
					-3						-2
	-2										
						1		-1			

5. Write out the complete definition of the filter.

$$x2 = x1 - x1[-1] >> (-1)$$
$$x3 = x2 + x1 >> 2$$
$$y = -x1 + x3 >> 2 + x2 >> 10 - x3[-1] >> 5 - x2[-1] >> 11$$
$$-x2[-2] >> 1 + x1[-3] >> 6 - x1[-3] >> 8.$$

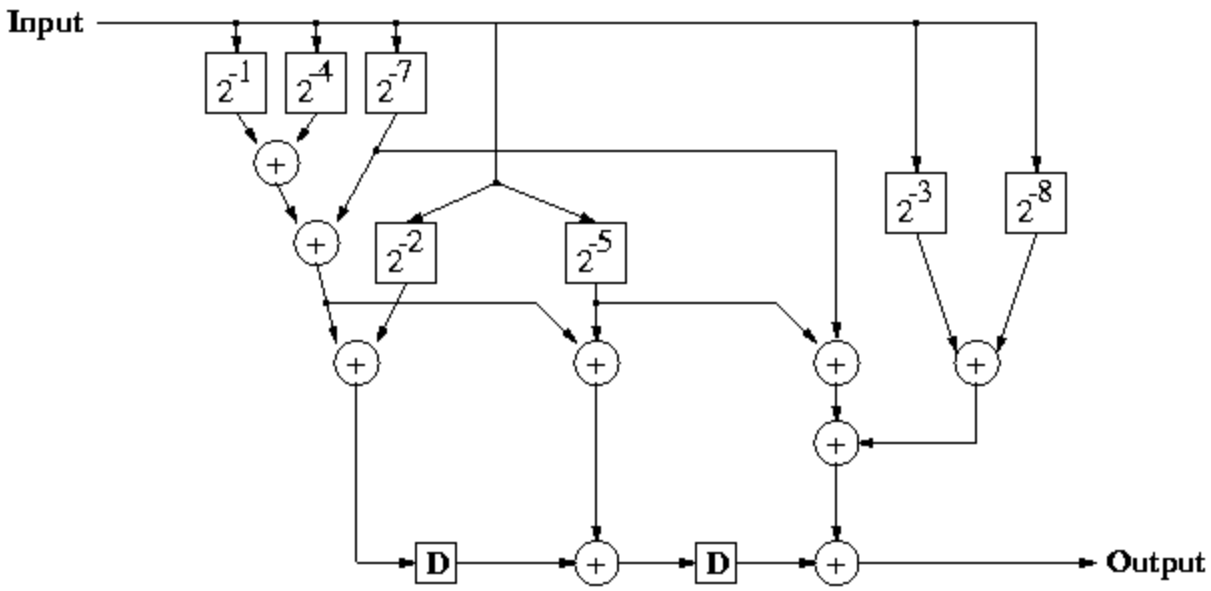
- If any sub-expression definition involves negative shift, then modify the definition and subsequent uses of that variable to remove the negative shift as shown below:

$$x2 = x1 >> 1 - x1[-1]$$

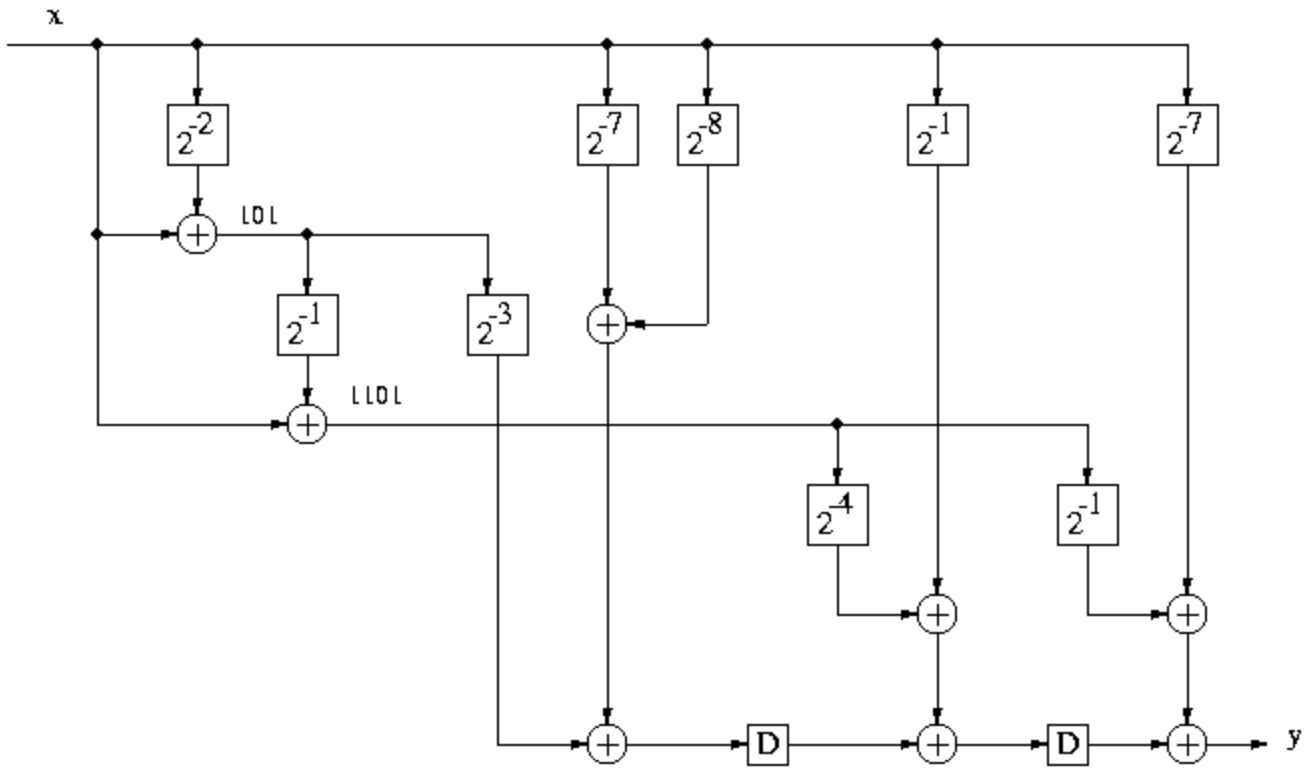
$$x3 = x2 + x1 >> 3$$

$$y = -x1 + x3 >> 1 + x2 >> 9 - x3[-1] >> 4 - x2[-1] >> 10$$

$$- x2[-2] + x1[-3] >> 6 - x1[-3] >> 8.$$



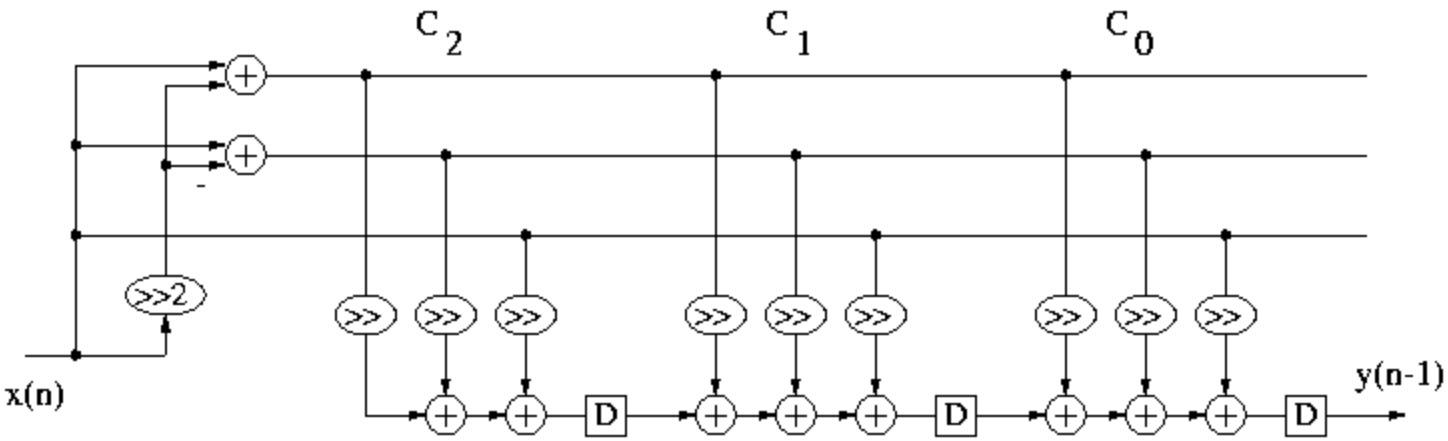
3-tap FIR filter with sub-expression sharing for 3-tap FIR filter with coefficients $c_2 = 0.11010010$, $c_1 = 0.10011010$ and $c_0 = 0.00101011$.
This requires 7 shifts and 9 additions compared to 12 shifts and 11 additions.



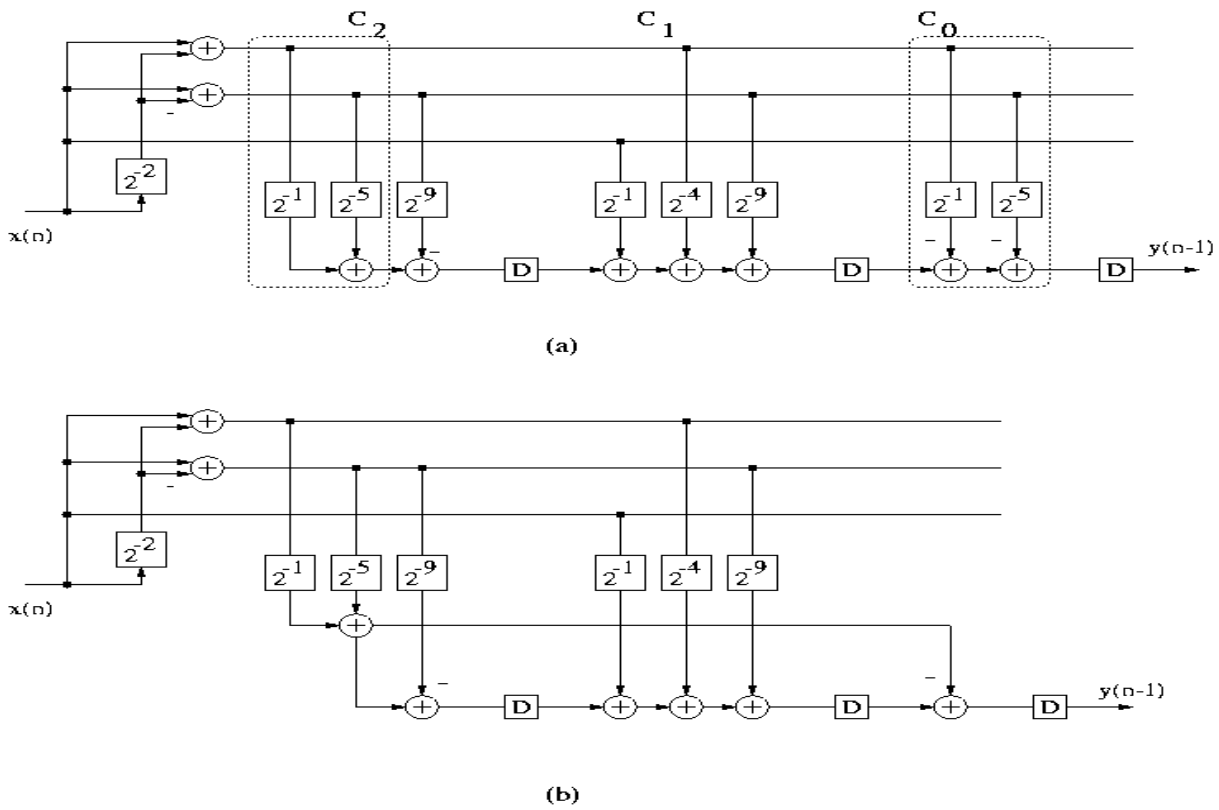
3-tap FIR filter with sub-expression sharing requiring 8 additions as compared to 9 in the previous implementation.

Using 2 most common sub-expressions in CSD representation

- $x - x >> 2$ and $x + x >> 2$ are the 2 most common sub-expressions in CSD representation.



An FIR filter using the term sharing, where the two most common sub-expressions in CSD numbers 101 and $10\bar{1}$, together with isolated 1 are shared among all filter coefficients.

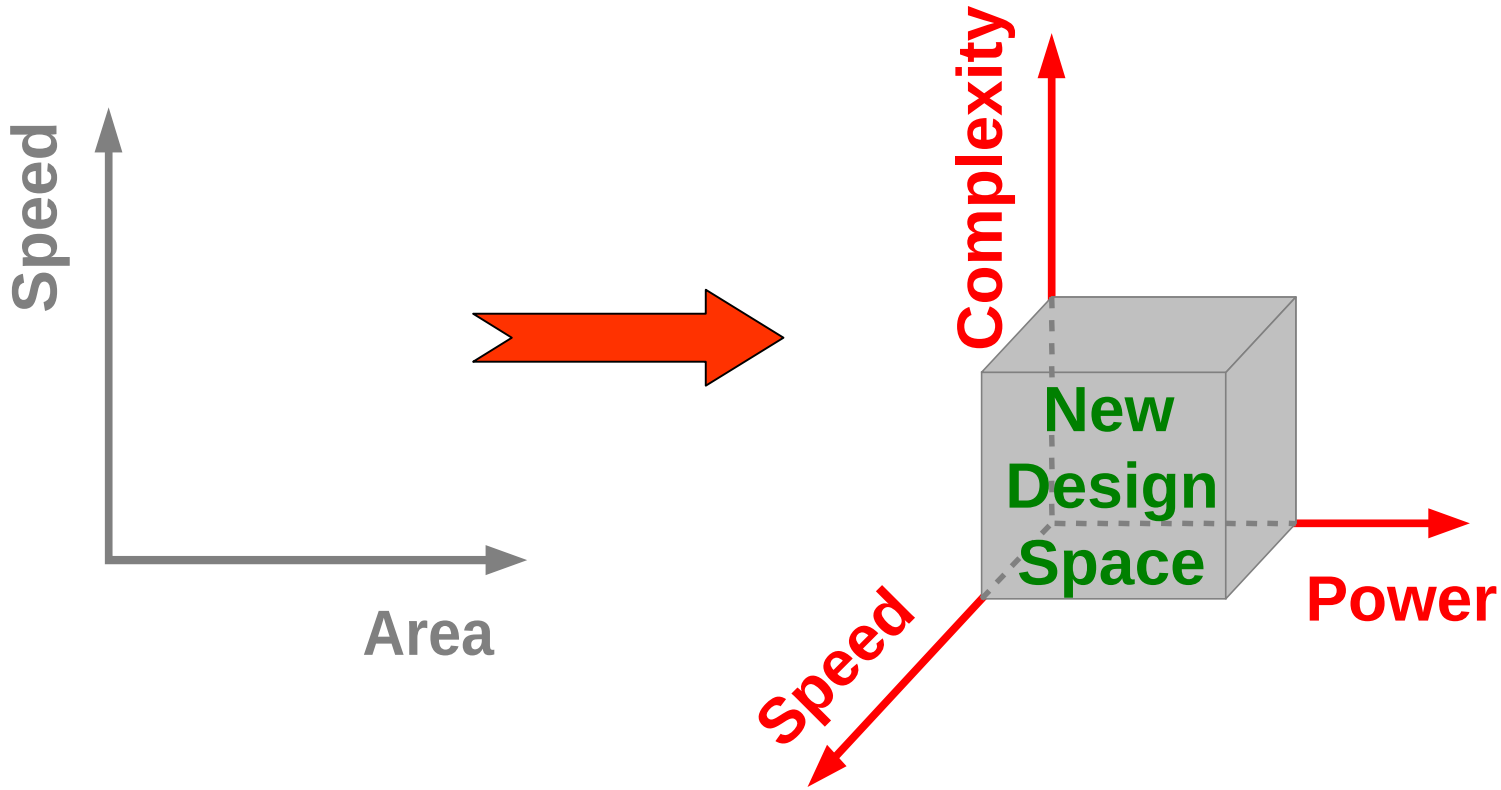


3-tap FIR filter with coefficients $c_2 = 0.101010\bar{1}010\bar{1}$, $c_1 = 0.10010100101\bar{1}$ and $c_0 = 0.\bar{1}0\bar{1}0\bar{1}010000$. 2 additions in the dotted square in (a) are shared in (b). Filter requires only 7 additions and 7 shifts as opposed to 12 adds and 12 shifts in standard multiplierless implementation.

Chapter 17: Low-Power Design

Keshab K. Parhi and Viktor Owall

IC Design Space



VLSI Digital Signal Processing Systems

- Technology trends:
  - 200-300M chips by 2010 (0.07 micron CMOS)
- Challenges:
  - Low-power DSP algorithms and architectures
  - Low-power dedicated / programmable systems
  - Multimedia & wireless system-driven architectures
  - Convergence of Voice, Video and Data
  - LAN, MAN, WAN, PAN
  - Telephone Lines, Cables, Fiber, Wireless
  - Standards and Interoperability

Power Consumption in DSP

- Low performance portable applications:
  - Cellular phones, personal digital assistants
  - Reasonable battery lifetime, low weight

- High performance portable systems:
  - Laptops, notebook computers
- Non-portable systems:
  - Workstations, communication systems
  - DEC alpha: 1 GHz, 120 Watts
  - Packaging costs, system reliability

Power Dissipation

Two measures are important

- **Peak power (Sets dimensions)**

$$P_{peak} = V_{DD} \times i_{DDmax}$$

- **Average power (Battery and cooling)**

$$P_{av} = \frac{V_{DD}}{T} \int_0^T i_{DD}(t)\, dt$$

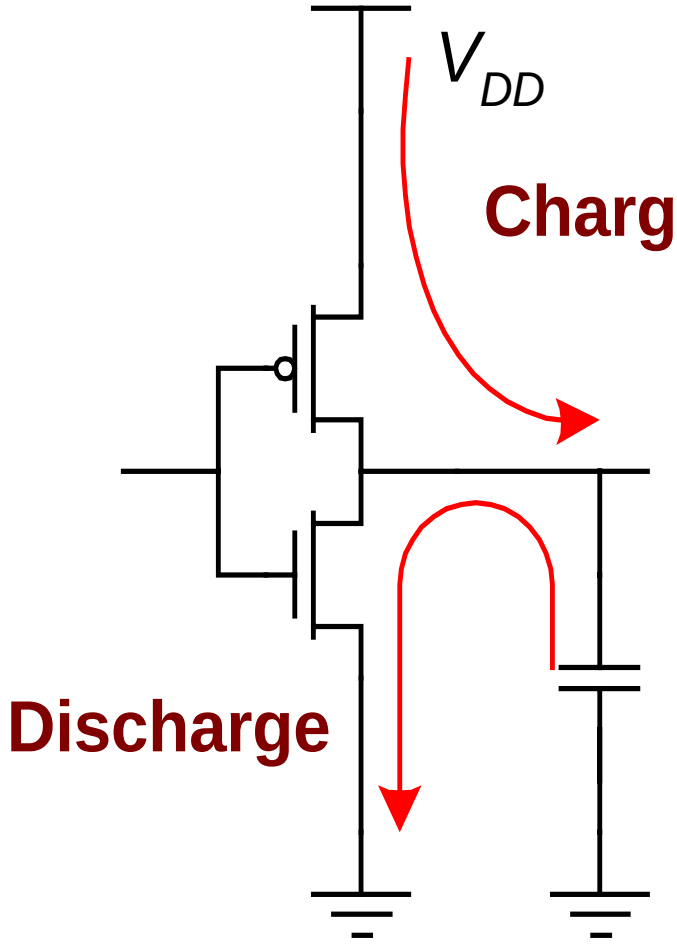
CMOS Power Consumption

$$P_{tot} = P_{dyn} + P_{sc} + P_{leakage} =$$

$$= \alpha f C_L V_{DD}^2 \quad + \quad V_{DD} I_{sc} + \quad I_{leakage} V_{DD}$$

α = probability for switching

Dynamic Power Consumption



Energy charged in a capacitor

$$E_C = CV^2/2 = C_L V_{DD}^2/2$$

Energy E_c is also discharged, i.e.

$$E_{tot} = C_L V_{DD}^2$$

Power consumption

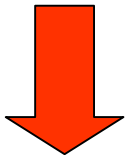
$$P = C_L V_{DD}^2 f$$

Off-Chip Connections have High Capacitive Load

Reduced off Chip Data Transfers by

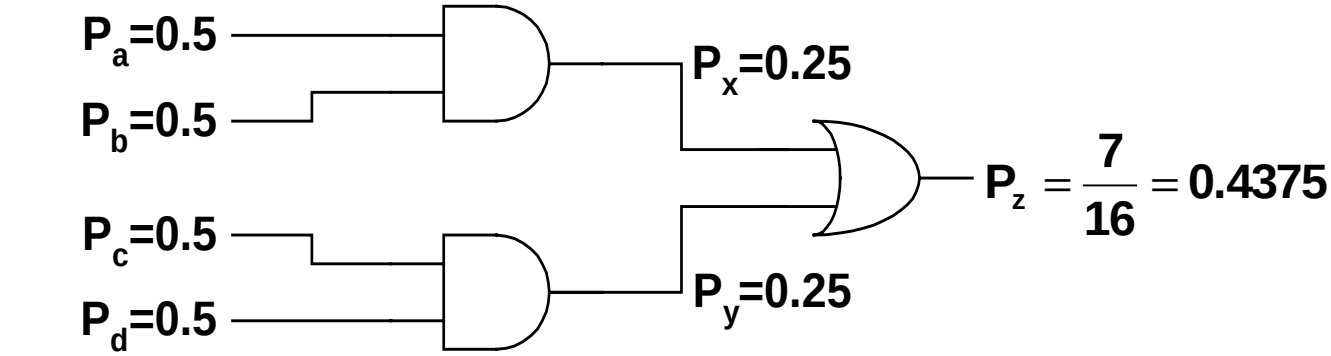
System Integration

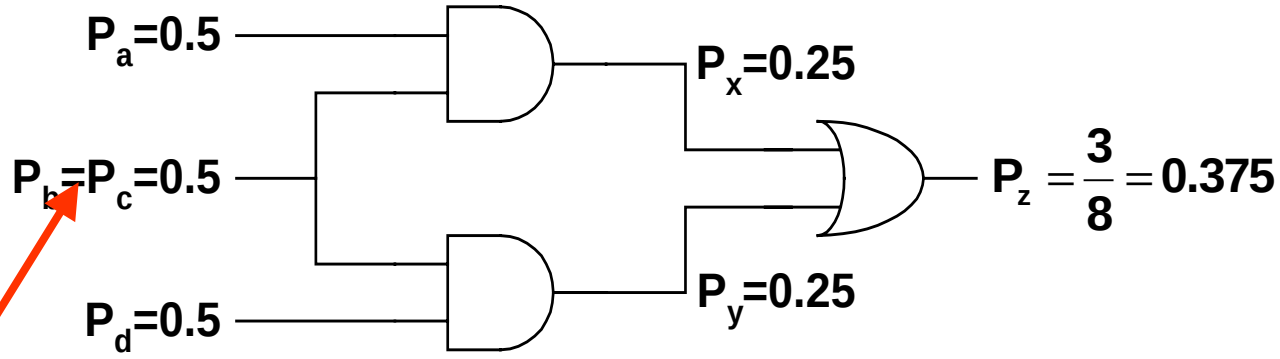
Ideally a Single Chip Solution



Reduced Power Consumption

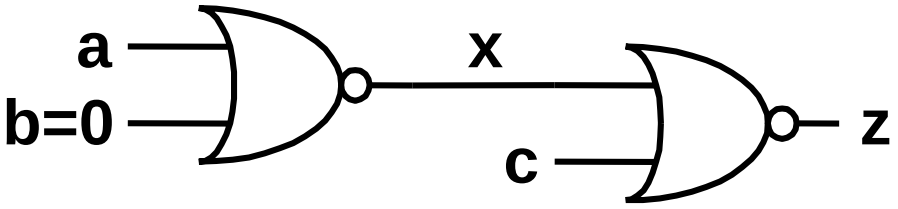
Switching Activity (α): Example

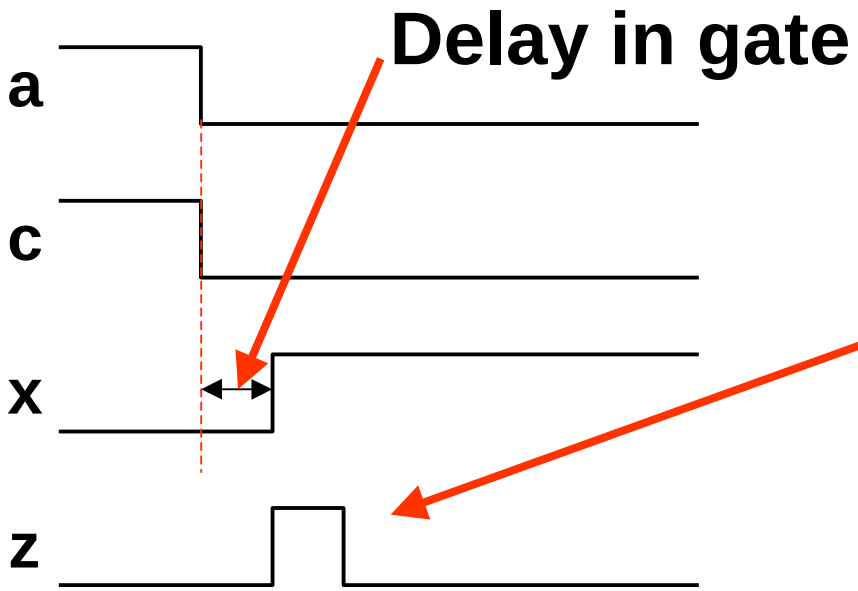




Due to correlation

Increased Switching Activity due to Glitching

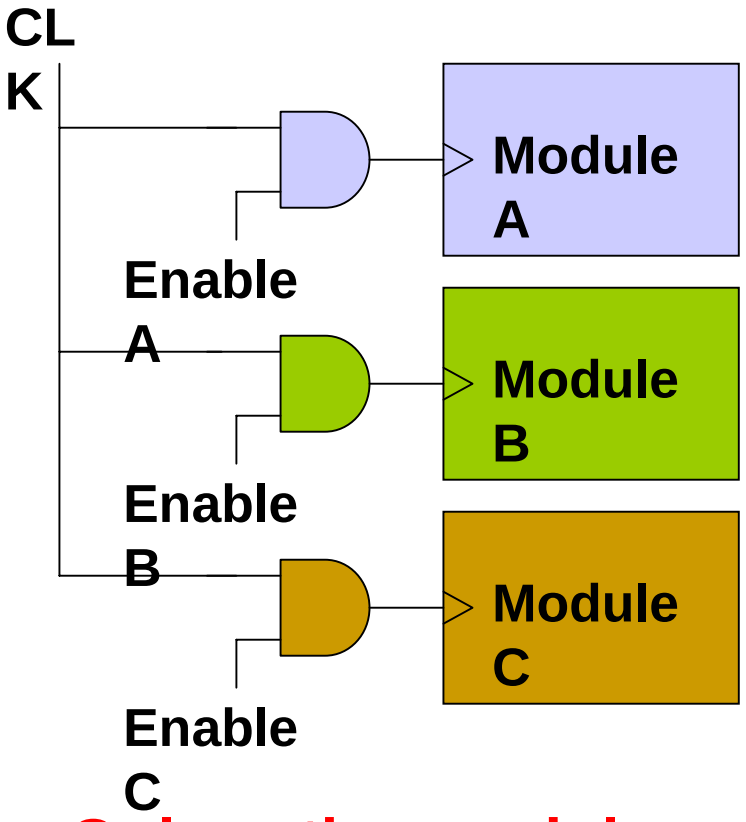




Extra transition due to race

Dissipates energy

Clock Gating and Power Down



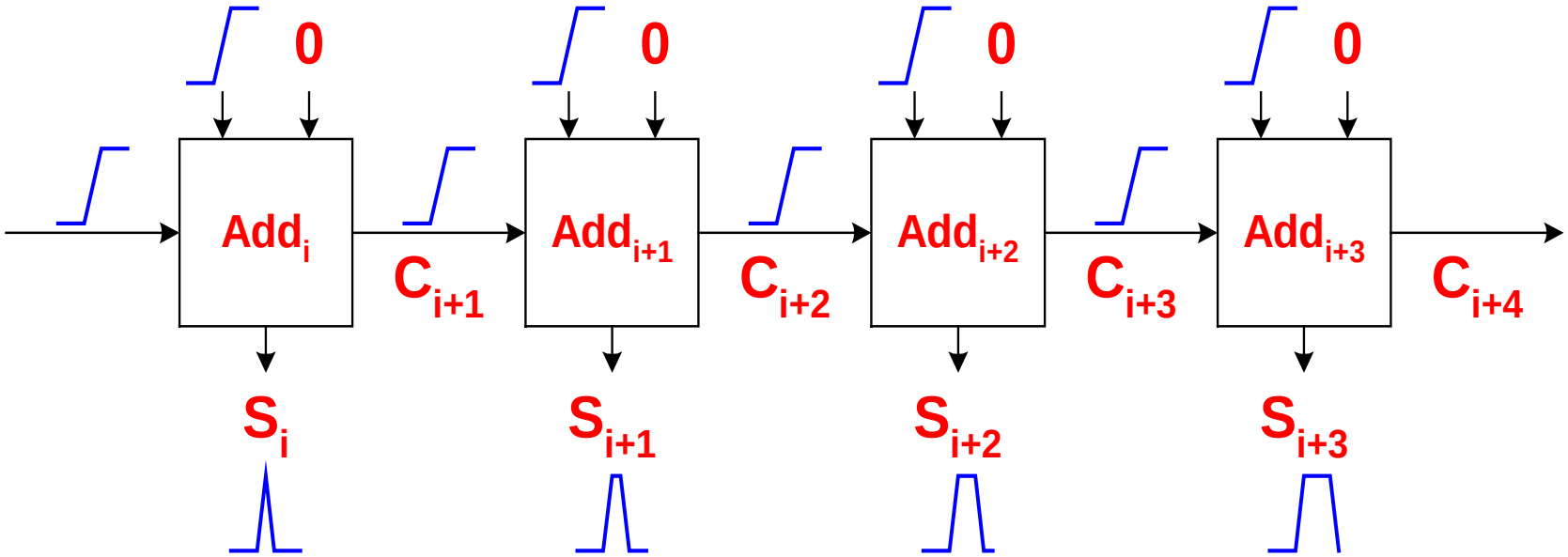
Control circuitry is needed for clock gating and power down

and

Needs wake-up

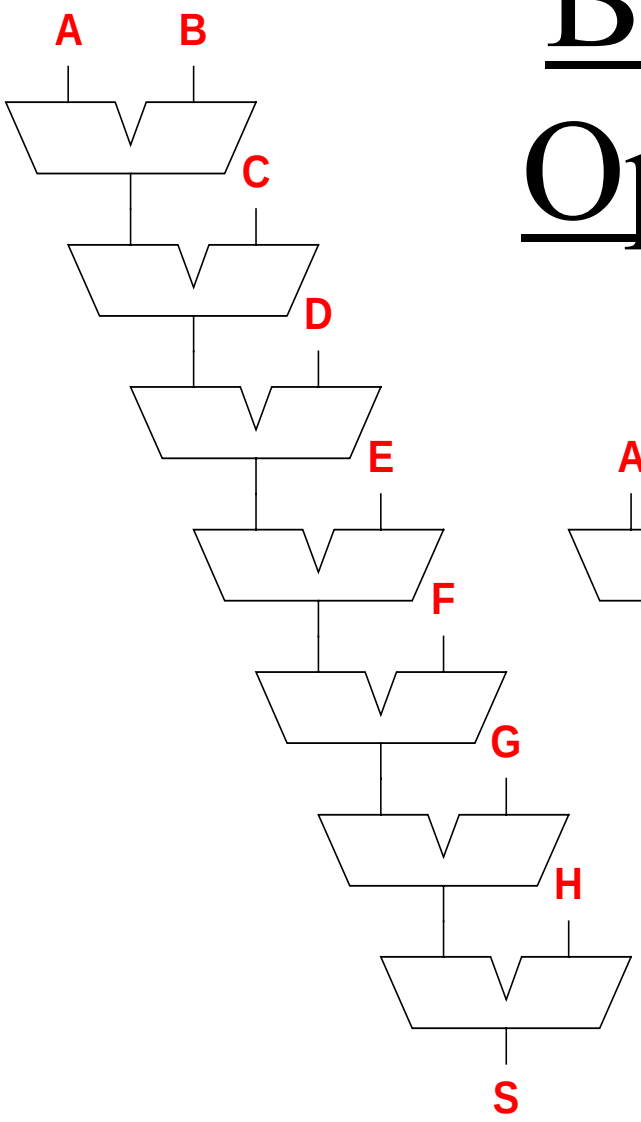
Only active modules should be clocked!

Carry Ripple

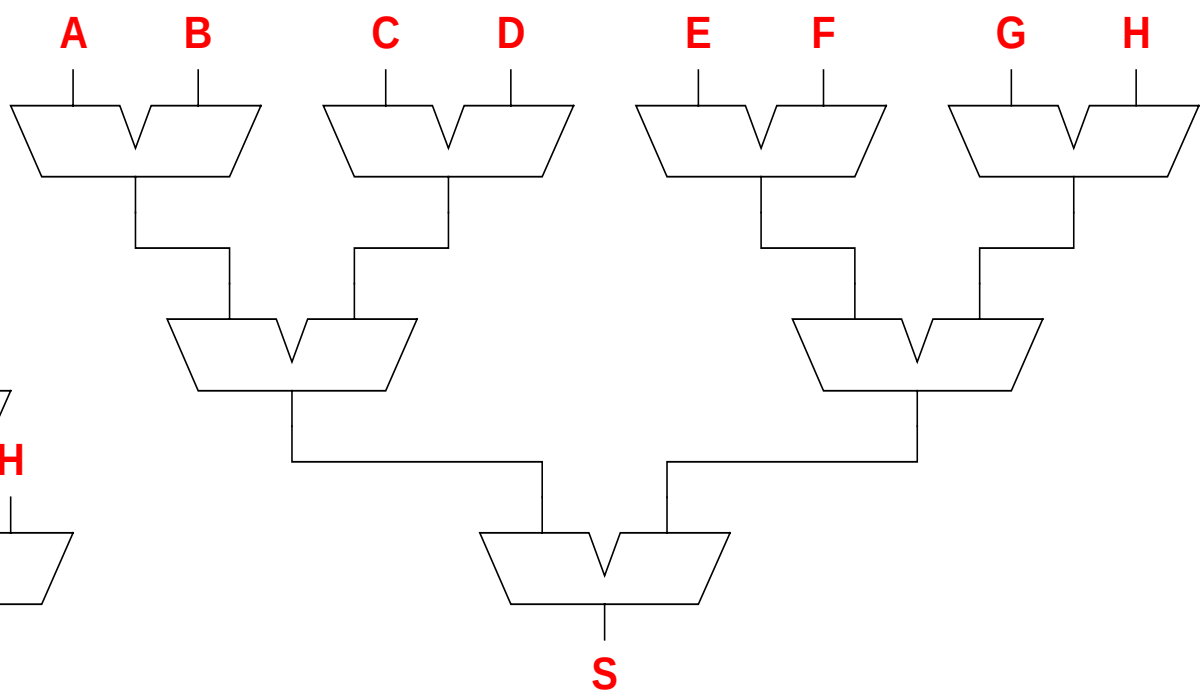


Transitions due to carry propagation

Balancing Operations



Example: Addition



Delay as function of Supply

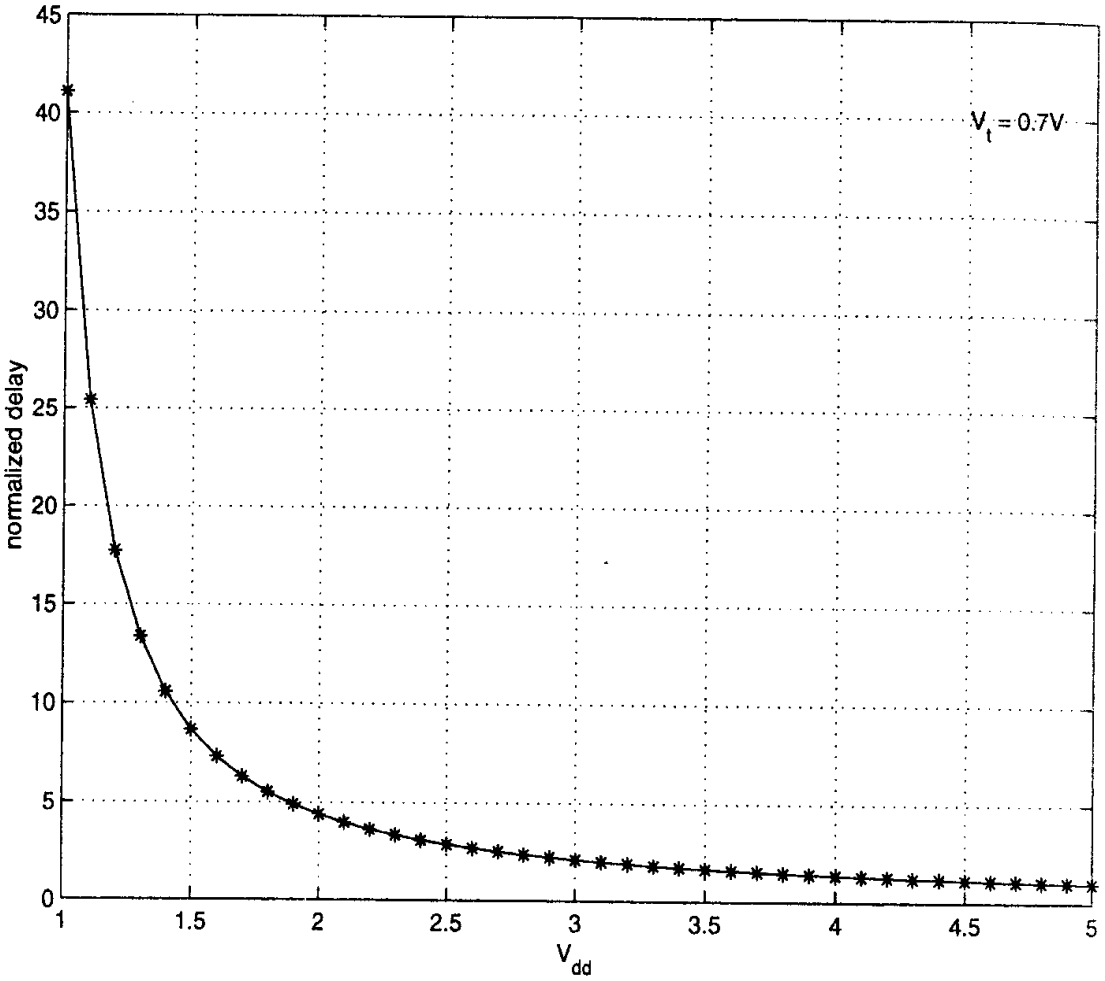


Fig. 17.3 Circuit delay as a function of supply voltage.

Delay as function of Threshold

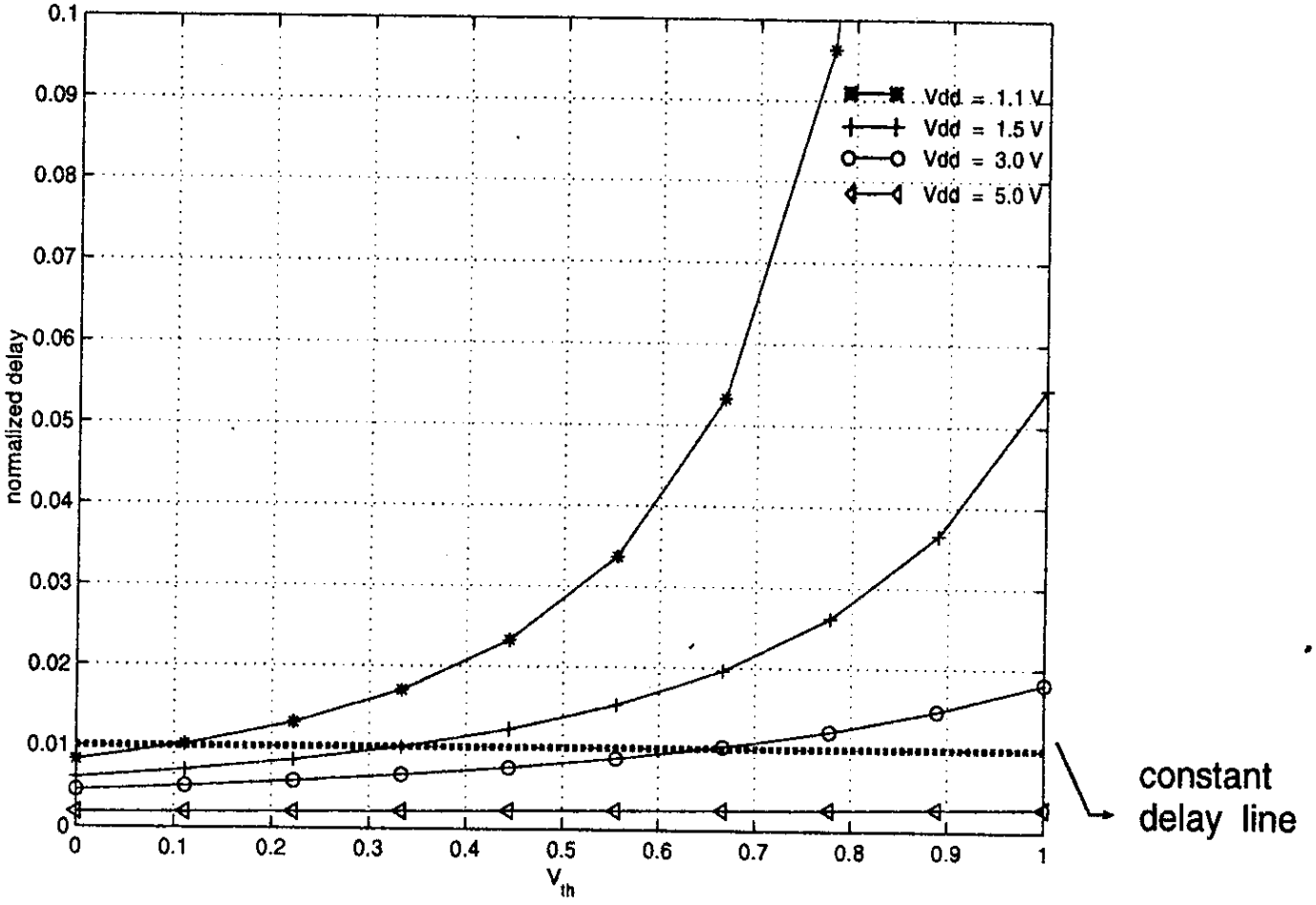
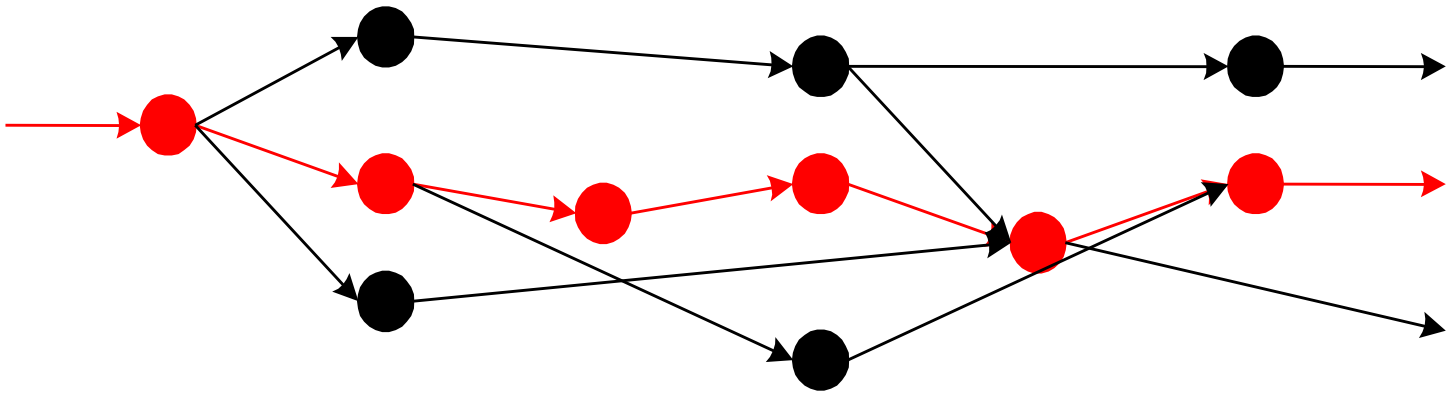


Fig. 17.4 Circuit delay as a function of supply voltage for varying threshold voltages.

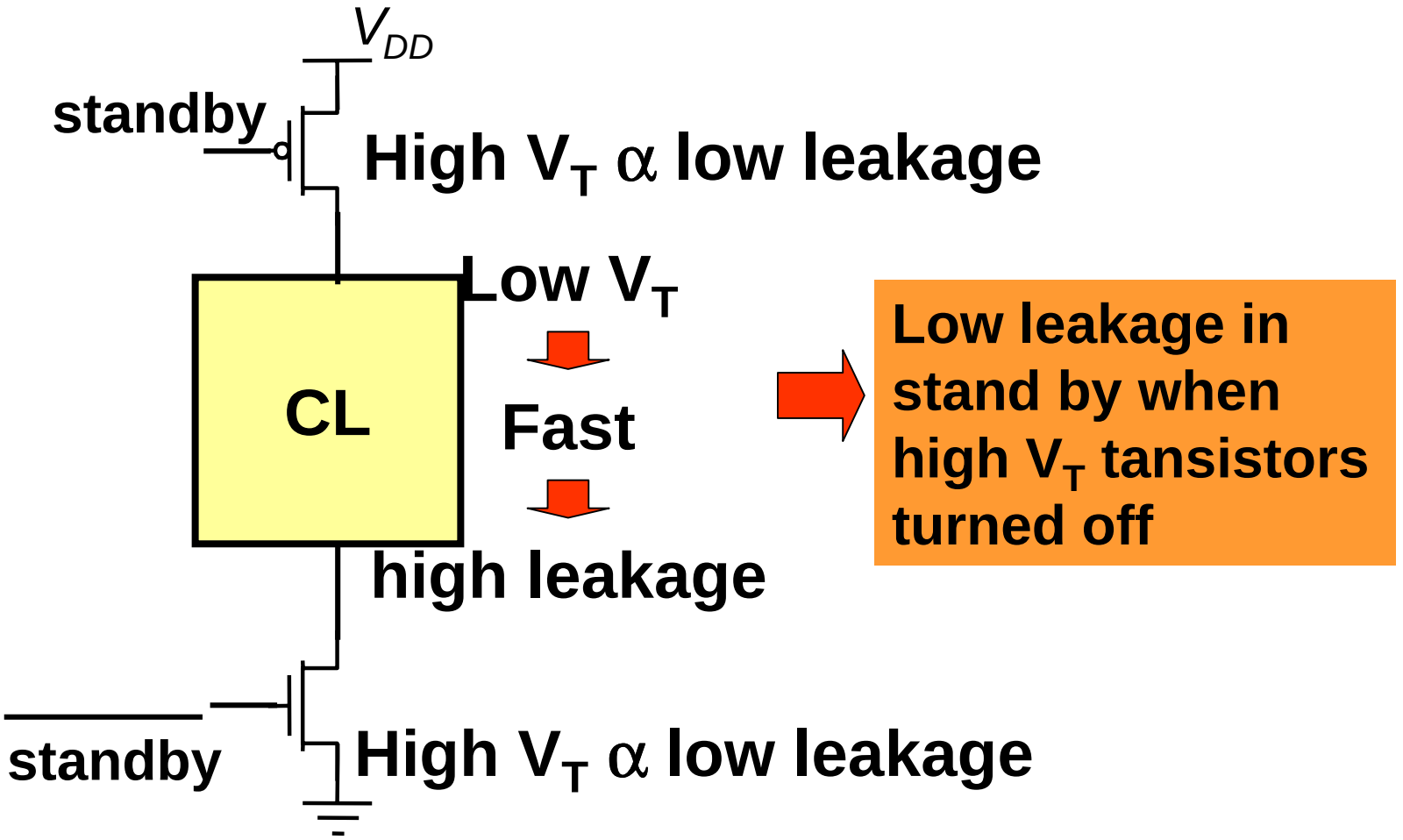
Dual V_T Technology

Reduced V_{DD} α Increased delay
Low V_T α Faster but Increased Leakage

Low V_T in critical path

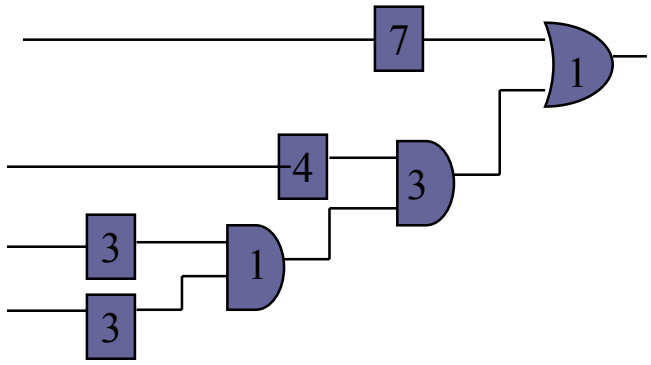


High V_T stand-by

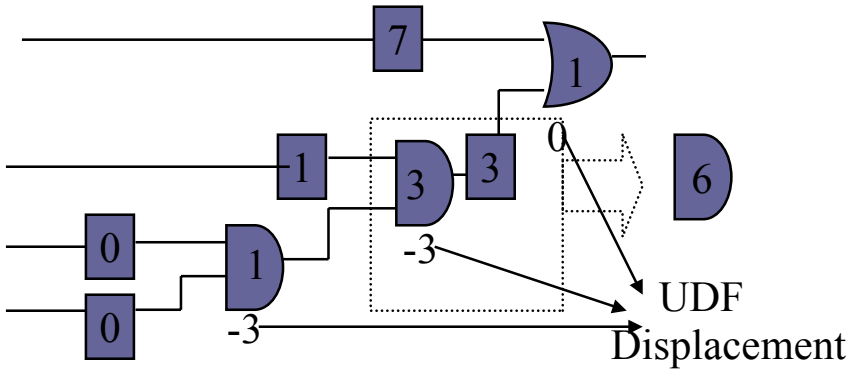


Low Power Gate Resizing

- Systematic capture and elimination of slack using fictitious entities called *Unit Delay Fictitious Buffers.*
- Replace unnecessary fast gates by slower lower power gates from an underlying gate library.
- Use a simple relation between a gate's speed and power and the UDF's in its fanout nets. Model the problem as an *efficiently solvable ILP* similar to retiming.
- In *Proceedings of ARVLSI'99* Georgia Tech.



Critical Path = 8, UDF's in Boxes



Critical Path = 8, UDF's in Boxes

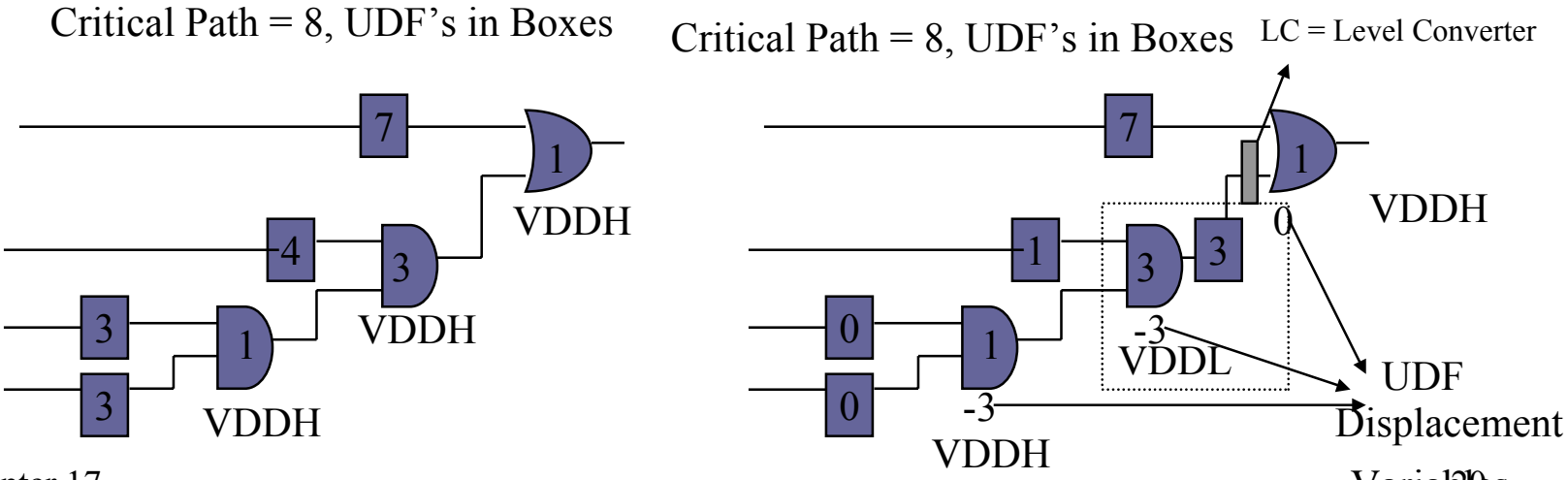
Dual Supply Voltages for Low Power

- Components on the Critical Path exhibit no slack but components off the critical path exhibit excessive slack.
- A high supply voltage VDDH for critical path components and a low supply voltage VDDL for non critical path components.
- Throughput is maintained and power consumption is lowered.

V. Sundararajan and K.K. Parhi, "Synthesis of Low Power CMOS VLSI Circuits using Dual Supply Voltages", Prof. of ACM\/IEEE Design Automation Conference, pp. 72-75, New Orleans, June 1999

Dual Supply Voltages for Low Power

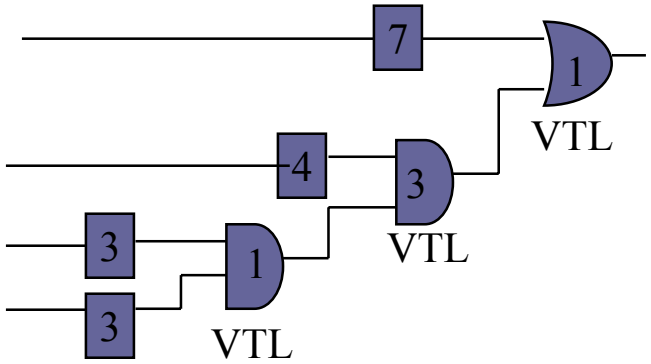
- Systematic capture and elimination of slack using fictitious entities called *Unit Delay Fictitious Buffers.*
- Switch unnecessarily fast gates to to lower supply voltage VDDL thereby saving power, critical path gates have a high supply voltage of VDDH.
- Use a simple relation between a gate's speed/power and supply voltage with the UDF's in its fanout nets. Model the problem as an *approximately solvable ILP*.

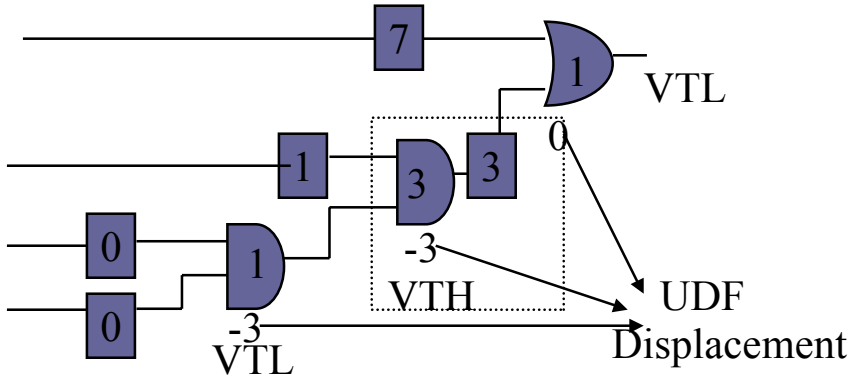


Dual Threshold CMOS VLSI for Low Power

- Systematic capture and elimination of slack using fictitious entities called *Unit Delay Fictitious Buffers.*
- Gates on the critical path have a low threshold voltage VTL and unnecessarily fast gates are switched to a high threshold voltage VTH.
- Use a simple relation between a gate's speed /power and threshold voltage with the UDF's in its fanout nets. Model the problem as an *efficiently approximable 0-1 ILP.*

Critical Path = 8, UDF's in Boxes





Critical Path = 8, UDF's in Boxes

Experimental Results

- Table :ISCAS'85 Benchmark Ckts Resizing (20 Sizes) Dual VDD Dual

				(5v, 2.4v)		Vt
Ckt	#Gates	Power Savings	CPU(s)	Power Savings	CPU(s)	Power Savings
C1908	880	15.27%	87.5	49.5%	739.05	84.92%
c2670	1211	28.91%	164.38	57.6%	1229.37	90.25%
c3540	1705	37.11%	312.51	57.7%	1743.75	83.36%
c5315	2351	41.91%	660.56	62.4%	4243.63	91.56%
c6288	2416	5.57%	69.58	62.7%	7736.05	61.75%
c7552	3624	54.05%	1256.76	59.6%	9475.1	90.90%

V. Sundararajan and K.K. Parhi, "Low Power Synthesis of Dual Threshold Voltage CMOS VLSI Circuits" Proc. of 1999 IEEE Int. Symp. on Low-Power Electronics and Design, pp. 139-144, San Diego, Aug. 1999

HEAT: Hierarchical Energy Analysis Tool

- Salient features:
  - Based on stochastic techniques
  - Transistor-level analysis
  - Effectively models glitching activity
  - Reasonably fast due to its hierarchical nature

Theoretical Background

- Signal probability:
  - S=T_{clk} / T_{gd} ,where
    T_{clk} :clock period
    T_{gd} : smallest gate delay
- Transition probability:
- Conditional probability:

$$p_{x_i}^1 = \lim_{N\to\infty} \frac{\sum_{j=1}^{NS} x_i(j)}{NS}$$

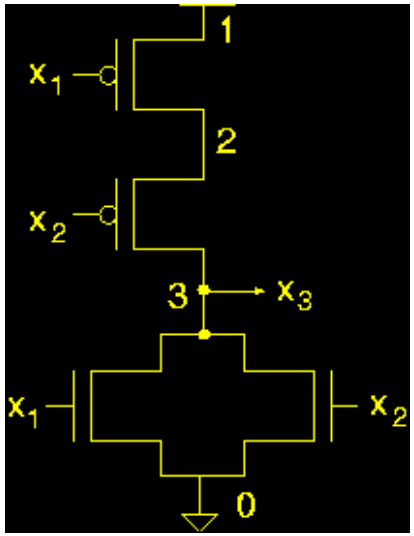
$$p_{x_i}^0 = 1 - p_{x_i}^1$$

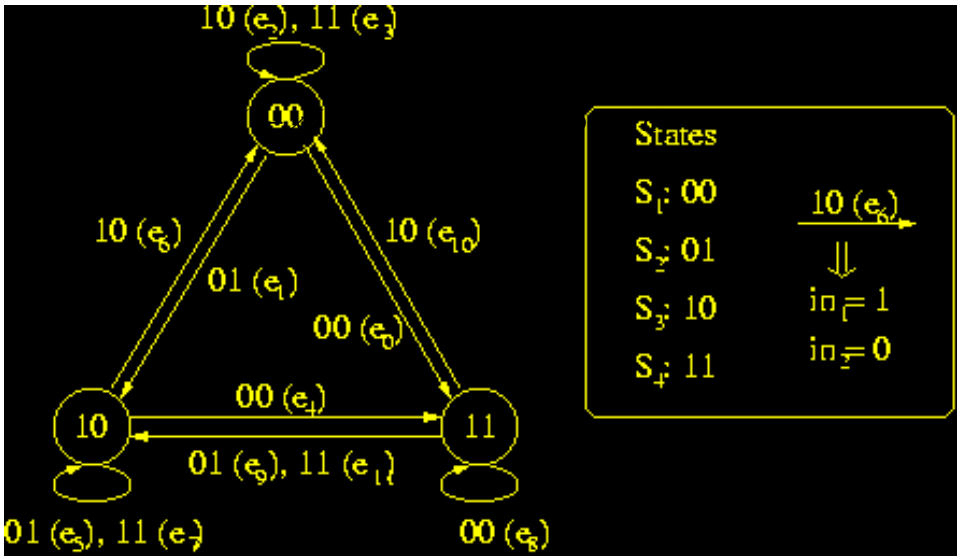
$$p_{x_i}^{1\to 0} = \lim_{N\to\infty} \frac{\sum_{j=1}^{NS} x_i(j)\overline{x_i(j+1)}}{NS}$$

$$p_{x_i}^{1\to 0} + p_{x_i}^{1\to 1} + p_{x_i}^{0\to 1} + p_{x_i}^{0\to 0} = 1$$

$$p_{x_i}^{1/0} = \frac{p_{x_i}^{0\to 1}}{p_{x_i}^{0\to 1} + p_{x_i}^{0\to 0}}$$

State Transition Diagram Modeling



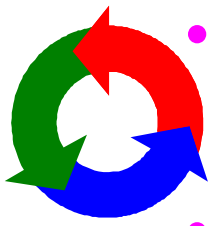


$$node_2(n+1) = (1 - x_1(n)) + x_1(n) \cdot x_2(n) \cdot node_2(n)$$

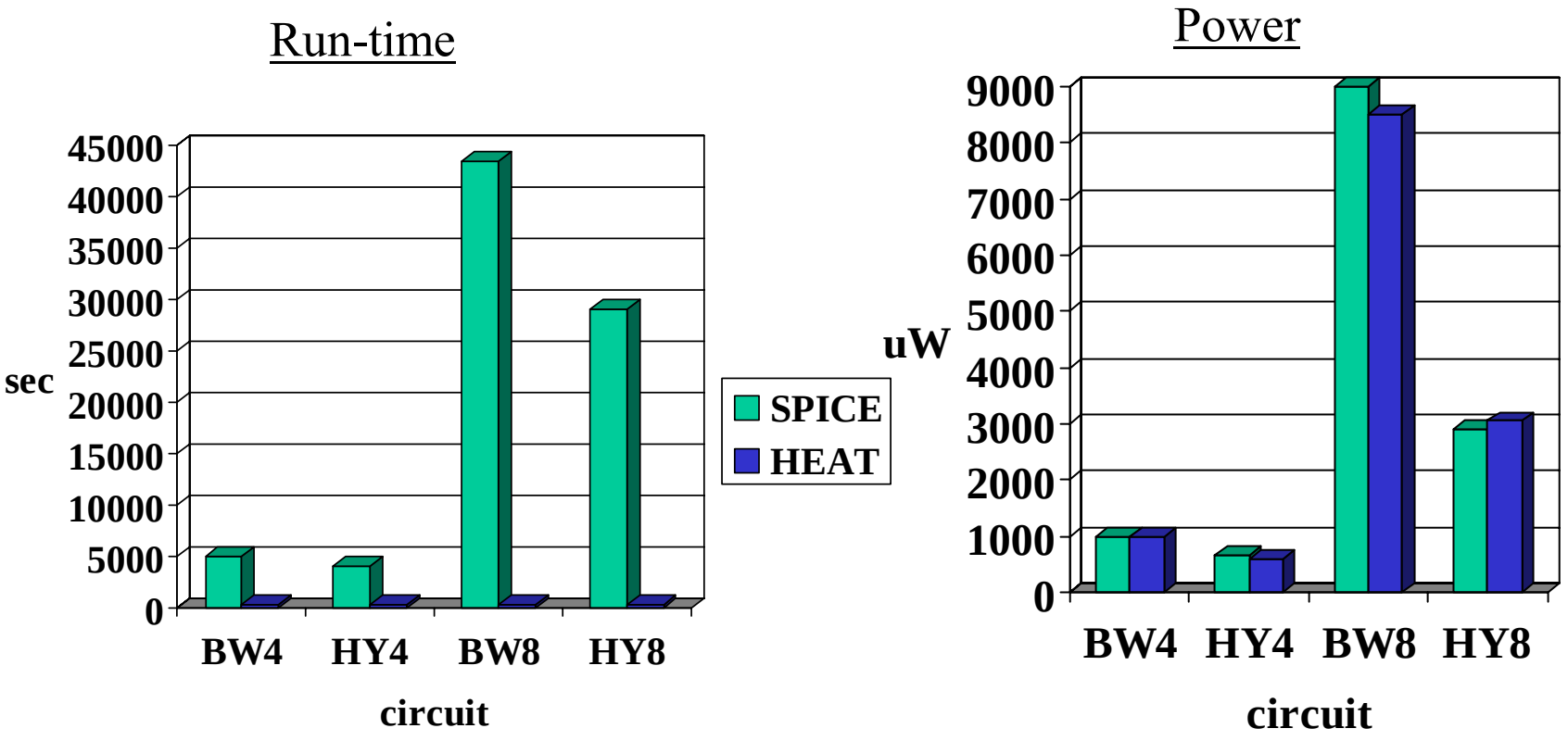
$$node_3(n+1) = (1 - x_1(n)) + (1 - x_2(n))$$

The HEAT algorithm

- Partitioning of systems unit into smaller sub-units
- State transition diagram modeling
- Edge energy computation (HSPICE)
- Computation of steady-state probabilities (MATLAB)
- Edge activity computation
- Computation of average energy



Performance Comparison



J. Satyanarayana and K.K. Parhi, "Power Estimation of Digital Datapaths using HEAT Tool", IEEE Design and Test Magazine, 17(2), pp. 101-110, April-June 2000

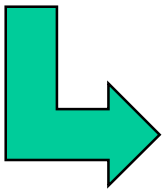
Finite field arithmetic -- Addition and Multiplication

$$A = a_{m-1}\alpha^{m-1} + \ldots + a_1\alpha + a_0$$

$$B = b_{m-1}\alpha^{m-1} + \ldots + b_1\alpha + b_0$$

$$A + B = (a_{m-1} + b_{m-1})\alpha^{m-1} + \ldots + (a_1 + b_1)\alpha + (a_0 + b_0)$$

$$A \cdot B = (a_{m-1}\alpha^{m-1} + \ldots + a_1\alpha + a_0)(b_{m-1}\alpha^{m-1} + \ldots + b_1\alpha + b_0) \bmod (p(x))$$



Polynomial addition over GF(2)

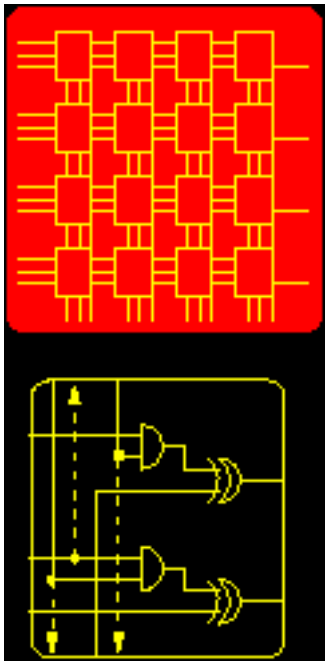
one's complement operation --> XOR gates

Polynomial multiplication and modulo operation

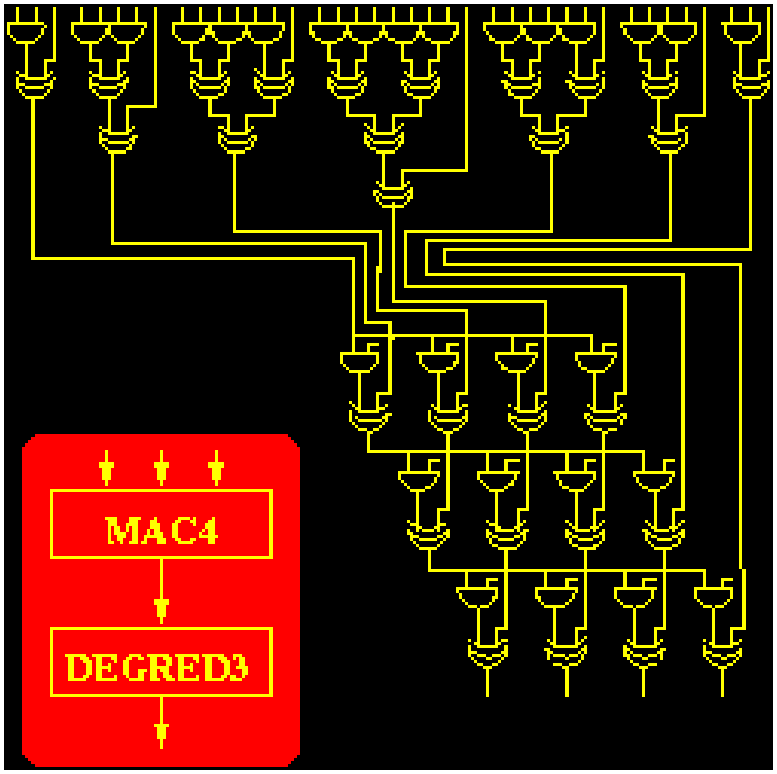
(modulo primitive polynomial p(x))

Programmable finite field multiplier

Array-type



Parallel



Digit-serial

MAC2
+
DEGRED2

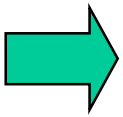
Four Instr.
- MAC2
- MAC2
- DEGRED2
- DEGRED2

Finite field arithmetic-- programmable finite field multipliers

Programmability:-primitive polynomial p(x)
-field order m

How to achieve programmability:-control circuitry
-zero, pre & post padding

Polynomial multiplication
Polynomial modulo operation



Array-type multiplication
Fully parallel multiplication
Digit-serial/parallel multiplication

L. Song and K. K. Parhi, "Low-energy digit-serial/parallel finite field multipliers", Journal of VLSI Signal Processing, 19(2), pp. 149-166, June 1998

Data-path architectures for low energy RS codecs

- Advantages of having two separate sub-arrays
  - Example: Vector-vector multiplication over GF(2^m)

$$\begin{bmatrix} A_0 & A_1 & \dots A_{n-1} \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ \dots \\ B_{n-1} \end{bmatrix} = (A_0 B_0 + \dots + A_{n-1} B_{n-1}) \bmod (p(x))$$

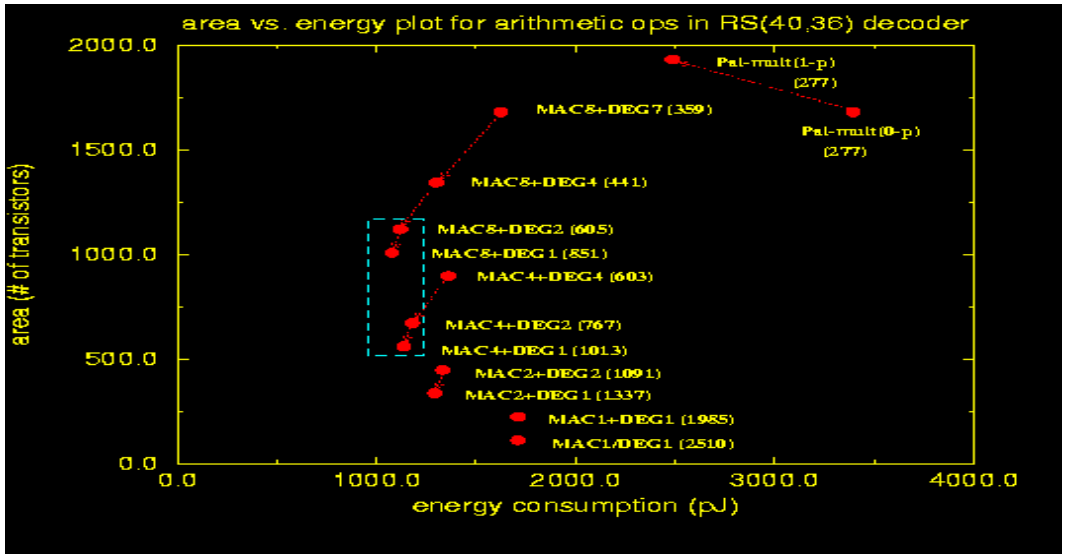
  - Assume energy(parallel multiplier)=Eng

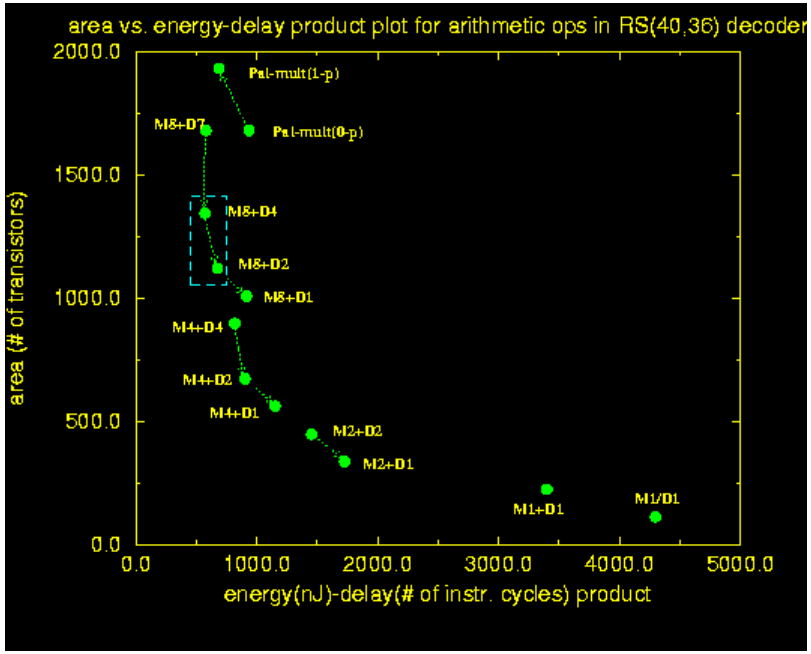
Energy(MAC8x8)=0.25 Eng
Energy(DEGRED7)=0.75 Eng

Total Energy(parallel)=Eng*n
Total Energy(MAC-D7)=0.25Eng*n+0.75Eng

$$s = \frac{Eng \cdot (n - (0.25n + 0.75))}{Eng \cdot n} \cong 75\%$$

Data-path architectures for low-power RS encoder

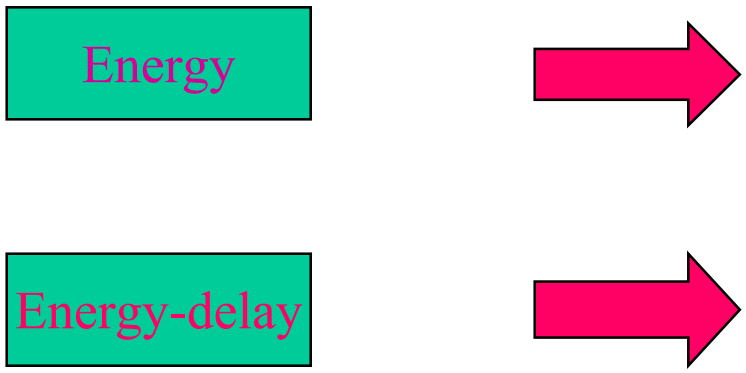




- Data-paths
  - One parallel finite field multiplier
  - Digit-serial multiplication: MACx and DEGREDy

Data-path architectures for low energy RS codecs

- Data-path:
  - one parallel finite field multiplier
  - Digit-serial multiplication: MACx and DEGREDy



MAC8 + DEGRED2
MAC8 + DEGRED1
MAC4 + DEGRED2
MAC4 + DEGRED1

MAC8 + DEGRED4
MAC8 + DEGRED2

L. Song, K.K. Parhi, I. Kuroda, T. Nishitani, "Hardware/Software Codesign of Finite Field Datapath for Low-Energy Reed-Solomon Codecs", IEEE Trans. on VLSI Systems, 8(2), pp. 160-172, Apr. 2000

Low power design challenges

- System Integration
- Application Specific architectures for Wireless/ADSL/Security
- Programmable DSPs to handle new application requirements
- Low-Power Architectures driven by Interconnect, Crosstalk in DSM technology
- How Far are we away from PDAs/Cell Phones for wireless video, internet access and e-commerce?

Chapter 18: Programmable DSPs

Keshab K. Parhi and Viktor Owall

DSP Applications

DSP applications are often real time but with a wide variety of sample rates

- **High rates**
  - **Radar**
  - **Video**
- **Medium rates**
  - **Audio**
  - **Speech**
- **Low rates**
  - **Weather**
  - **Finance**

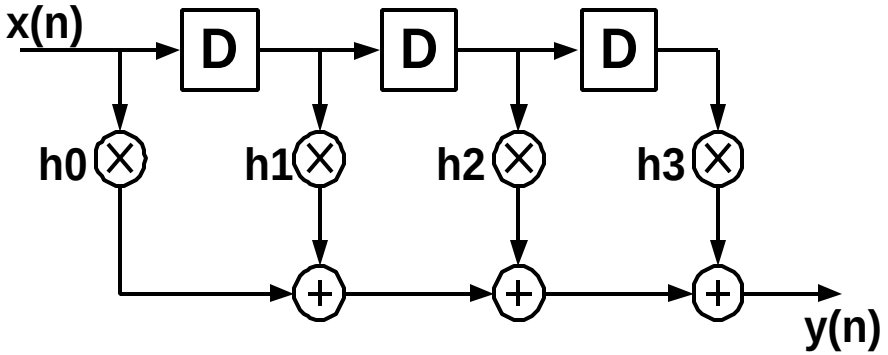
...with different demands on

- **numeric representation**
  - **float or fixed**
  - **and nmber of bits**
- **Throughput/speed**
- **Power/energy dissipation**
- **Cost**

DSP features

Fast Multiply/Accumulate (MAC)

- **FIR**
- **FFT**
- **etc.**



- **Multiple Access Memories**
- **Specialized addressing modes**
- **Specialized execution control (loops)**
- **Specialized interfaces, e.g. AD/DA**

Addressing Modes

- **Implied addressing**
  P=X*Y; operation sets location
- **Immediate data**
  AX0=1234
- **Memory direct**
  R1=Mem[101]
- **Register direct**
  sub R1, R2
- **Register indirect**
  A0=A0+ *R5
- **Register indirect with increment/decrement**
  A0=A0+ *R5++
  A0=A0+ *R5--

Standard DSP Alternatives

PCs or Workstations

- **Non-real time**
- **low requirements**

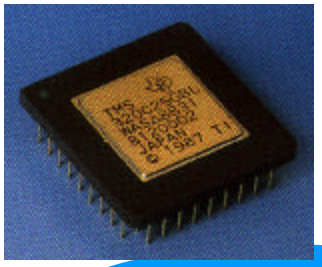
General purpose microprocessors

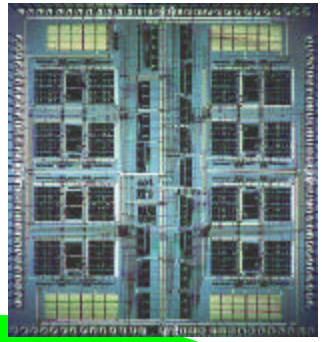
- **slower for DSP applications**
- **might be one mproc. there anyway**

Custom

- **perfomance**
- **low cost at volume**
- **High development cost**

Standard Processors vs. Special Purpose

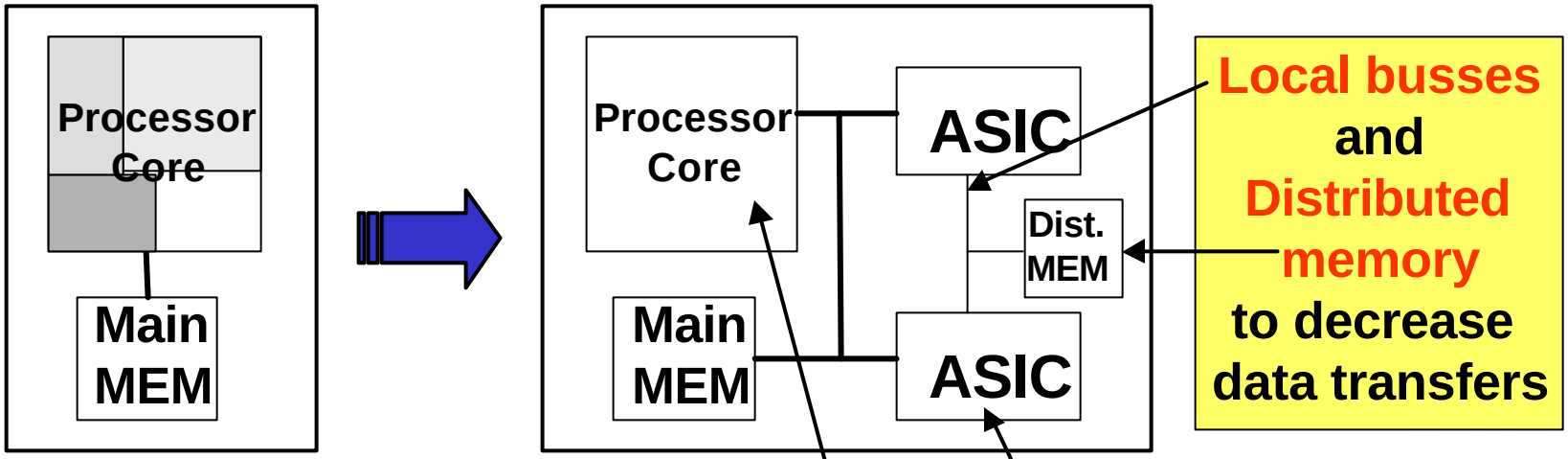




- Programmable
- Low Design cost
- Standard Interface
- Good supply of tools

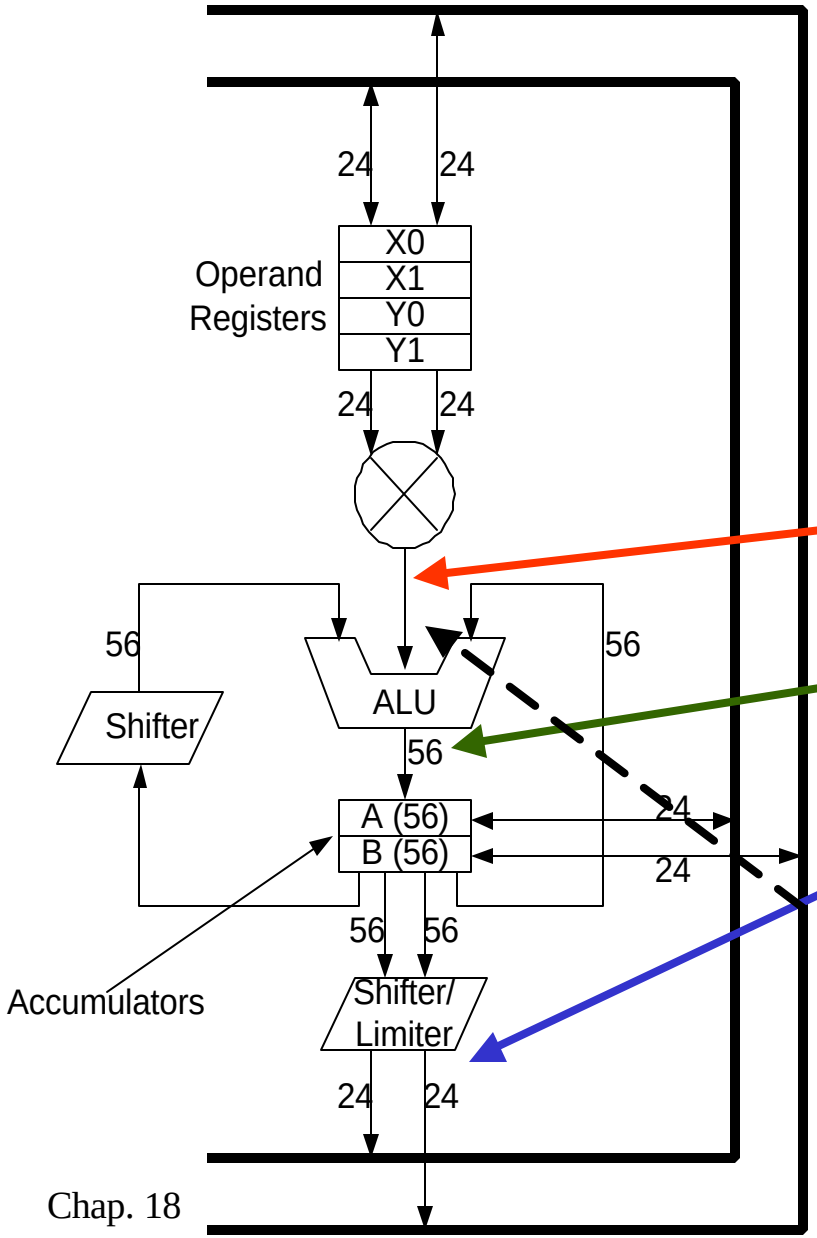
- High Calculation Capacity
- Low Power
- User defined Interface
- Variable Wordlength
- Low Price at Volume

Architectural Partitioning



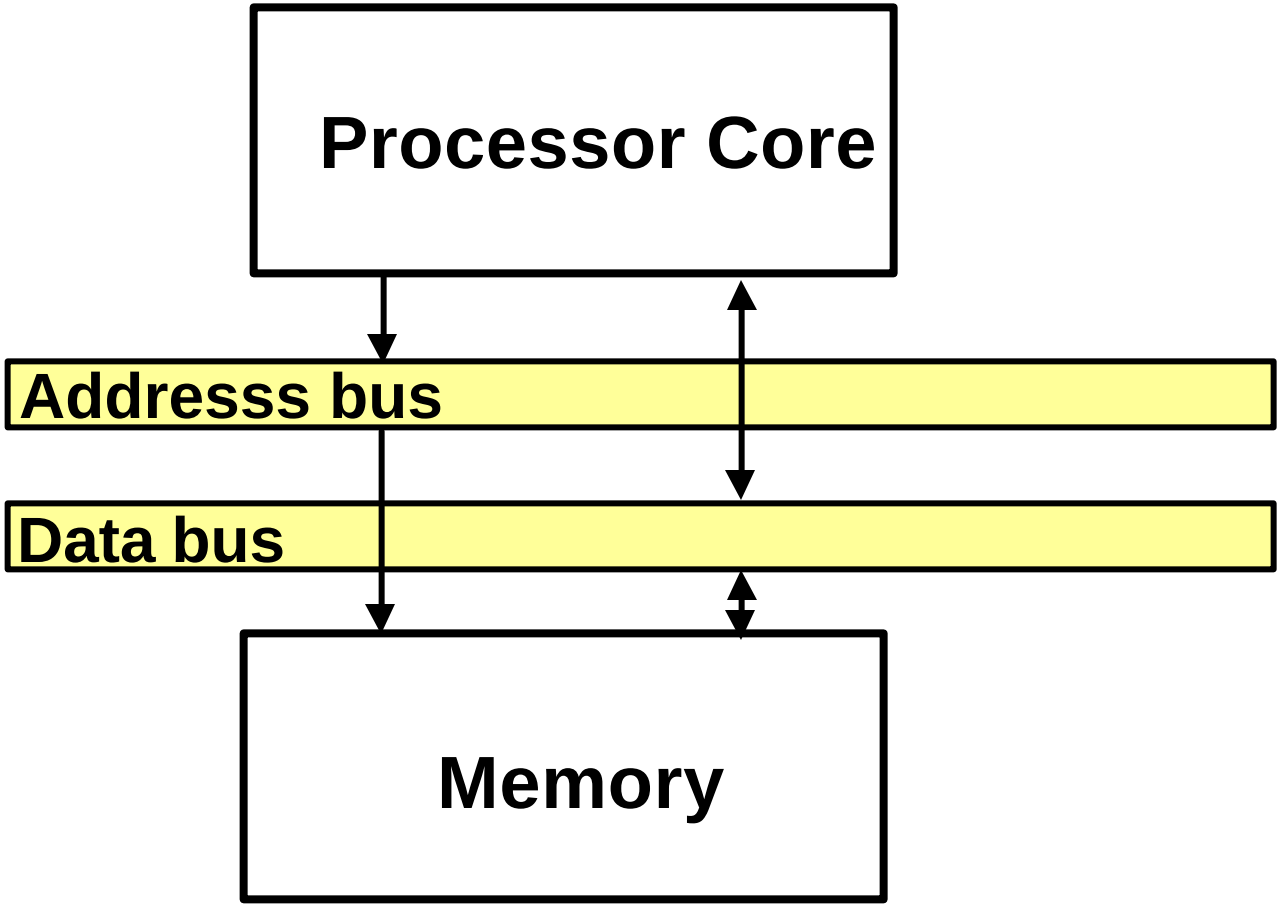
Fixed point DSP

Motorola DSP56000x

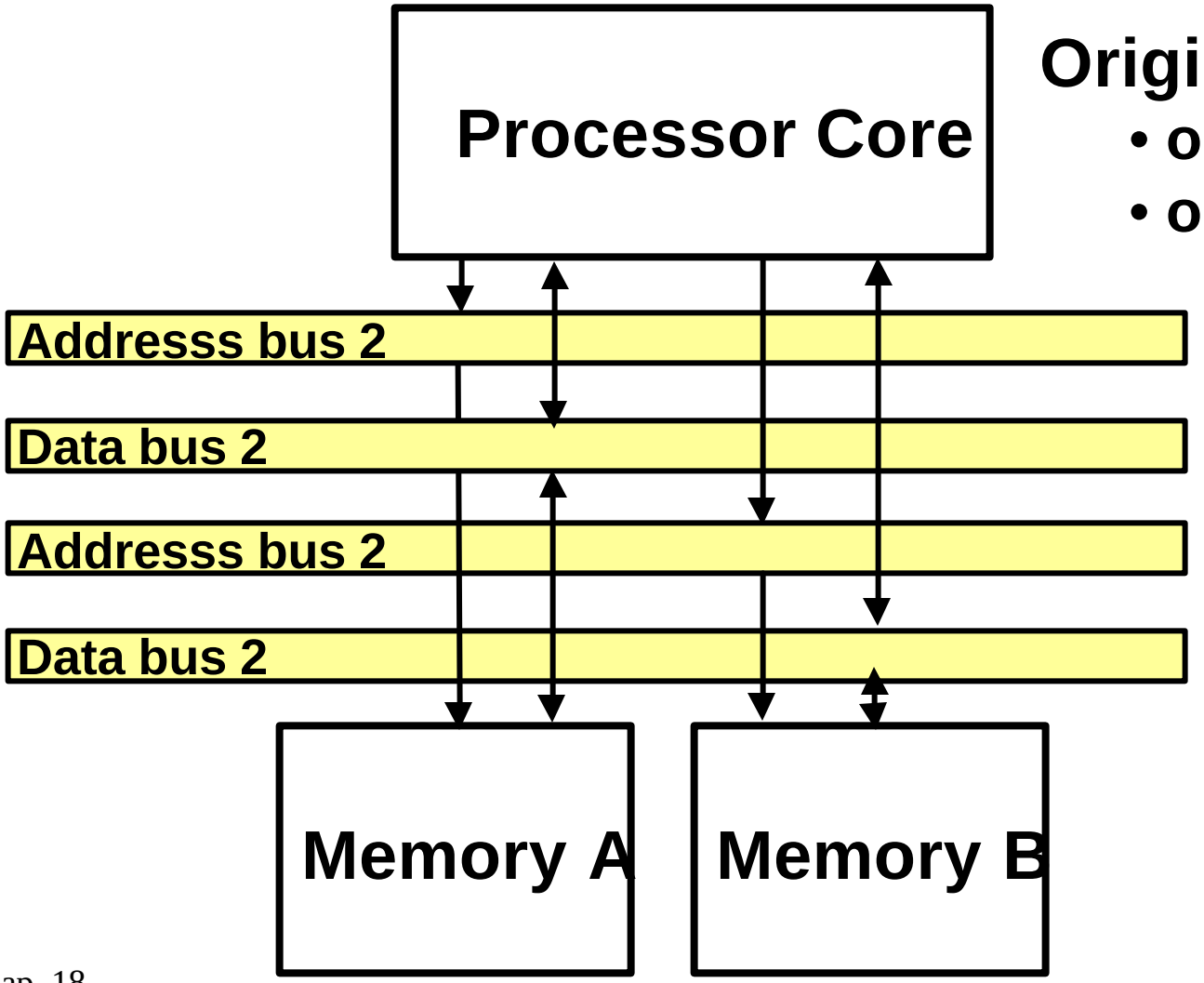


- **Usually DSP has single cycle multiplier, may be pipelined**
- **Double wordlength out**
- **+ guard bits**
- **scaling**
- **Altenative is mult with reduced wordlength output, e.g. 24**

Memory Structures, von Neuman



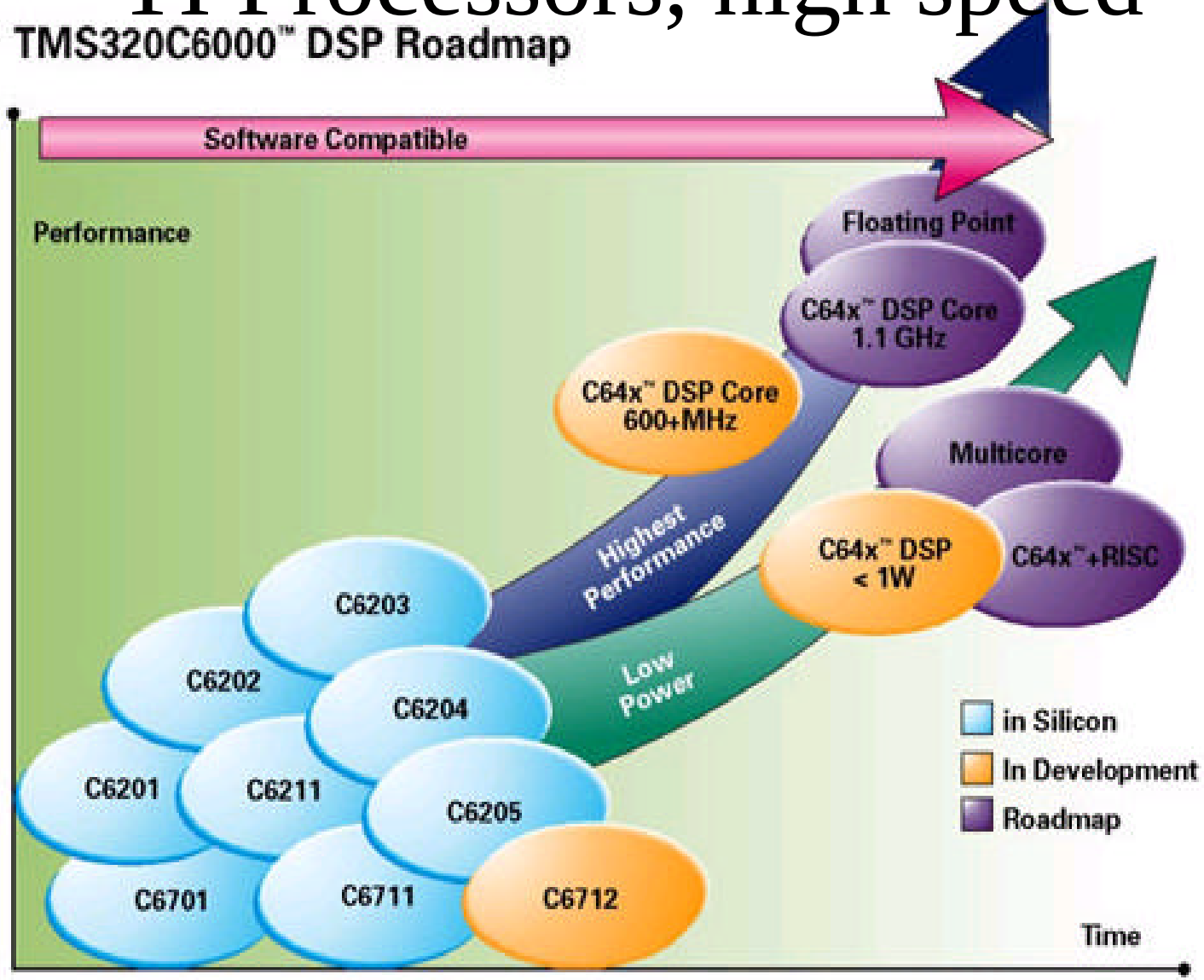
Memory Structures, Harvard



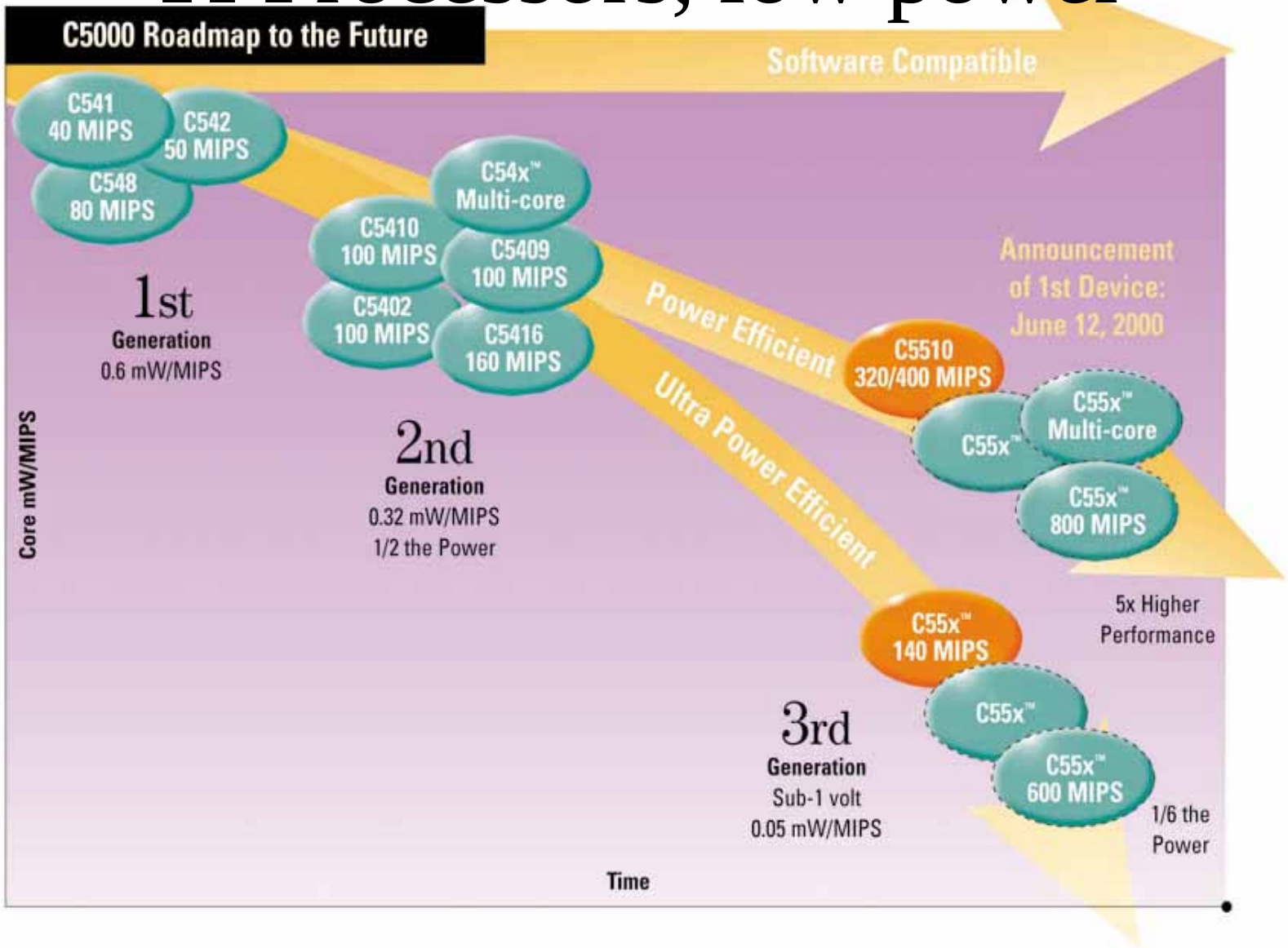
Original Harvard

- **one data**
- **one program**

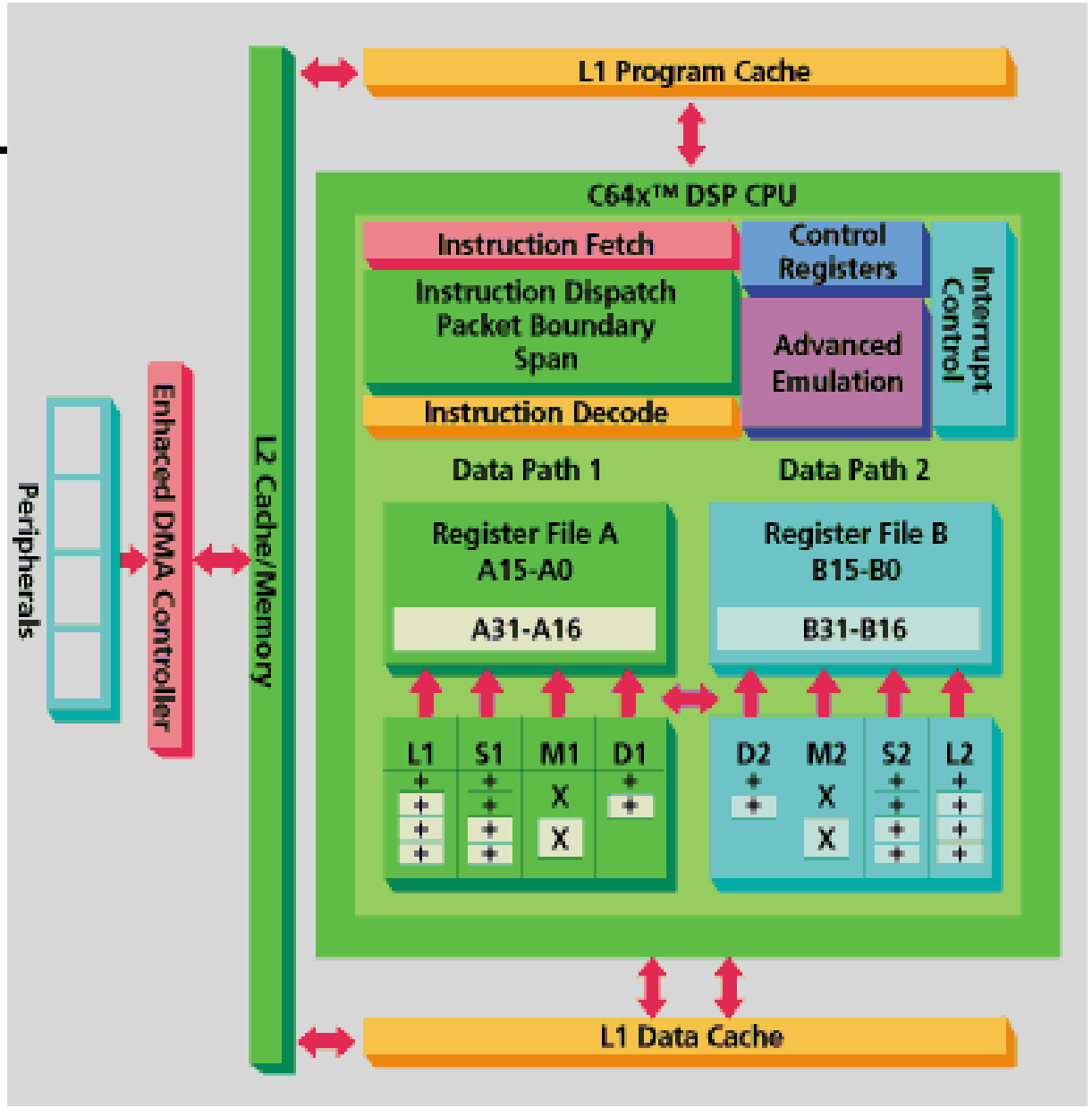
TI Processors, high speed



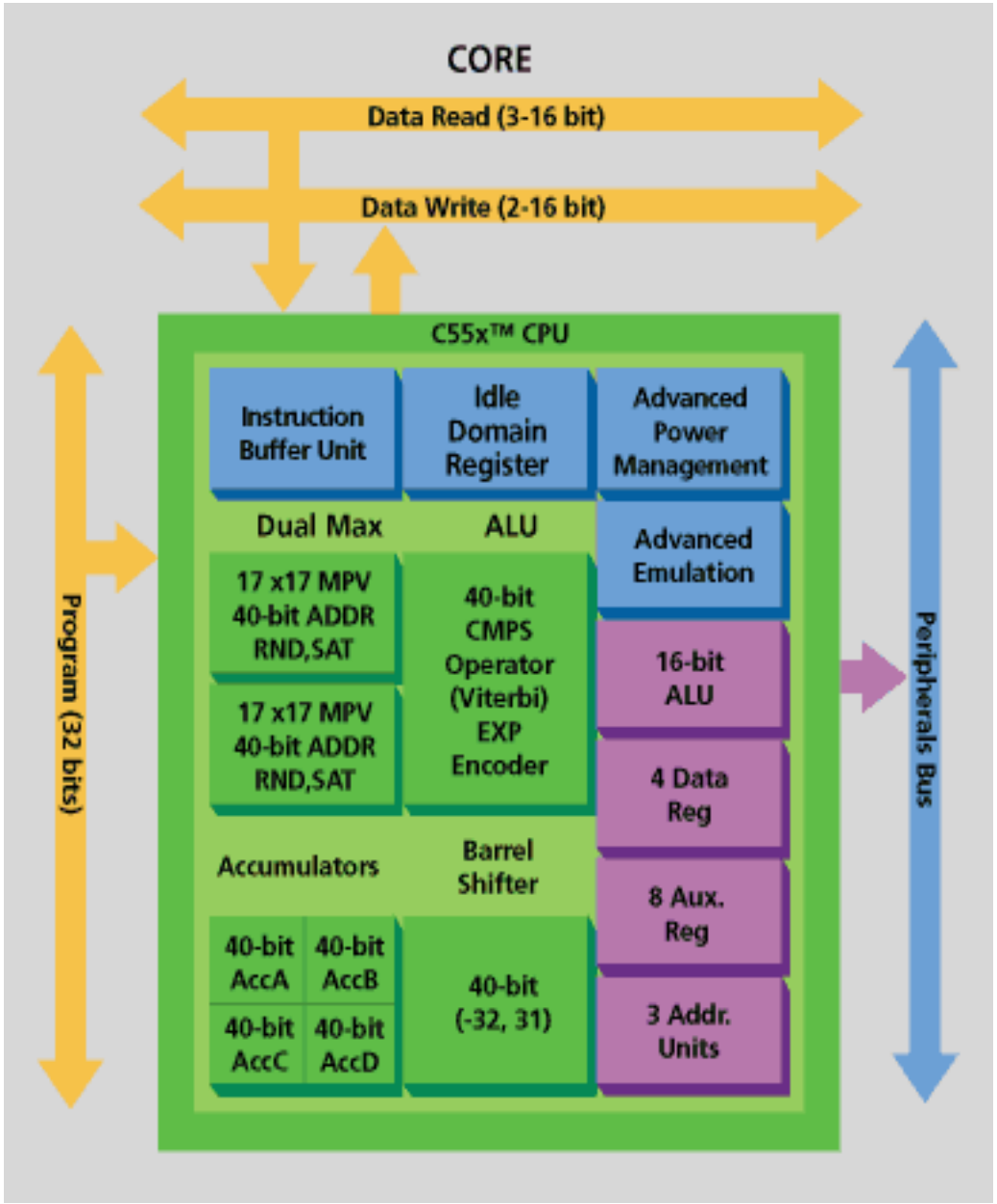
TI Processors, low power



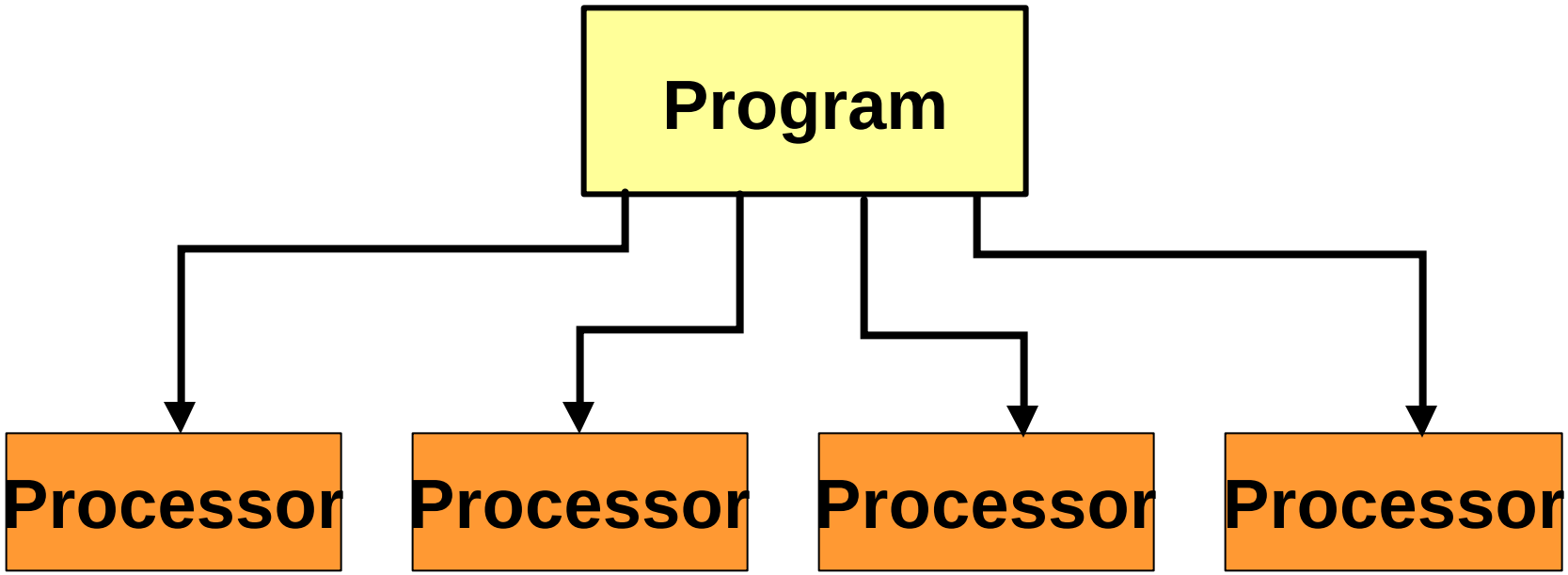
TI, C64


L1 Program Cache
C64x™ DSP CPU
Instruction Fetch
Instruction Dispatch
Packet Boundary
Span
Instruction Decode
Control
Registers
Advanced
Emulation
Interrupt
Control
Data Path 1
Data Path 2
Register File A
A15-A0
A31-A16
Register File B
B15-B0
B31-B16
L1
S1
M1
D1
D2
M2
S2
L2
X
X
X
Peripherals
Enhaced DMA Controller
L2 Cache/Memory
L1 Data Cache

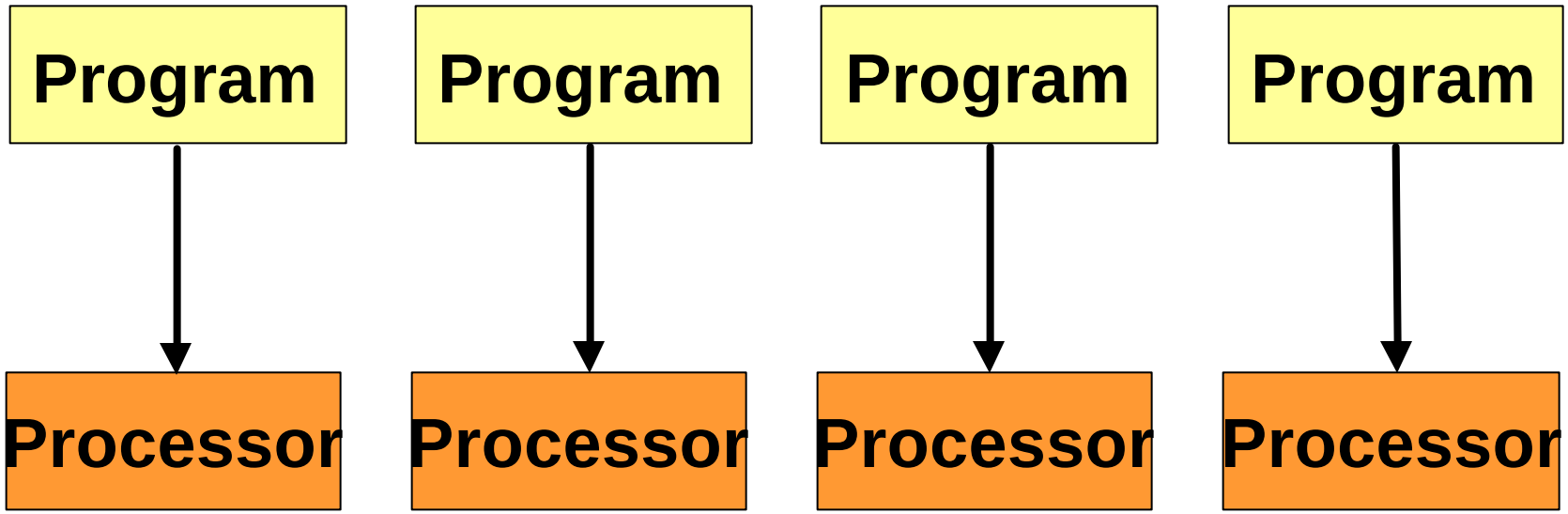
TI, C55


CORE
Data Read (3-16 bit)
Data Write (2-16 bit)
C55x™ CPU
Instruction Buffer Unit
Idle Domain Register
Advanced Power Management
Dual Max
ALU
Advanced Emulation
17 x17 MPV 40-bit ADDR RND,SAT
17 x17 MPV 40-bit ADDR RND,SAT
40-bit CMPS Operator (Viterbi) EXP Encoder
16-bit ALU
4 Data Reg
Accumulators
Barrel Shifter
8 Aux. Reg
40-bit AccA
40-bit AccB
40-bit AccC
40-bit AccD
40-bit (-32, 31)
3 Addr. Units
Program (32 bits)
Peripherals Bus

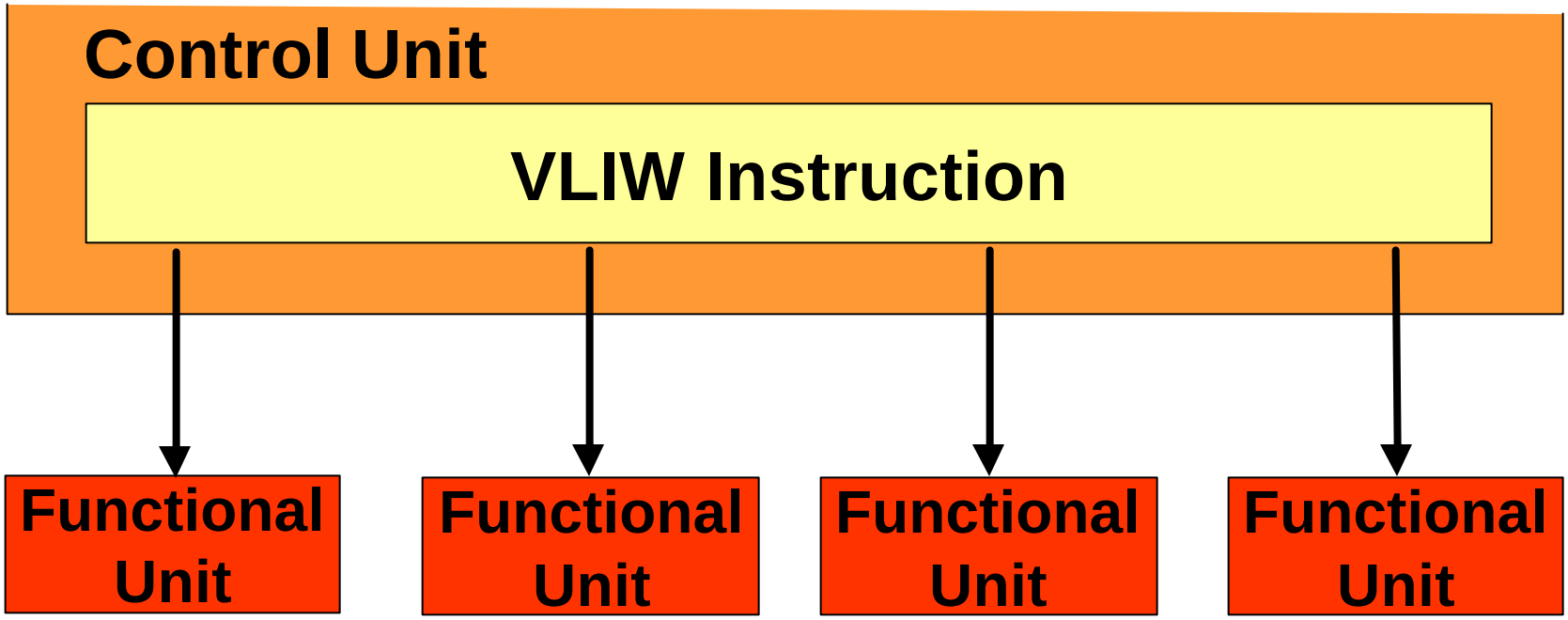
Processor Architectures
SIMD –
Single Instruction Multiple Data



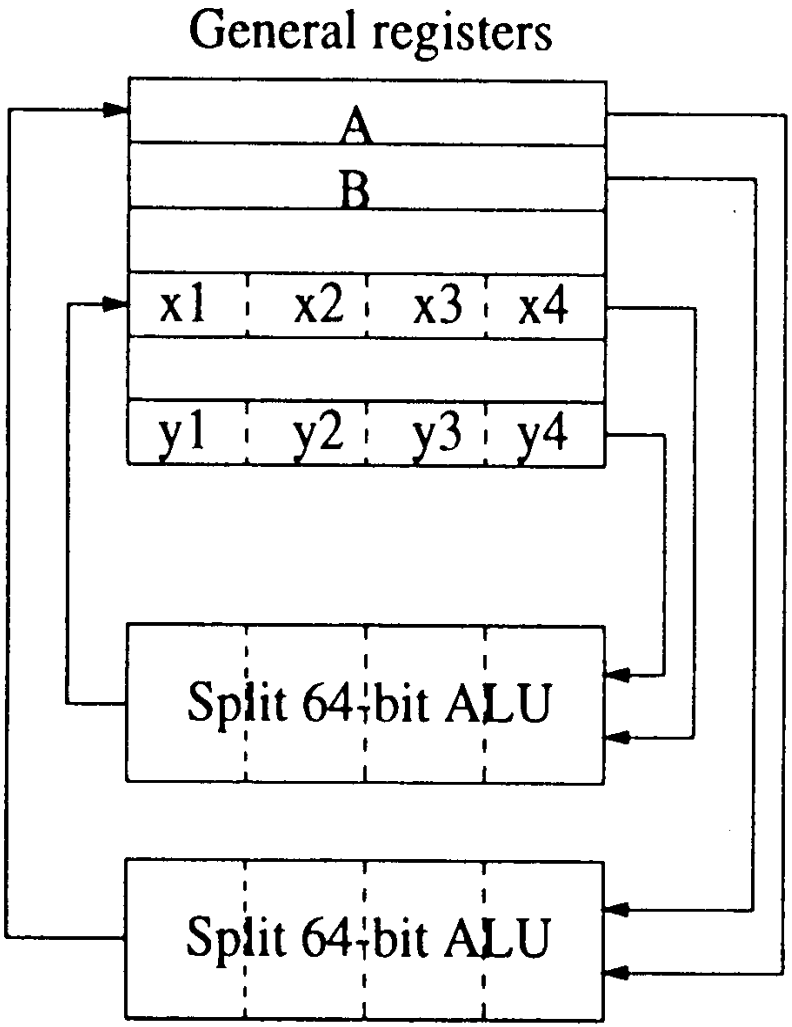
Processor Architectures
MIMD –
Multiple Instruction Multiple Data



Processor Architectures
VLIW –
Very Long Instruction Words

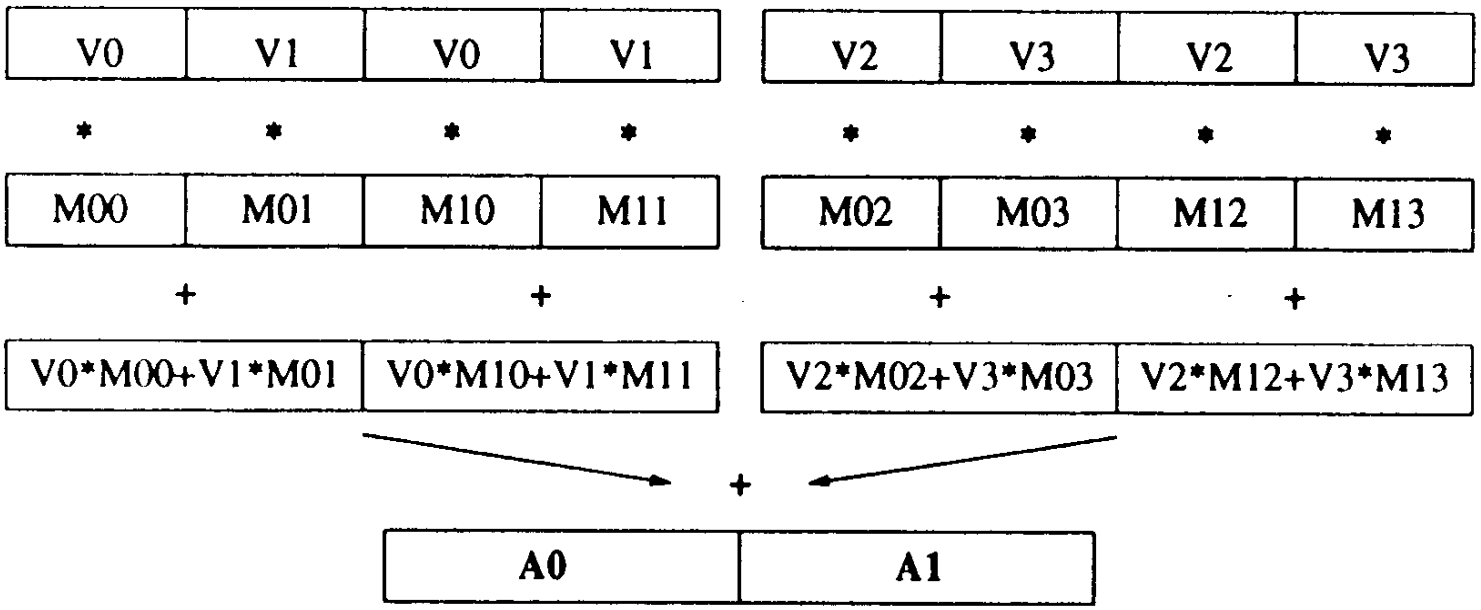


Split Processors



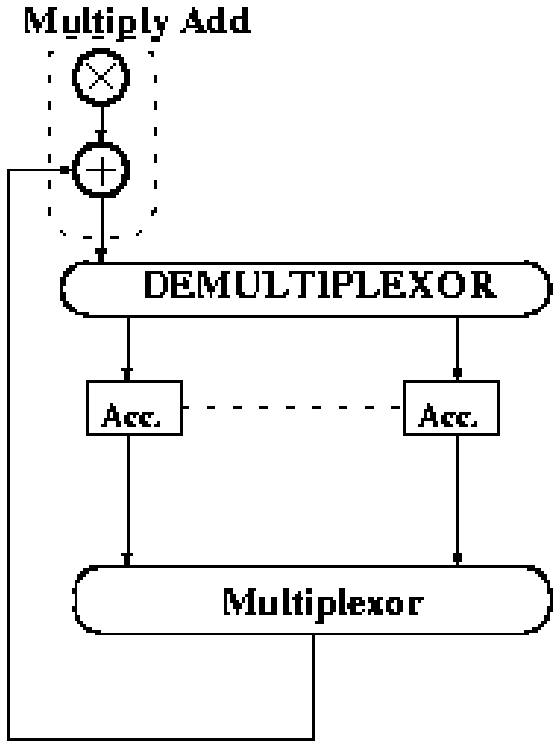
Functional units can be split into submodules, e.g. for images (8bits)
TI320C80,
1 RISC
4 x 32bit DSP which can be split into 8bit modules

Vector Processors



Low Power MMAC Multiplier Multiple Accumulator

V. Sundararajan and K.K. Parhi, "A Novel Multiply Multiple Accumulator Component for Low Power PDSP Design", Proc. of 2000 IEEE Int. Conf. on Acoustics, Speech and Signal Processing, Vol. 6, pp. 3247-3250, Istanbul, June 2000



MMAC architecture: the number of output iterations that can coexist is equal to the number of Accumulators

Low Power MMAC

Schedule 16-tap FIR 4 acc. MMAC

Cycle #	acc$_0$	acc$_1$	acc$_2$	acc$_3$
1	$a_{15} \cdot x(4j-15)$	-	-	-
2-3	$a_{14} \cdot x(4j-14)$	$a_{15} \cdot x(4j-14)$	-	-
4-6	$a_{13} \cdot x(4j-13)$	$a_{14} \cdot x(4j-13)$	$a_{15} \cdot x(4j-13)$	-
7-10	$a_{12} \cdot x(4j-12)$	$a_{13} \cdot x(4j-12)$	$a_{14} \cdot x(4j-12)$	$a_{15} \cdot x(4j-12)$
11-14	$a_{11} \cdot x(4j-11)$	$a_{12} \cdot x(4j-11)$	$a_{13} \cdot x(4j-11)$	$a_{14} \cdot x(4j-11)$
15-18	$a_{10} \cdot x(4j-10)$	$a_{11} \cdot x(4j-10)$	$a_{12} \cdot x(4j-10)$	$a_{13} \cdot x(4j-10)$
19-22	$a_9 \cdot x(4j-9)$	$a_{10} \cdot x(4j-9)$	$a_{11} \cdot x(4j-9)$	$a_{12} \cdot x(4j-9)$
23-26	$a_8 \cdot x(4j-8)$	$a_9 \cdot x(4j-8)$	$a_{10} \cdot x(4j-8)$	$a_{11} \cdot x(4j-8)$
27-30	$a_7 \cdot x(4j-7)$	$a_8 \cdot x(4j-7)$	$a_9 \cdot x(4j-7)$	$a_{10} \cdot x(4j-7)$
31-34	$a_6 \cdot x(4j-6)$	$a_7 \cdot x(4j-6)$	$a_8 \cdot x(4j-6)$	$a_9 \cdot x(4j-6)$
35-38	$a_5 \cdot x(4j-5)$	$a_6 \cdot x(4j-5)$	$a_7 \cdot x(4j-5)$	$a_8 \cdot x(4j-5)$
39-42	$a_4 \cdot x(4j-4)$	$a_5 \cdot x(4j-4)$	$a_6 \cdot x(4j-4)$	$a_7 \cdot x(4j-4)$
43-46	$a_3 \cdot x(4j-3)$	$a_4 \cdot x(4j-3)$	$a_5 \cdot x(4j-3)$	$a_6 \cdot x(4j-3)$
47-50	$a_2 \cdot x(4j-2)$	$a_3 \cdot x(4j-2)$	$a_4 \cdot x(4j-2)$	$a_5 \cdot x(4j-2)$
51-54	$a_1 \cdot x(4j-1)$	$a_2 \cdot x(4j-1)$	$a_3 \cdot x(4j-1)$	$a_4 \cdot x(4j-1)$
55-58	$a_0 \cdot x(4j)$	$a_1 \cdot x(4j)$	$a_2 \cdot x(4j)$	$a_3 \cdot x(4j)$
59-61	—	$a_0 \cdot x(4j+1)$	$a_1 \cdot x(4j+1)$	$a_2 \cdot x(4j+1)$
62-63	-	-	$a_0 \cdot x(4j+2)$	$a_1 \cdot x(4j+2)$
64	-	-	-	$a_0 \cdot x(4j+3)$