ECE 598NSG/498NSU
Deep Learning in Hardware
Fall 2020

Mapping Algorithms to Architectures Systematically – Algorithm Transforms

Naresh Shanbhag

Department of Electrical and Computer Engineering

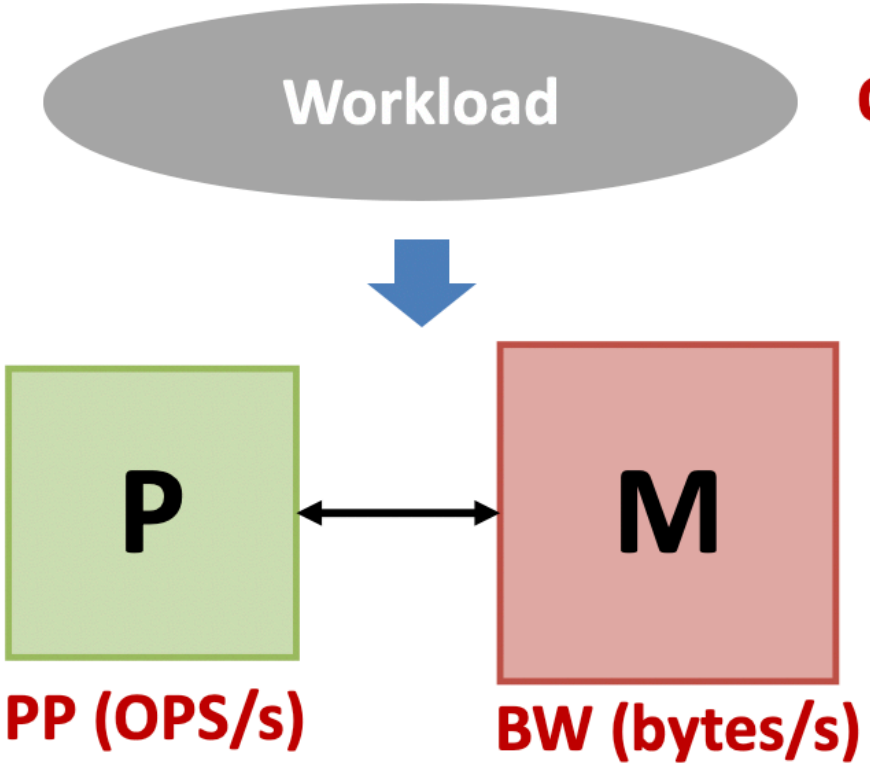
University of Illinois at Urbana-Champaign

http://shanbhag.ece.uiuc.edu

Course Progress

- basic DNN function, terminology, complexity (computational and storage costs)
  - Simple inference engine – linear regressor/predictor -> optimum weight vector can be obtained analytically
  - LMS algorithm (learning rule) -> update the weights to approach the optimum solution -> MSE
  - DNN – back-prop algorithm (learning rule); cross entropy loss
  - LMS and back-prop -> fundamental SGD
- DNN inference and training in fixed-point – first step towards complexity reduction
  - Quantization leads loss in accuracy -> minimize the loss in accuracy
  - Precision assignment for forward path and for the training loop/backward path -> post training quantization (with guarantees)
- Low-complexity DNNs – designing networks from scratch
  - Depth-wise separable convolutions; delaying the use of averaging layers
  - Trained quantization – in-training quantization (no guarantees)
- Now – we have a compact, fixed-point, network to implement -> architectures

A Generic DNN Architecture



workload dependent

$$\text{Achievable Performance } (AP) = \min(\text{PP}, \text{OI}\times\text{BW})$$

processor dependent

memory interface dependent

- Enhancing OI: data reuse & HW mapping
- Enhancing PP: systolic architectures & **algorithm transforms**
- Enhancing BW: new memory technologies + in-memory computing

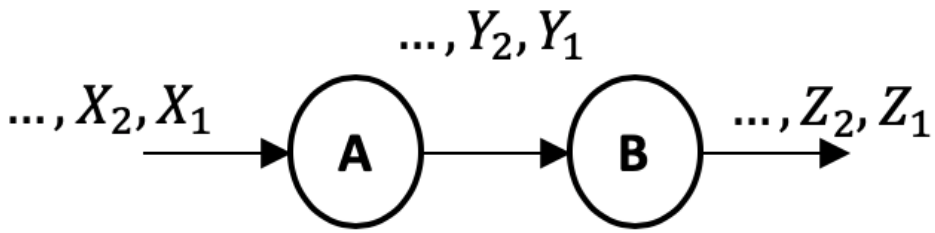
Algorithm Transforms

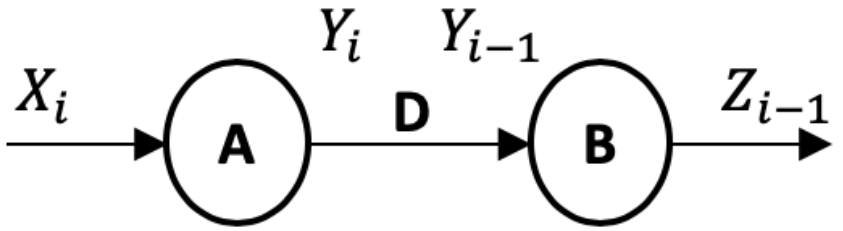
- the goal of this lecture is to study algorithm-to-architecture mapping techniques:
  - data flow-graphs to describe algorithms
  - retiming
  - pipelining and parallelization
  - folding and unfolding

- These operate on Data Flow Graphs (DFGs). Part of many automated CAD tool-flows for DSP architecture synthesis, e.g., Hyper-LP [anantha-CAD95]. Useful for manually generated architectures as well.

Data-flow Graphs (DFGs)

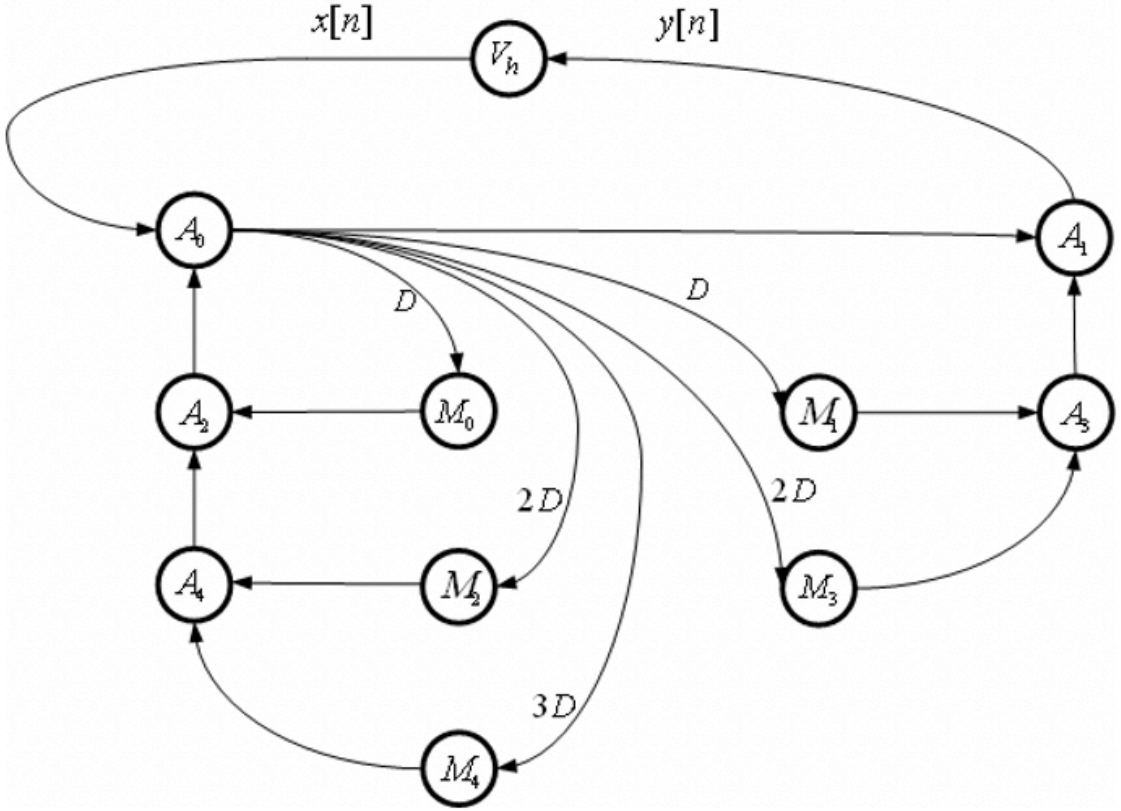
Modeling Computation





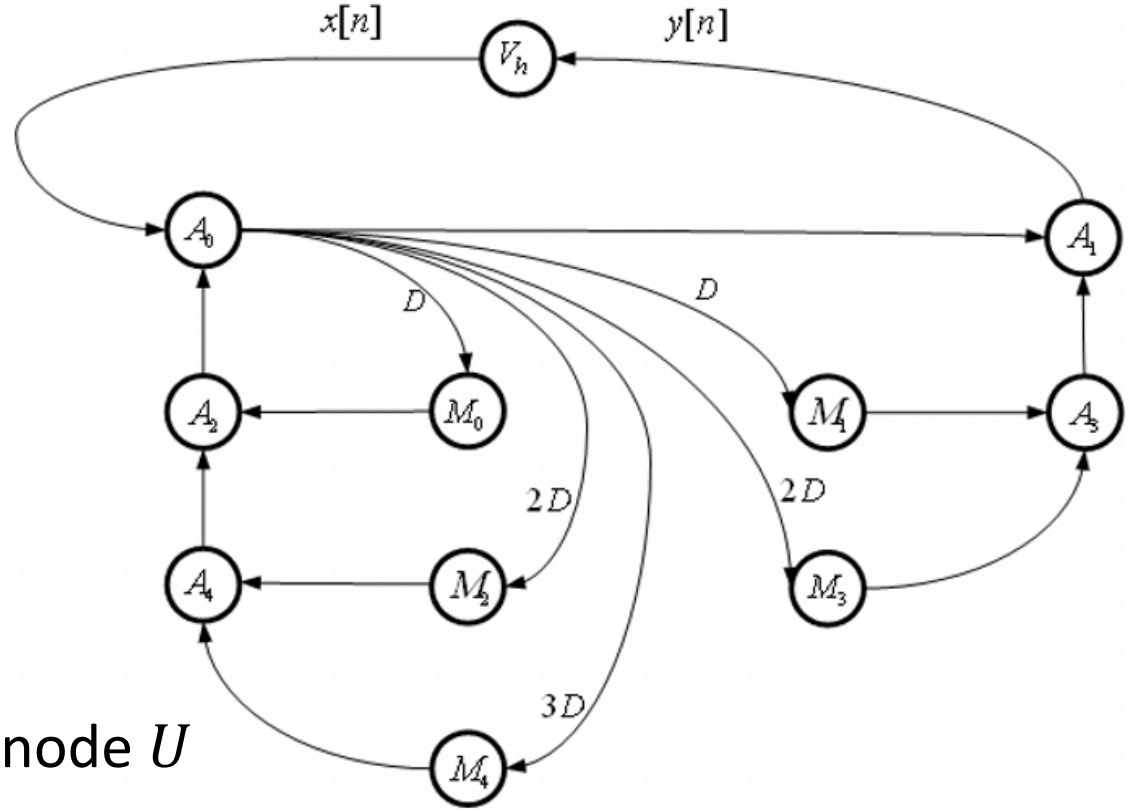
- learning → repeated computations on different data
- input data X_i (sample or vector): i is the sample index
  - For a time series: $i = n$ (time index)
- node: a memoryless computation or mapping (e.g., MVM in DNNs)
- arc/edge: communication between nodes
- D: storage (register) representing 1 sample/input delay; also called arc weight

Data Flow Graphs (DFG)



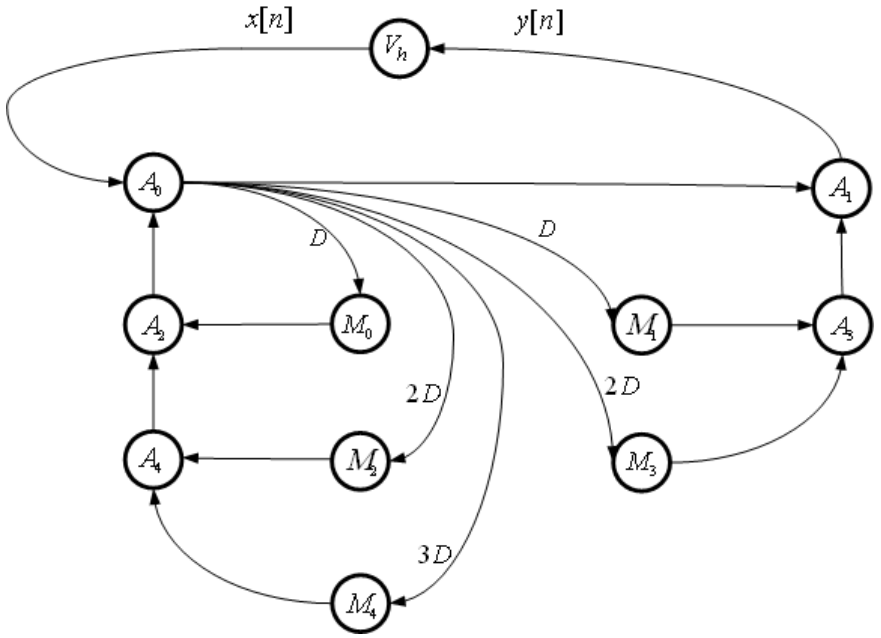
- a DFG is composed of a:
  - set of nodes S_U : $S_U = \{ A_{0\text{-}4} , M_{0\text{-}4} \}$
  - set of arcs/edges S_e : $A_0 \rightarrow M_0, M_0 \rightarrow A_2 \in S_e$

Data Flow Graphs (DFG)

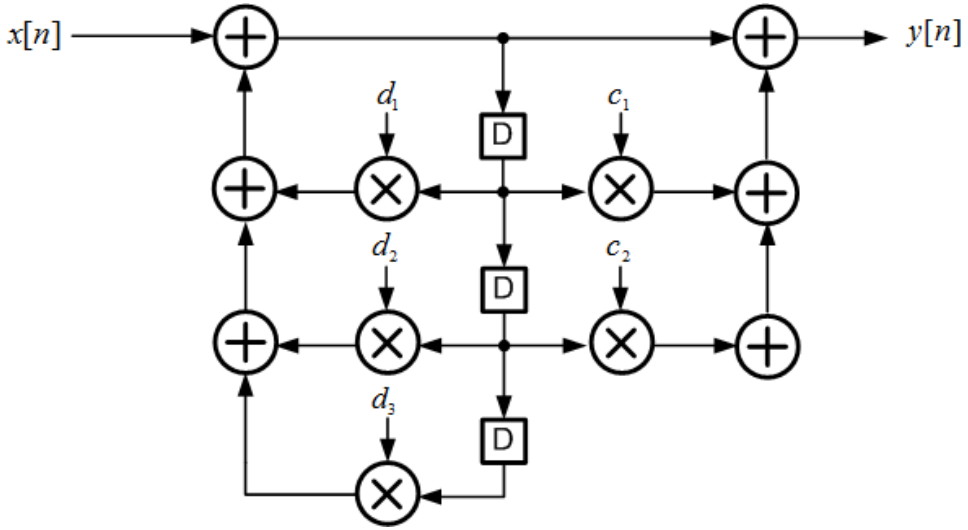


- $d(U)$: deterministic worst case delay of node U
- $w(e)$: arc weight
- $w(e) = 0 \rightarrow$ signifies intra-iteration precedence
- $w(e) > 0 \rightarrow$ signifies inter-iteration precedence

Mapping DFG to Architecture



DFG



direct-mapped architecture

- each node mapped to a dedicated hardware unit
- each arc mapped to a dedicated interconnect
- referred to as a direct-mapped architecture

Iteration Period (IP) vs. Clock Period

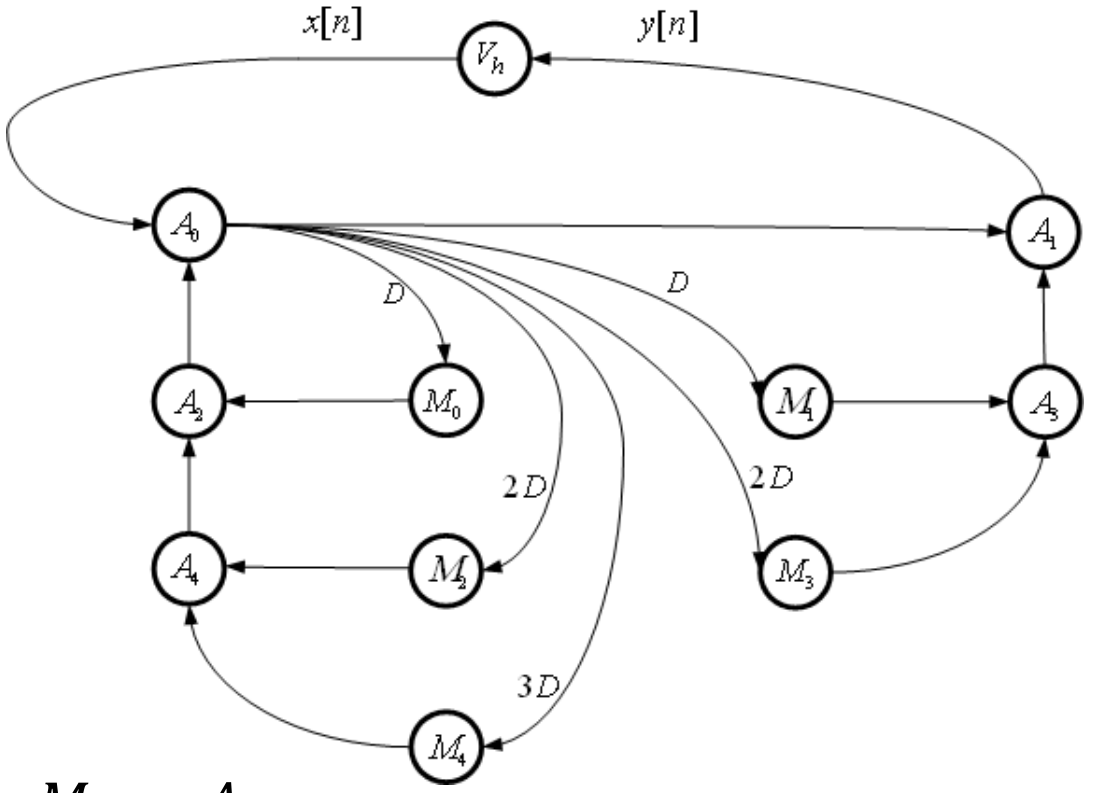
- IP: smallest sample period (of the DFG) required to complete execution of one iteration or one input sample
- T_{CLK}: smallest clock period (of the direct-mapped architecture) required to execute all computations correctly

- usually $IP = T_{CLK}$

- a DFG can be transformed into another functionally equivalent DFG via algorithm transforms → modifies the IP and the T_{CLK}

- IPB: iteration period bound → lower bound on the IP

DFG Properties

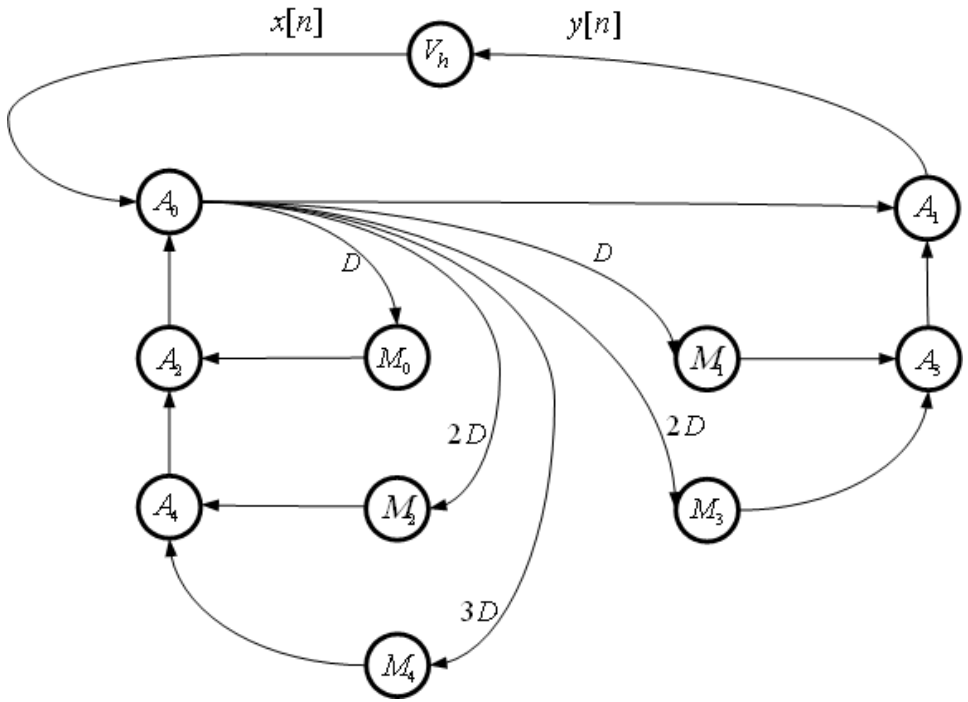


- path p: $U \rightarrow W$
  - Sequence of nodes connected by arcs: $A_0 \rightarrow M_0 \rightarrow A_2$
  - U=source node, W=destination node, $w(p)$=path weight, $d(p)$=path delay

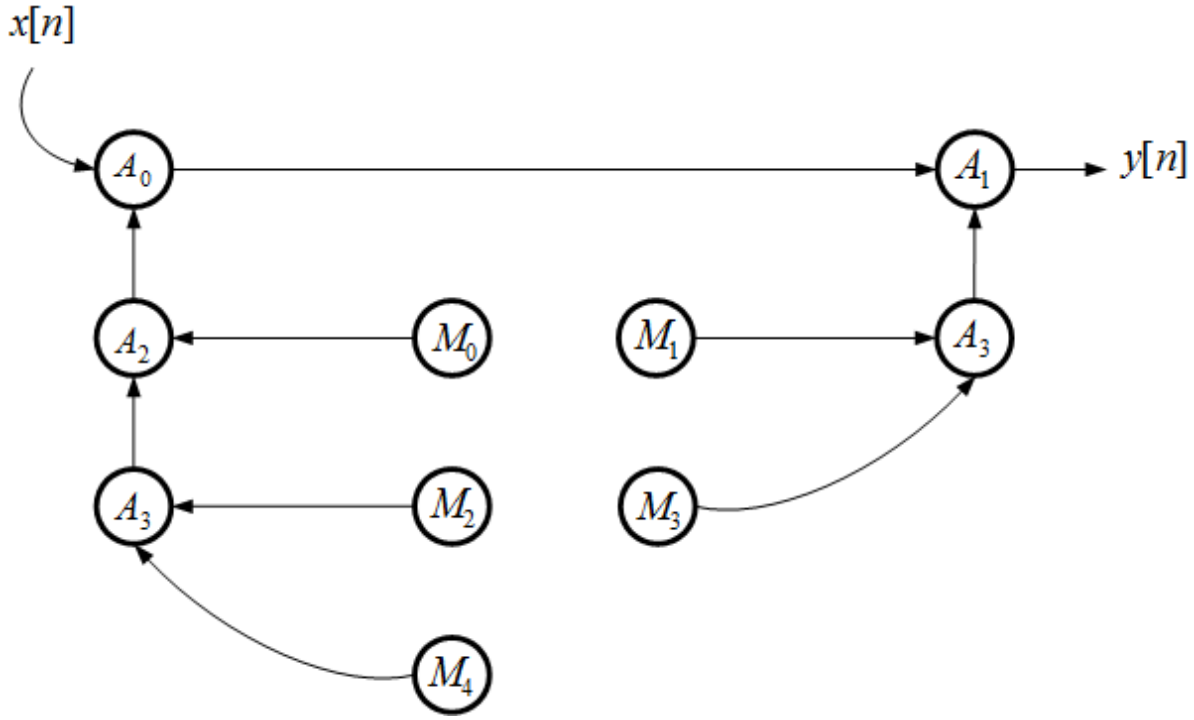
$$w(p) = \sum_{e_i \in p} w(e_i) \qquad d(p) = \sum_{U_i \in p} d(U_i)$$

- loop
  - path with identical source and destination nodes
  - E.g., $A_0 \rightarrow M_0 \rightarrow A_2 \rightarrow A_0$

Acyclic DFG (aDFG)



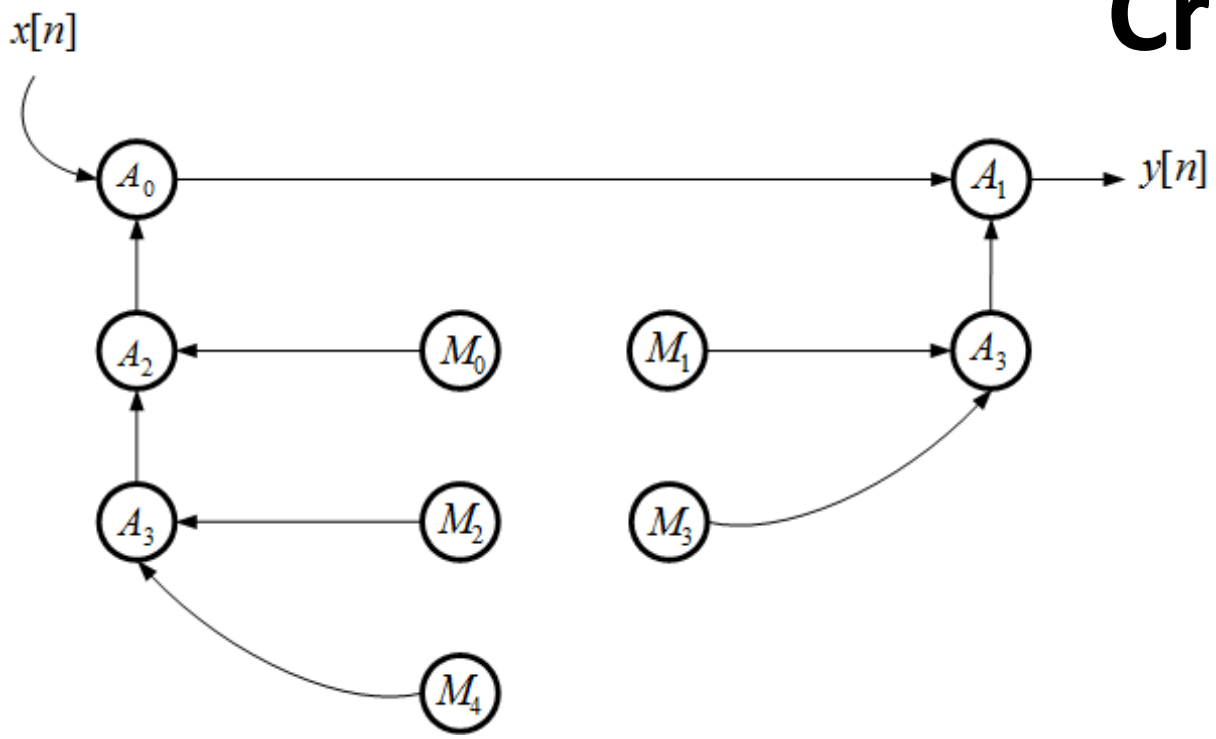
DFG



aDFG

- aDFG: DFG with non-zero weighted arcs removed

Critical Path in a DFG

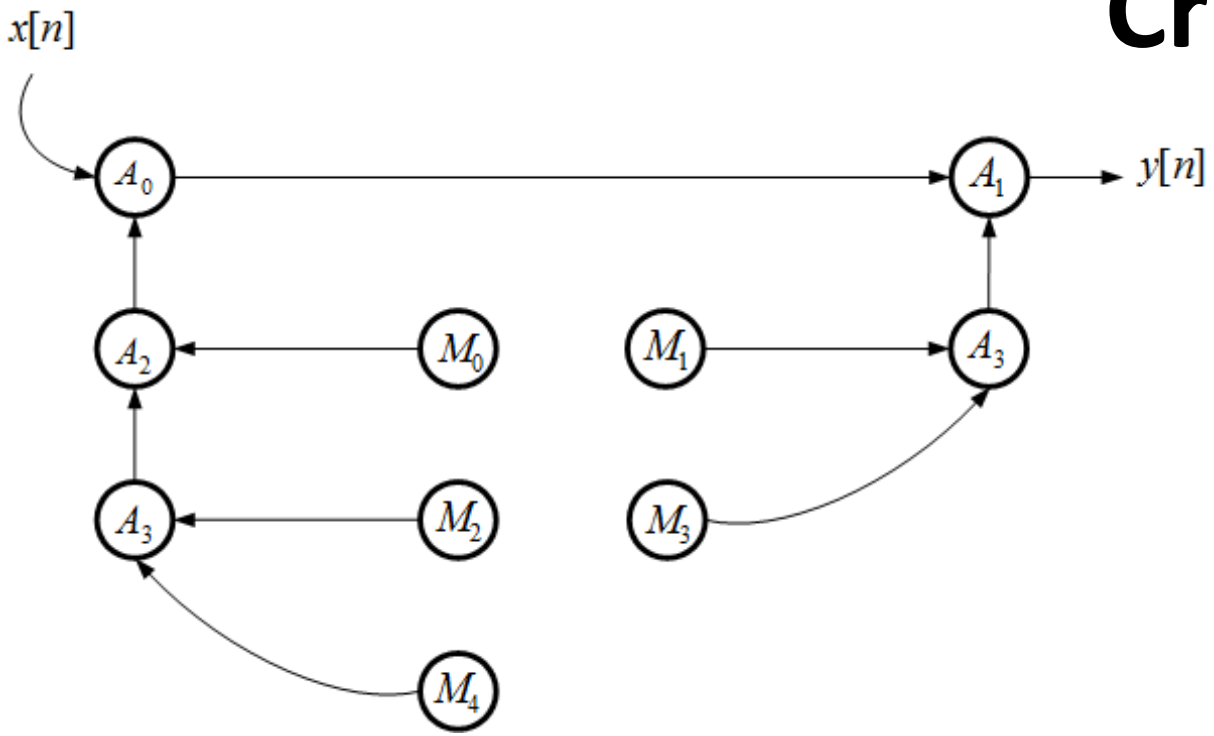


aDFG

- path with the maximum delay in the corresponding aDFG
  - *d(M)*=2 a.u. (arbitrary units), *d(A)*=1 a.u

  CP: $M_4 \rightarrow A_3 \rightarrow A_2 \rightarrow A_0 \rightarrow A_1$; $d(cp)=T_{CP} = d(M)+4d(A) = 6$
- $IP = d(cp) = T_{CP}$: iteration period equals the critical path delay

Critical Path in a DFG



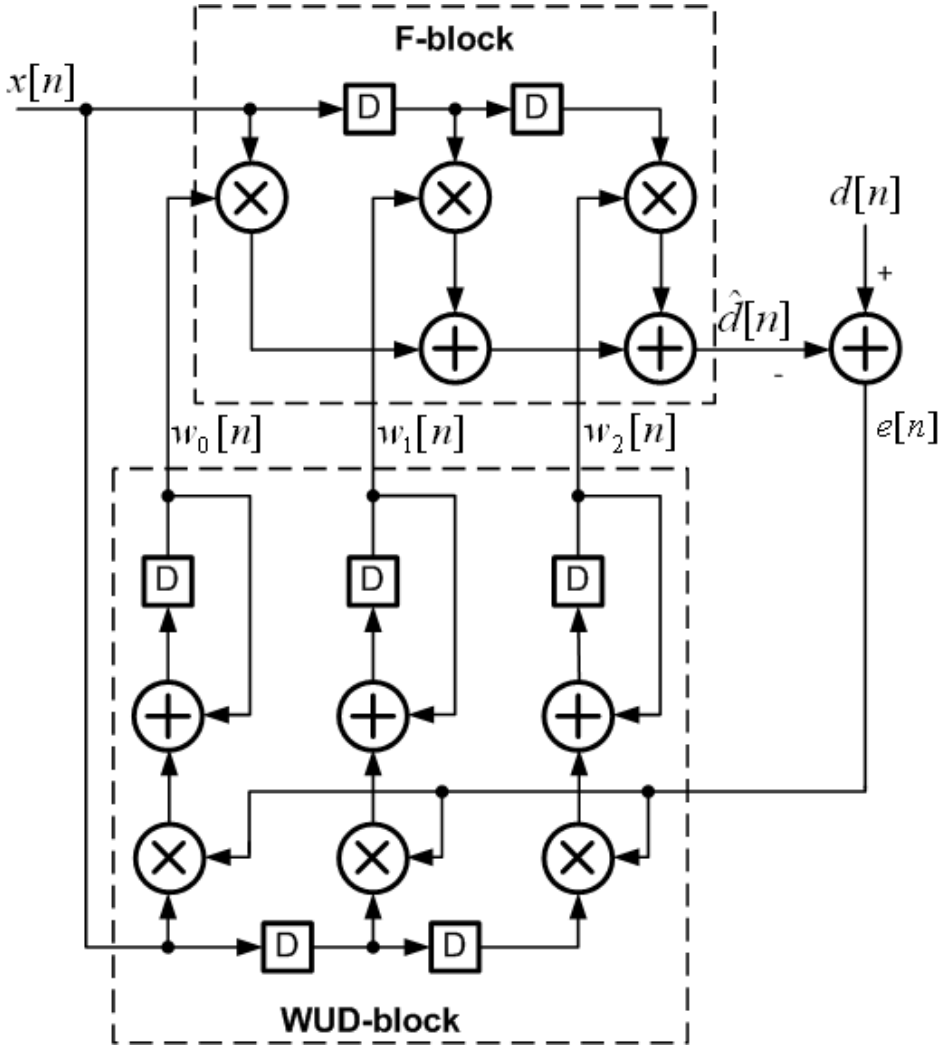
aDFG

$T_{CP} = 6$

- $IP = d(cp) = T_{CP}$: iteration period equals the critical path delay

- $\frac{1}{T_{CP}}$ = peak performance (PP)

Example: 3-Tap LMS Adaptive Filter



D: one sample delay (register)

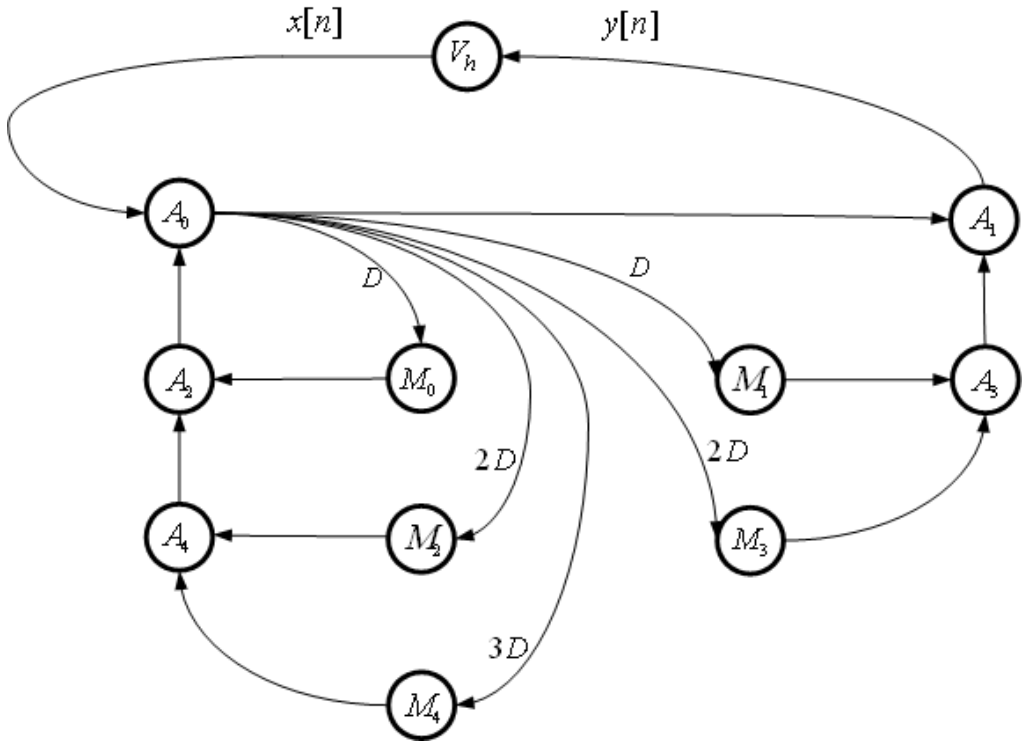
- critical path delay:

$$T_{cp} = 2T_m + 4T_A$$

- PP $= 1/T_{cp}$
- T_{cp} can be reduced via algorithm transforms
- latency = $2T_m + 4T_A = T_{cp}$

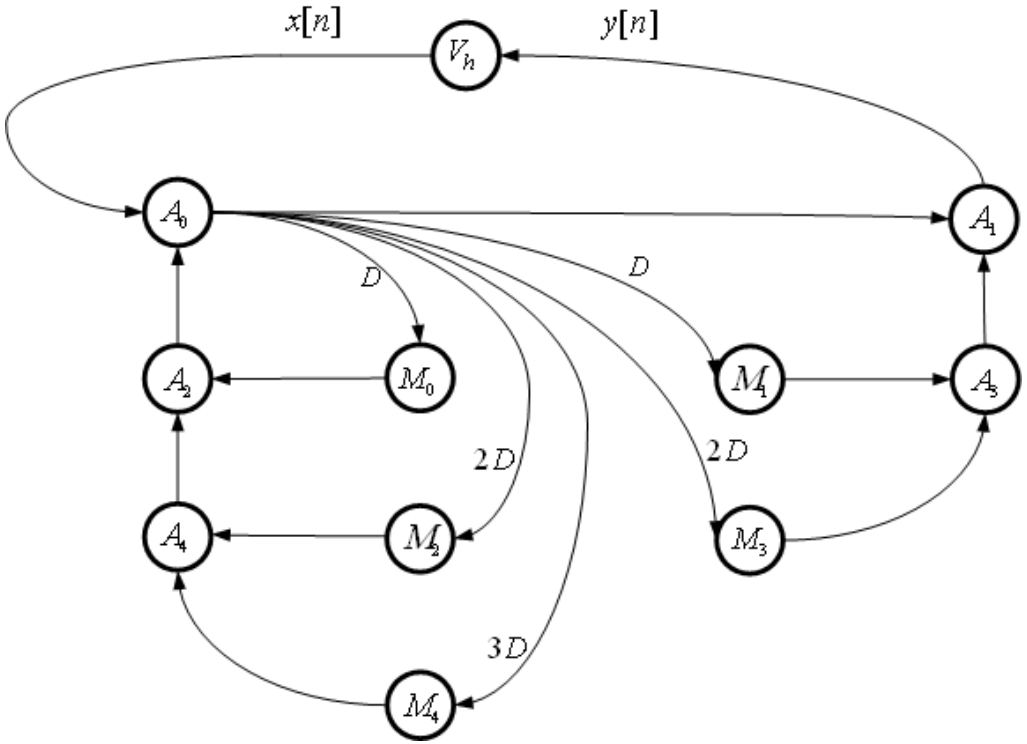
Iteration Period Bound

$$IPB = \max \frac{\sum_U d(U)}{\sum_e w(e)}$$



- IPB: lower-bound on IP over all equivalent DFGs
- $IP = IPB$ can be achieved in:
  - many-core implementations via *unfolding*
  - dedicated ASIC implementations via *retiming*

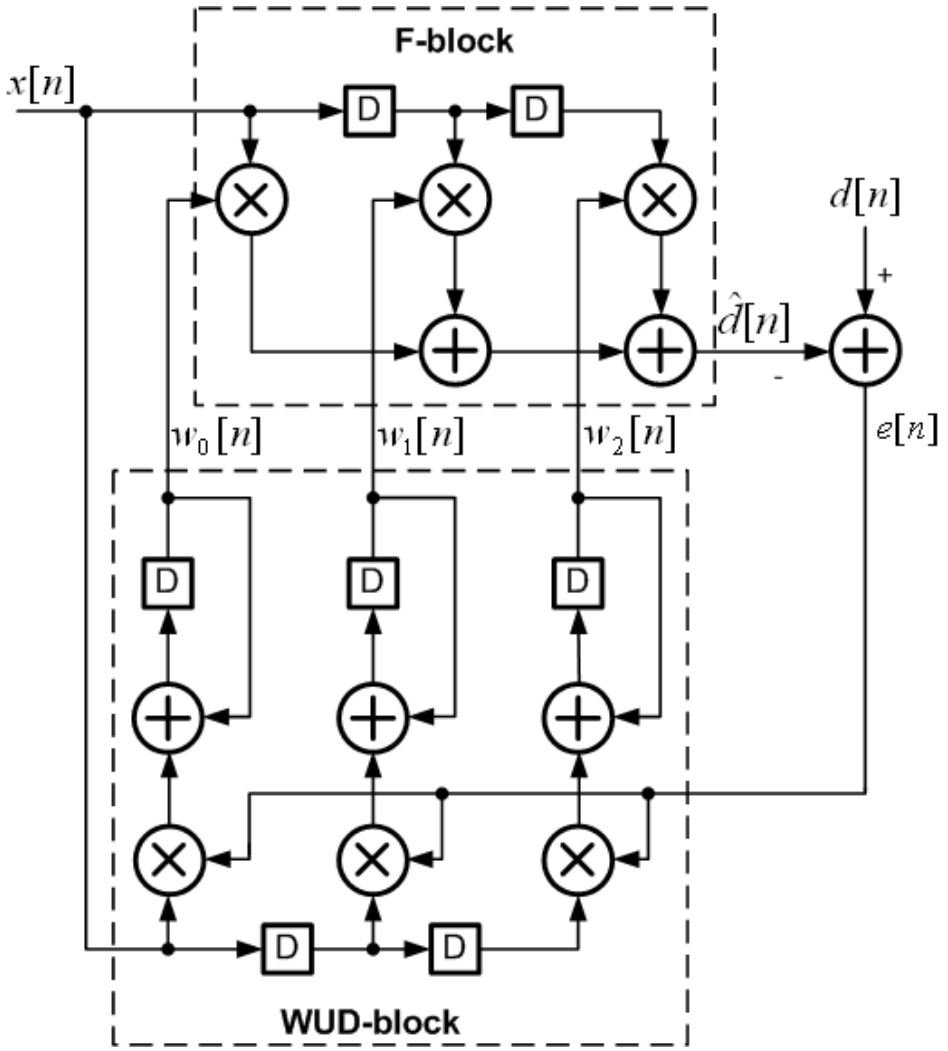
Example



$$IPB = \max \frac{\sum_U d(U)}{\sum_e w(e)}$$

- assume : $d(A) = 1$; $d(M) = 2$ then
- critical path : $M_4 \rightarrow A_4 \rightarrow A_2 \rightarrow A_0 \rightarrow A_1 \rightarrow$
- $IP = 2 + 1 + 1 + 1 + 1 = 6$
- $IPB = \max\left[\frac{4}{1}, \frac{5}{2}, \frac{5}{3}\right] = 4$
- no equivalent DFG can achieve an $IP < 4$

Example: 3-Tap LMS Adaptive Filter



D: one sample delay (register)

- critical path delay:

$$T_{cp} = 2T_m + 4T_A$$

- Throughput = PP $= 1/T_{cp}$
- $IPB = max\left[\frac{2T_m+4T_A}{1}, \frac{T_A}{1}\right] = T_{cp}$
- T_{cp} can be reduced via retiming, pipelining and parallelization

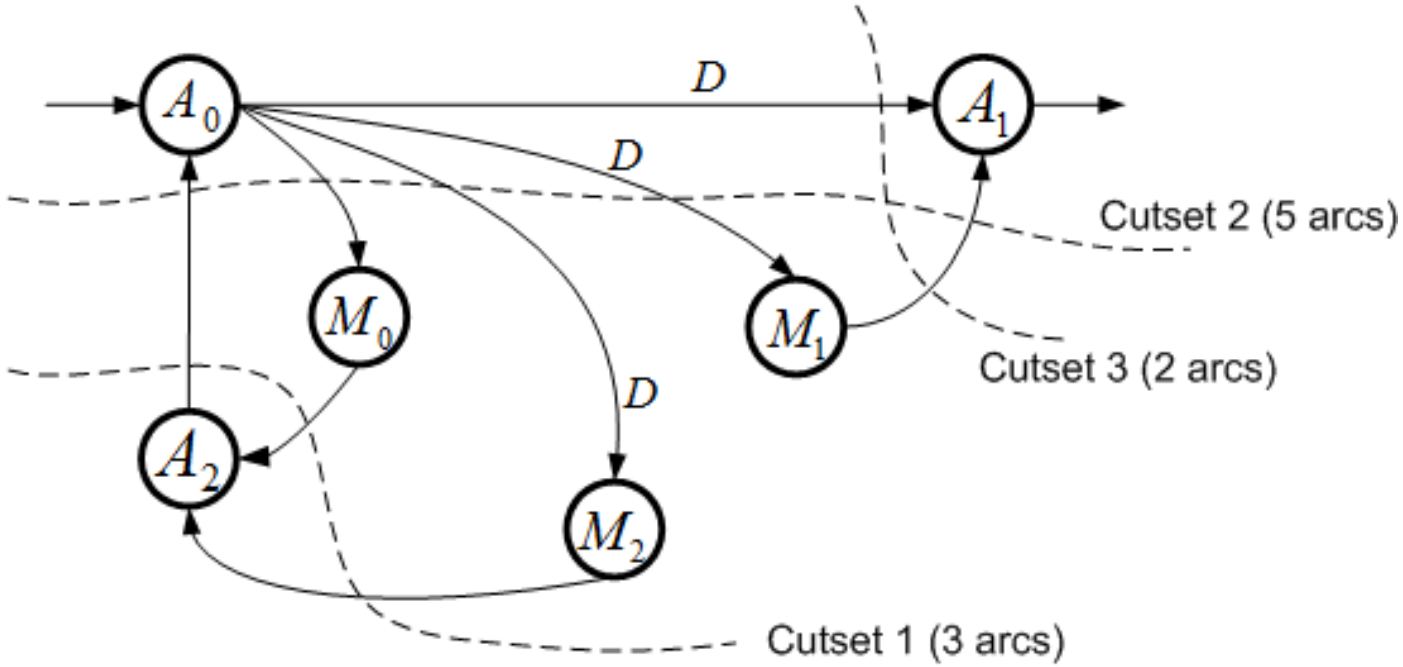
Retiming

Retiming

[leiserson]

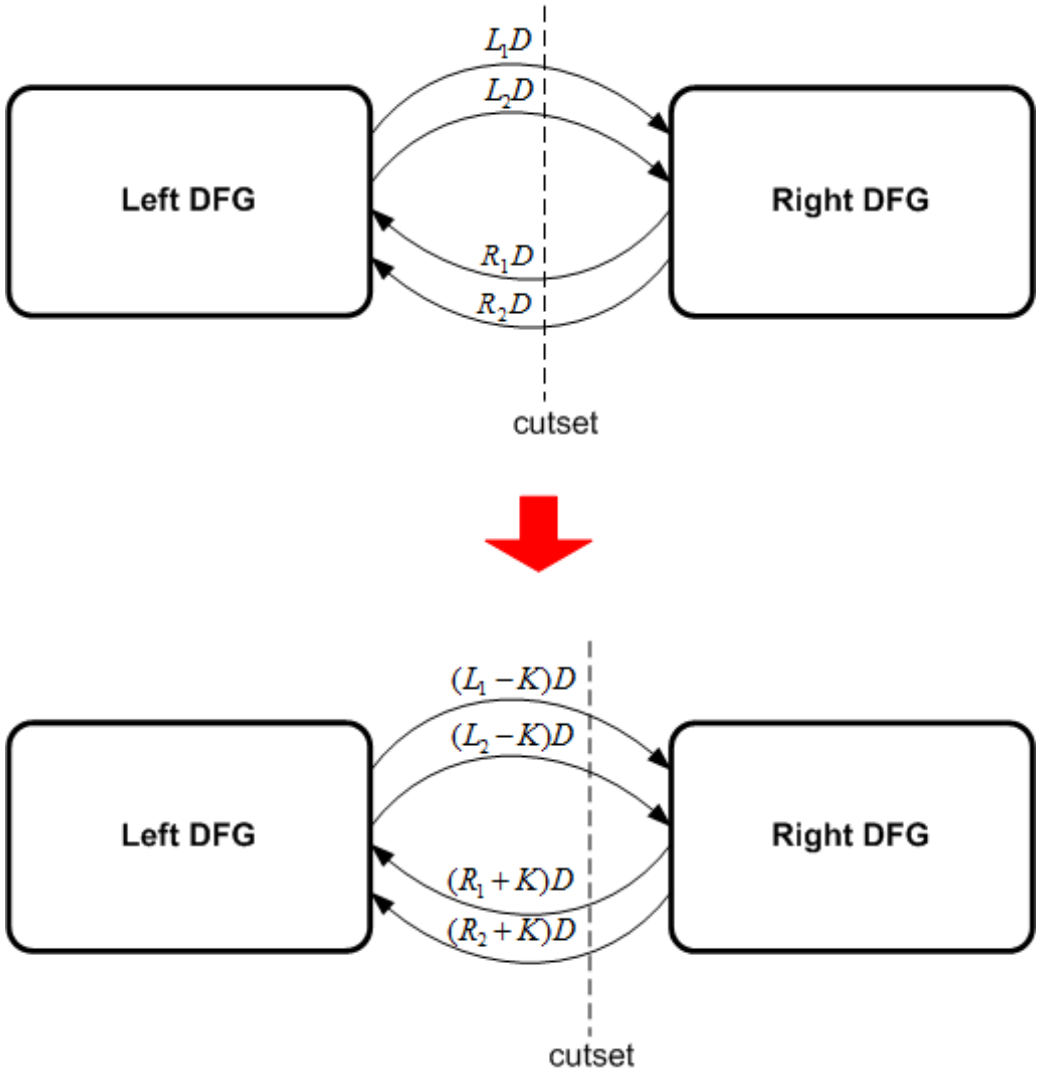
- relocate delays in DFG without changing input-output characteristics
- useful for:
  - reduces IP without changing IPB
  - reducing power
  - reducing the number of registers and resources
  - improving scheduling on multi-core architectures
  - designing systolic (regular) architectures
- 2 types
  - cutset retiming
  - systolic retiming

Cutset Retiming



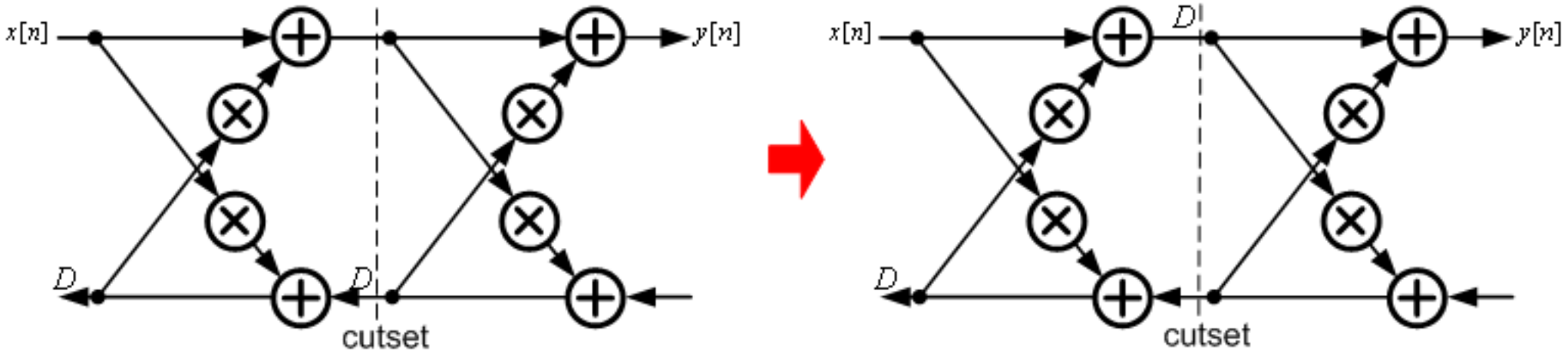
- 3 steps
  - identify a *cutset* (set of arcs that result in 2 disjoint DFGs when removed)
  - delay-scaling
  - delay transfer

Delay-Transfer



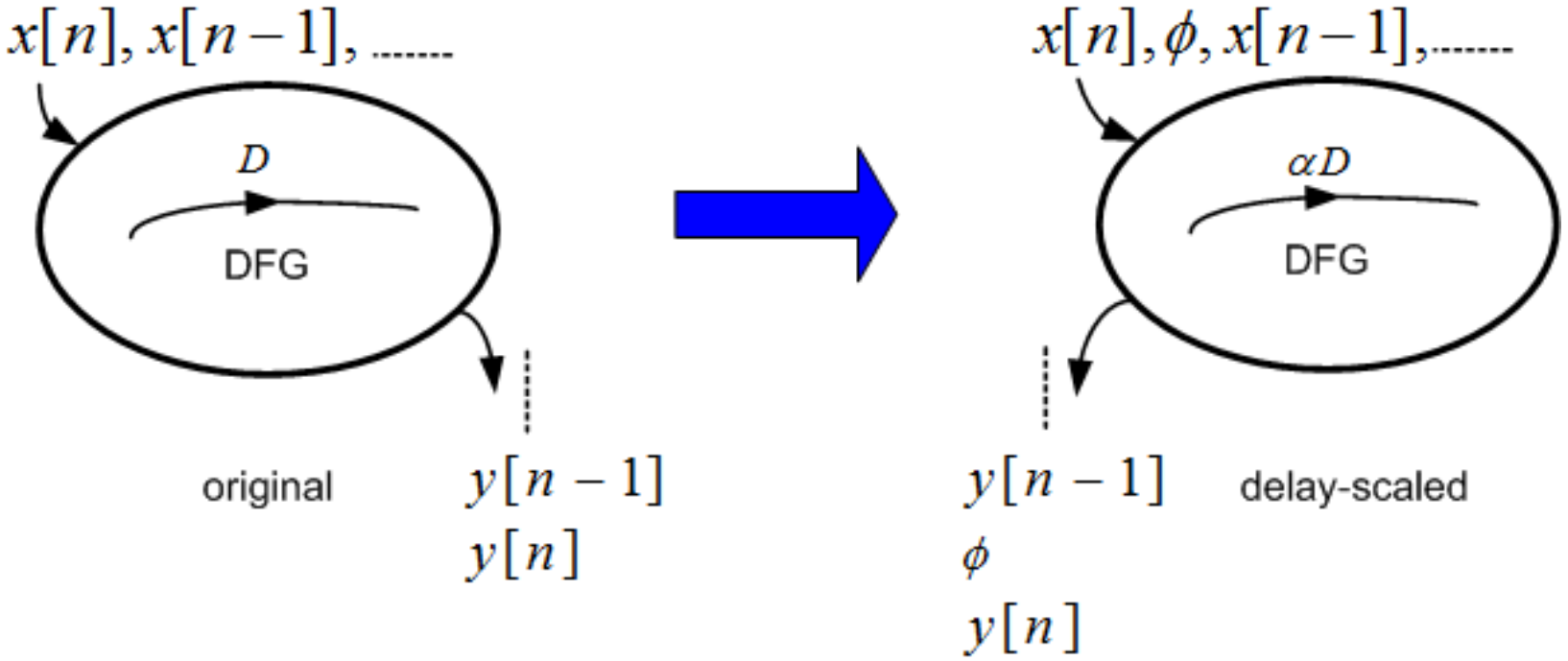
- transfer of *K* delays from in-bound (out-bound) to out-bound (in-bound) arcs of a cutset

Cutset Retiming Example



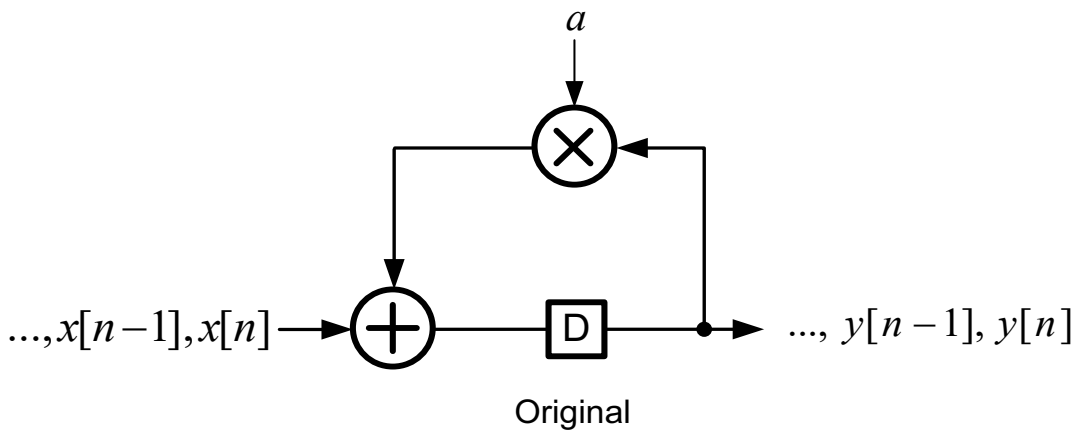
- one delay transferred from lower arc of cutset to upper arc
- assume $T_M = 2ns$ and $T_A = 1ns$;
  - $T_{cp} = 4T_M + 4T_A = 12ns$ (original)
  - $T_{cp} = 2T_M + 3T_A = 7ns$ (retimed)

Delay Scaling



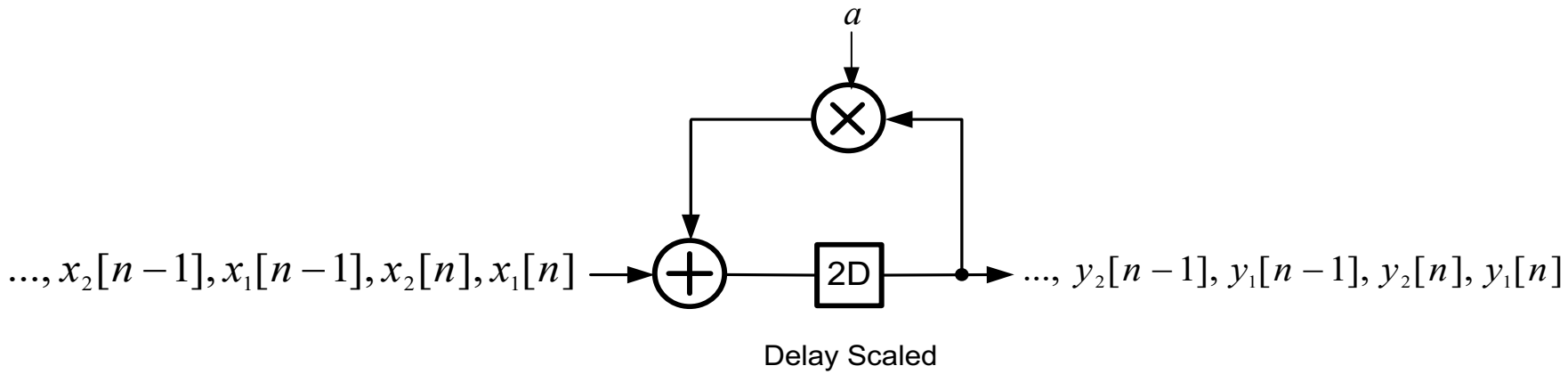
- replace all D by αD ($\alpha > 1$, delay scaling factor)
- interleave input stream by $\alpha - 1$ zero/null or independent input streams
- use multichannel processing to avoid underutilizing H/W

Delay Scaling Example



Original

- IP is unaltered



Delay Scaled

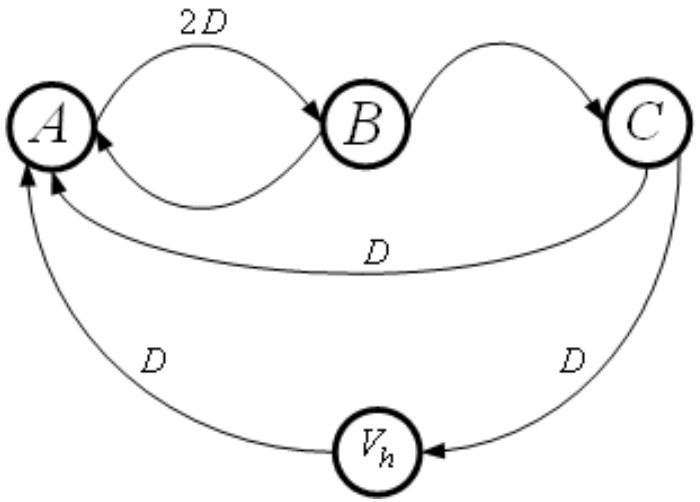
Retiming Lemma

- assign integer-valued $lag(U)$ to each node U in the original DFG
- assign new weights to each arc $e(U \rightarrow V)$: (new DFG is generated)

$$w_r(e) = w(e) + lag(V) - lag(U)$$

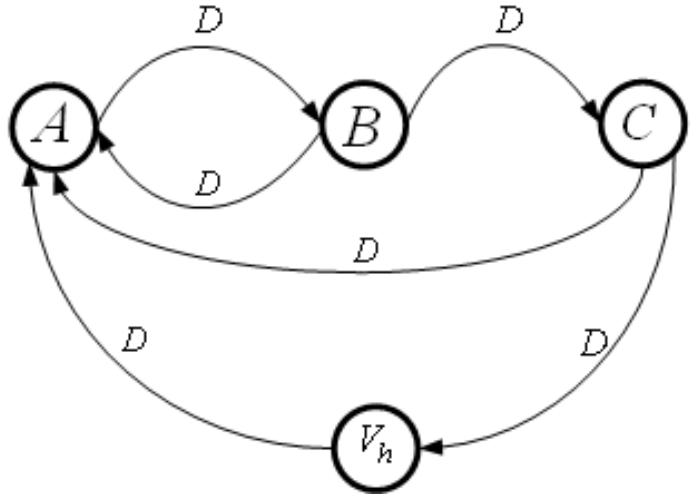
- Lemma guarantees: new (retimed) DFG is equivalent to the original one

Original DFG



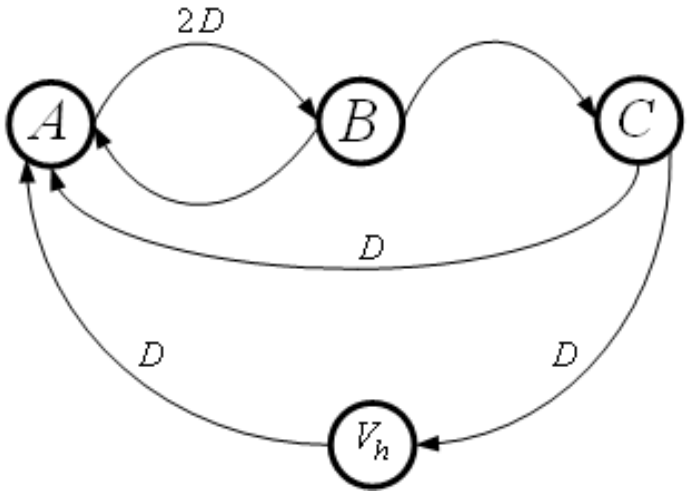
$$lag(A), lag(C), lag(V_h) = 0;$$
$$lag(B) = -1$$

Retimed DFG

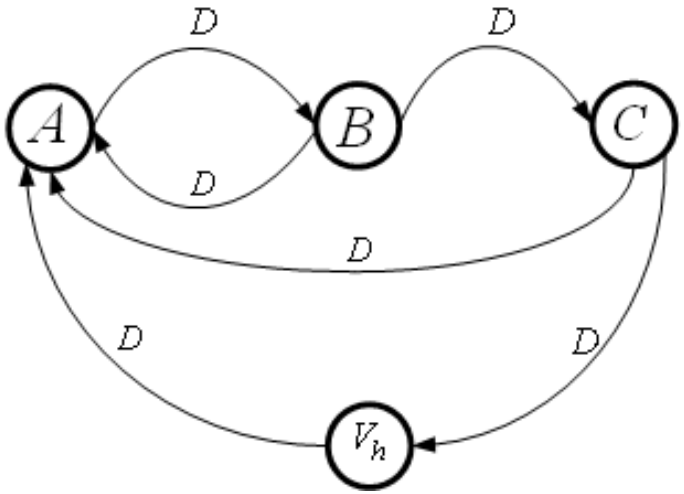


- a DFG with all arc weights $w(e) > 0$ is called systolic

Original DFG



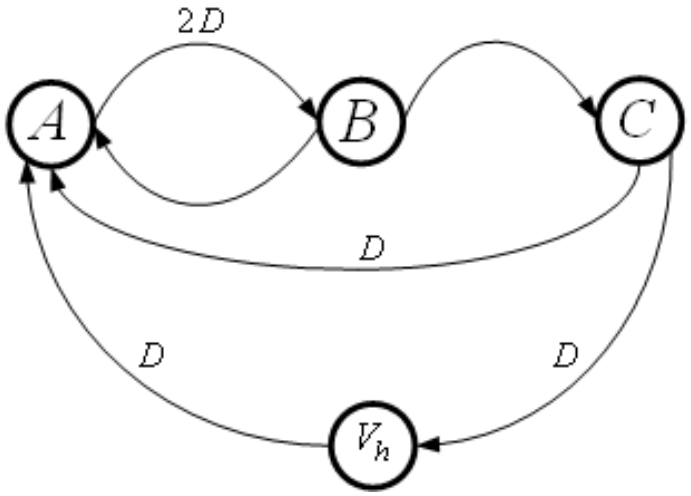
Retimed DFG



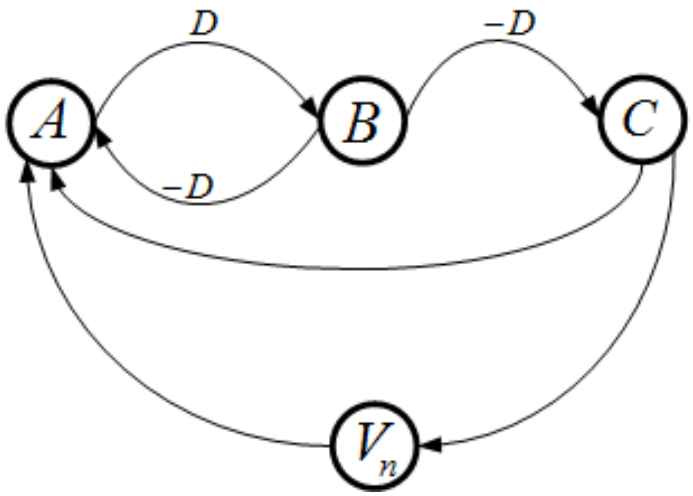
- retiming can be used to systolize a DFG
- how to find the systolizing $lag()$ function?

Systolic Conversion Theorem

Original DFG



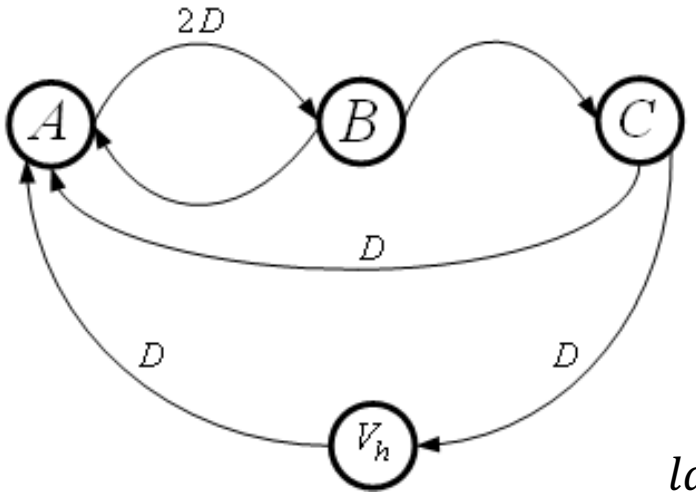
DFG G_{-1}



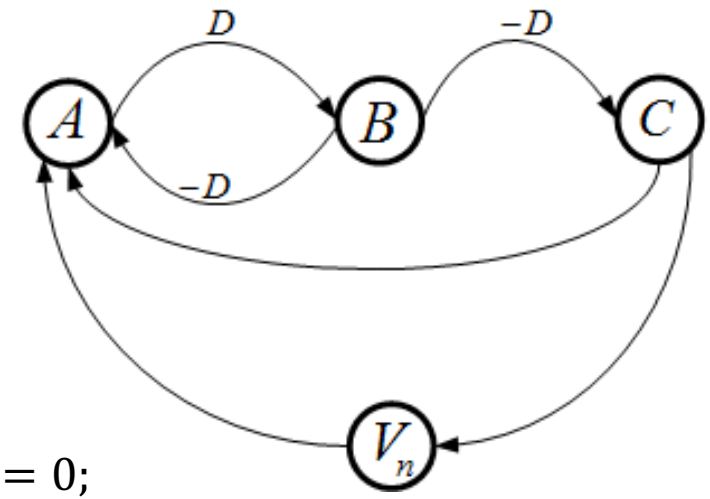
- construct a constraint graph G_{-1} by reducing all arc weights in G by 1.
- If G_{-1} does not have any negative weight cycles, then DFG G can be systolized via retiming

Finding Systolizing $lag()$ Function

Original DFG



DFG G_{-1}



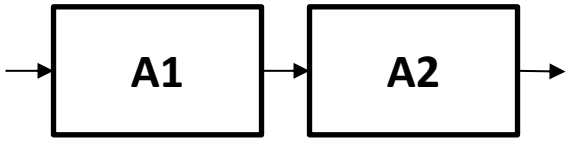
$$lag(A), lag(C), lag(V_h) = 0;$$
$$lag(B) = -1$$

- $lag(V) =$ the smallest weight of any path from V to V_h in G_{-1}
- V_h : host node with $lag(V_h) = 0$, all arcs entering/exiting V_h have zero weight for exact equivalence

Pipelining and Parallelization/Block Processing

Latency vs. Throughput

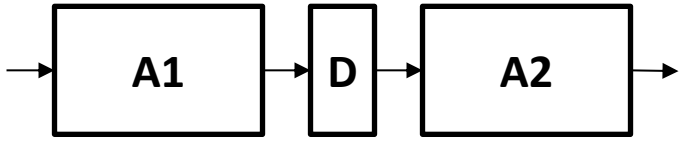
serial



Latency = $T_{A1} + T_{A2}$

Throughput = $\frac{1}{T_{A1}+T_{A2}}$

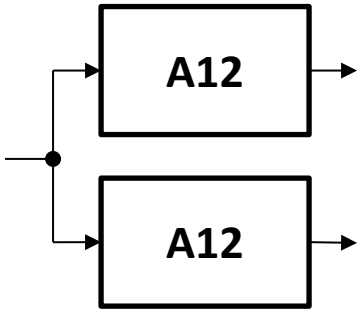
pipelined



Latency = $T_{A1} + T_{A2} + \Delta$

Throughput = $\frac{1}{\max(T_{A1},T_{A2})}$

parallel



Latency = $T_{A1} + T_{A2} + \Delta$

Throughput = $\frac{2}{T_{A1}+T_{A2}}$

Δ: timing overhead of pipelining/parallelization

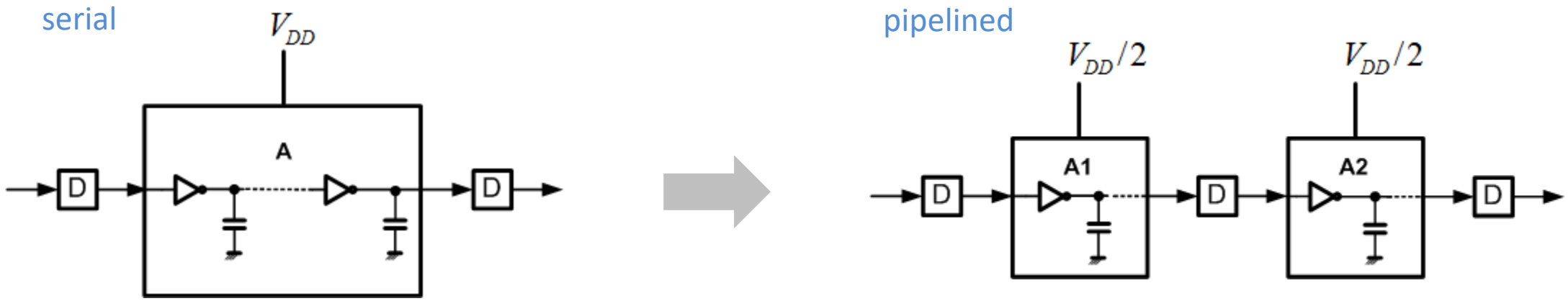
- Latency: time for the input to propagate to the output
- Throughput: rate at which outputs are generated
- In general: Throughput ≠ 1/Latency

Pipelining & Block Processing for Low-Power

IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 27, NO. 4, APRIL 1992 473

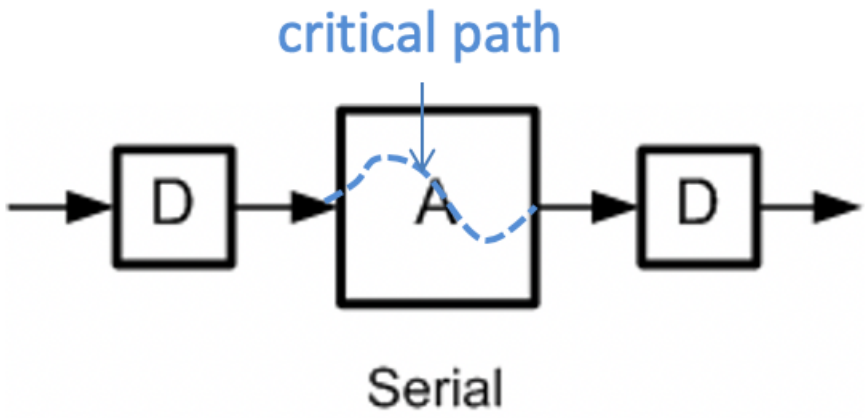
Low-Power CMOS Digital Design

Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen, *Fellow, IEEE*



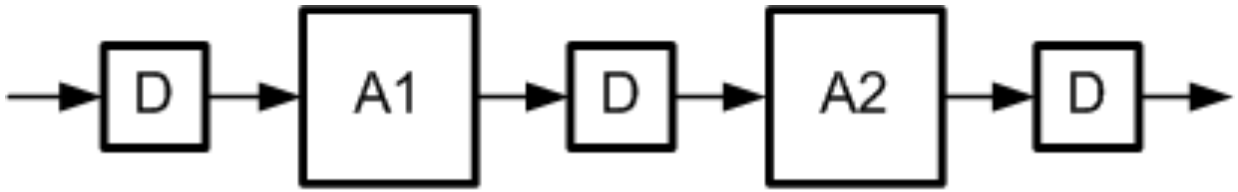
- basic idea: trade-off throughput increase from pipelining with power via supply voltage V_{dd} reduction
- what is the relationship between V_{dd}, throughput, power, energy? (next lecture)

Pipelining Non-recursive DFGs



- assume ideal registers → zero set-up, hold, clk-to-Q times
- assume timing constraints met: $T_{fast} \geq 0$ *and* $T_{cp} \leq T_{CLK}$
- maximum serial architecture clock frequency:

$$f_{CLK,serial} = \frac{1}{T_{cp,serial}}$$



Pipelined

- maximum serial architecture clock frequency:

$$f_{CLK,serial} = \frac{1}{T_{cp,serial}}$$

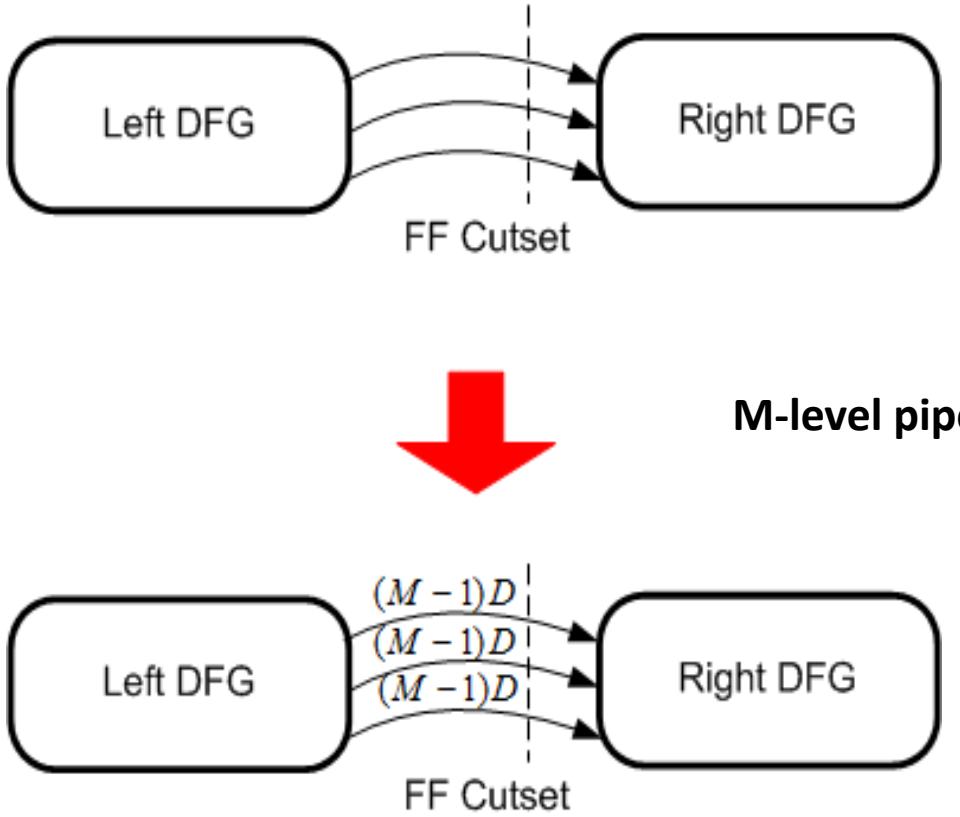
- pipelined critical path delay:

$$T_{cp,pipe} = \max\{T_{cp,A1}, T_{cp,A2}\} \leq T_{cp,serial}$$

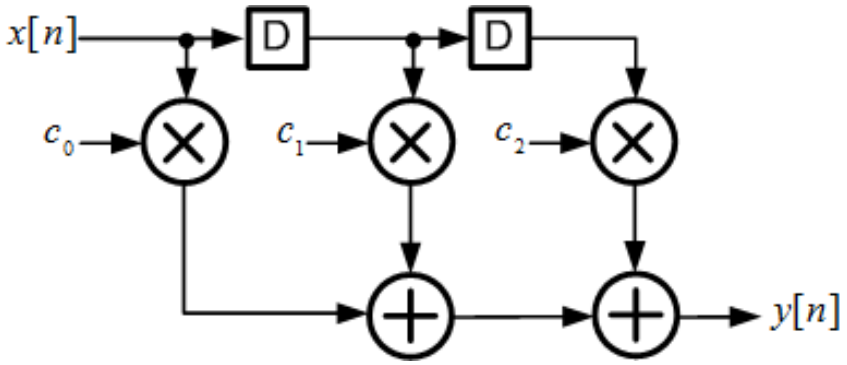
- for uniform pipelining ($T_{cp,A1} = T_{cp,A2} = 0.5T_{cp,serial}$)
  - *2x* speedup

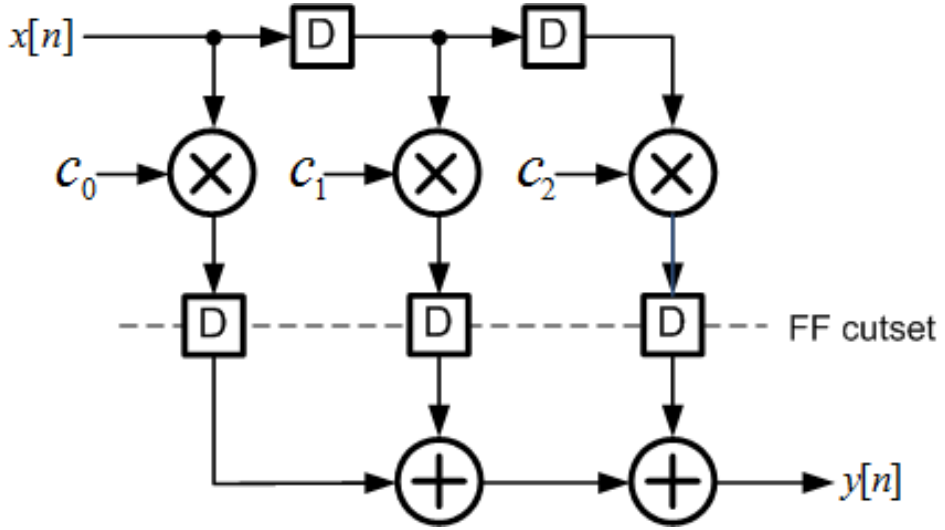
- place $M-1$ registers at a feed-forward (FF) cutset
- a FF cutset has all arcs pointing in the same direction

- speed-up compared to serial architecture=M
  - $M\times$ faster
- watch out for fast path constraint violation



Example - Pipelining an FIR Filter





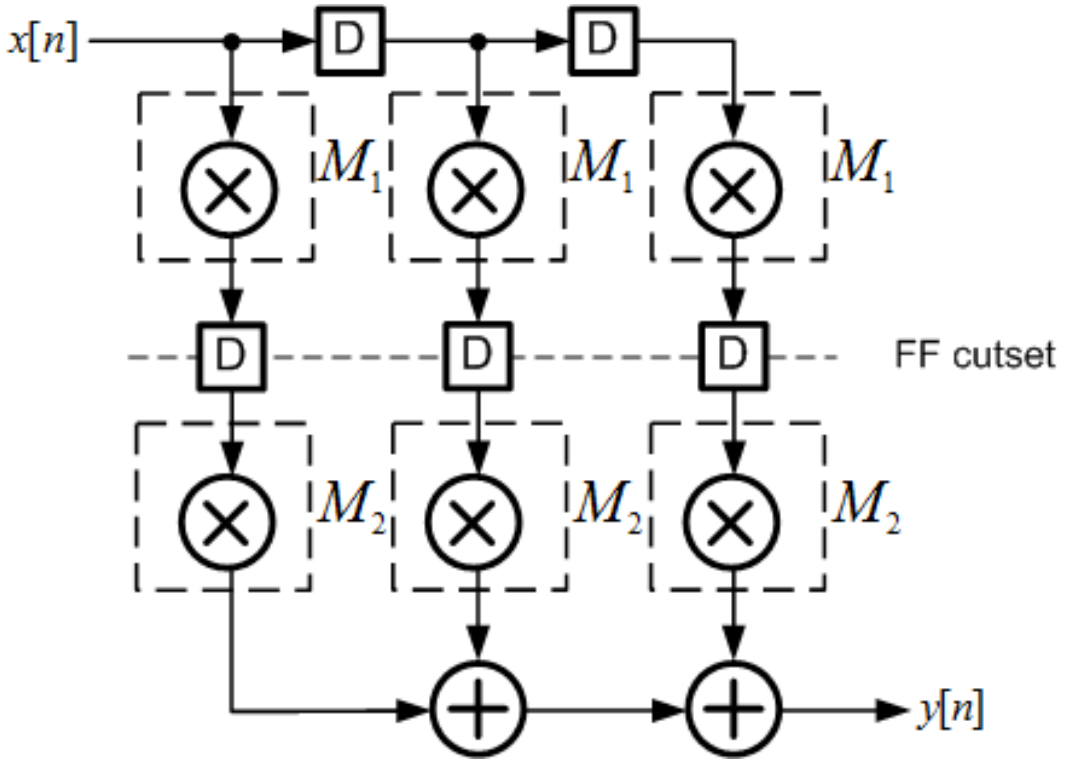
- assume $d(M) = 3ns, d(A) = 1ns$
- $T_{cp,\,serial} = d(M) + 2d(A) = 5ns$
- $T_{cp,\,pipe} = \max(d(M), 2d(A)) = 3ns$
- $\left(\frac{5}{3}\right) \times$ speed-up

Uniform Pipelining

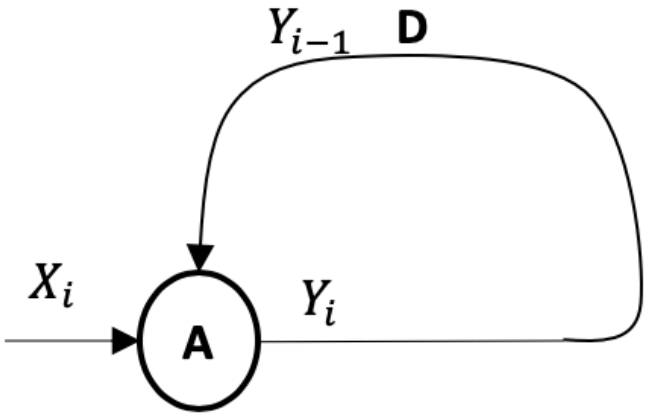
- pipeline with *equal* delay stages
- split multiplier:

$$d(M_1) = d(M_2) + 2d(A) = \ 2.5ns$$

- speed-up = $2\times$
- practical speed-up $< 2\times$ due to non-zero register delay

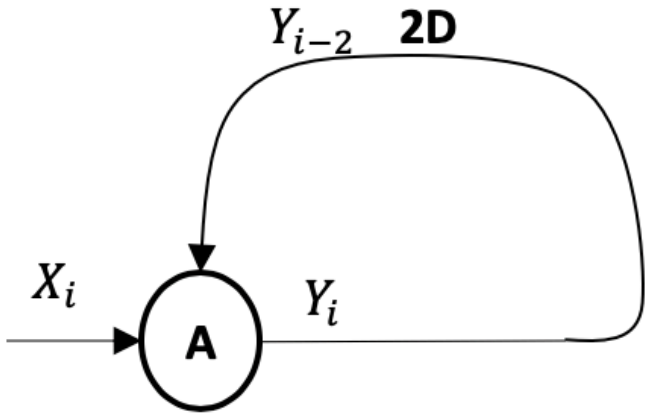


Pipelining DFGs with Loops



$$Y_i = A(X_i, Y_{i-1})$$

$\not\equiv$

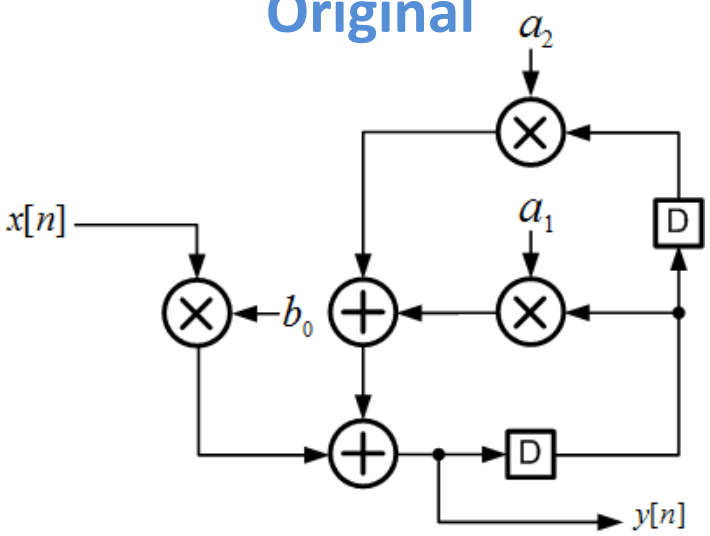


$$Y_i = A(X_i, Y_{i-2})$$

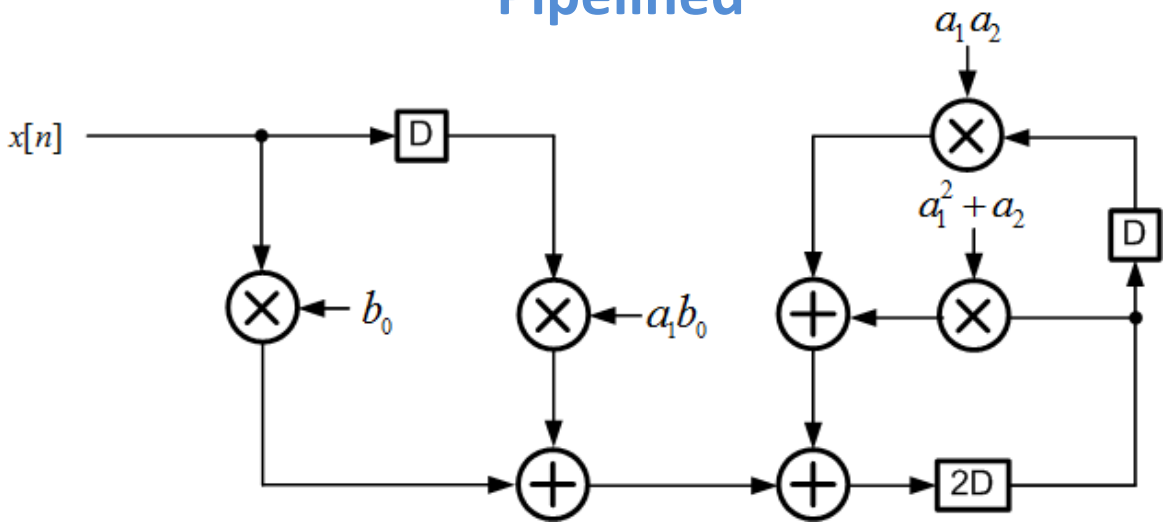
- IPB dominated by slowest loop
- no FF cutset exists
- need to introduce delays in loops to reduce IPB without altering functionality

Look-ahead Pipelining

Original



Pipelined

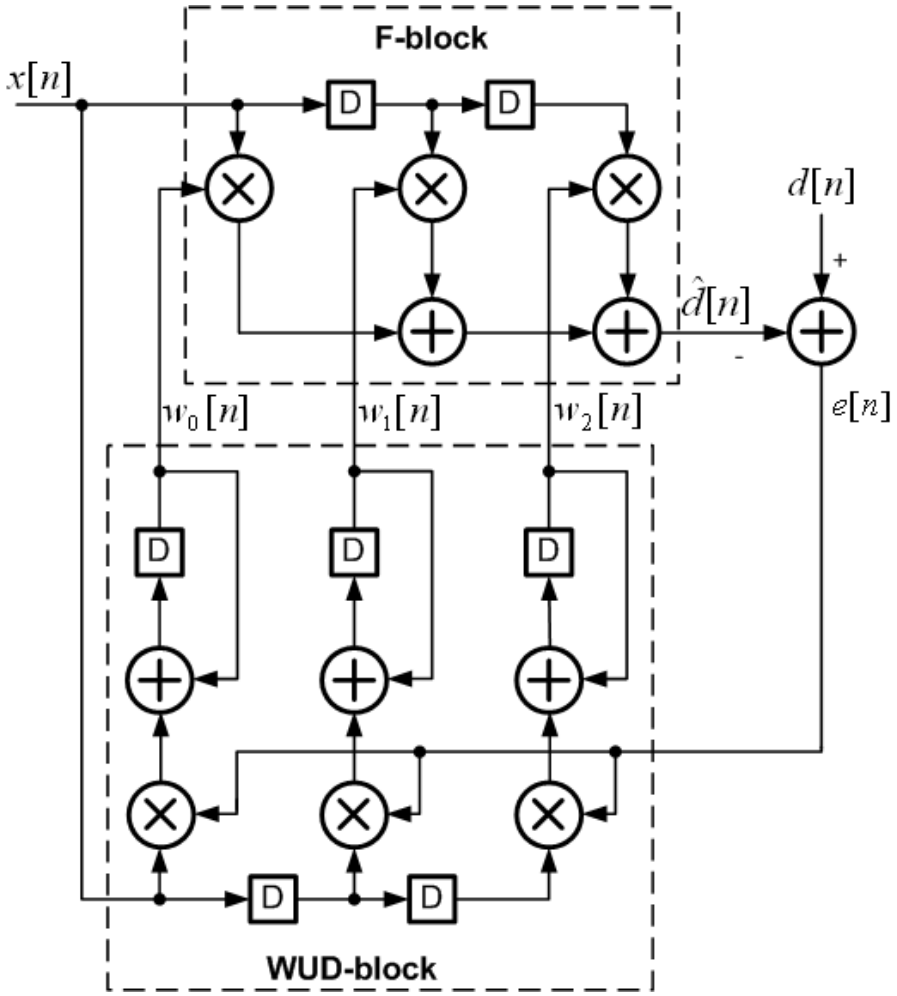


- $y[n] = b_0 x[n] + a_1 y[n-1] + a_2 y[n-2]$
- back substitute for $y[n-1]$ in terms of $y[n-2], y[n-3]$

$$\begin{aligned} y[n] &= b_0 x[n] + a_1 \left[b_0 x[n-1] + a_1 y[n-2] + a_2 y[n-3] \right] + a_2 y[n-2] \\ &= b_0 x[n] + a_1 b_0 x[n-1] + (a_1^2 + a_2) y[n-2] + a_1 a_2 y[n-3] \end{aligned}$$

- FF section (overhead) can be cutset pipelined

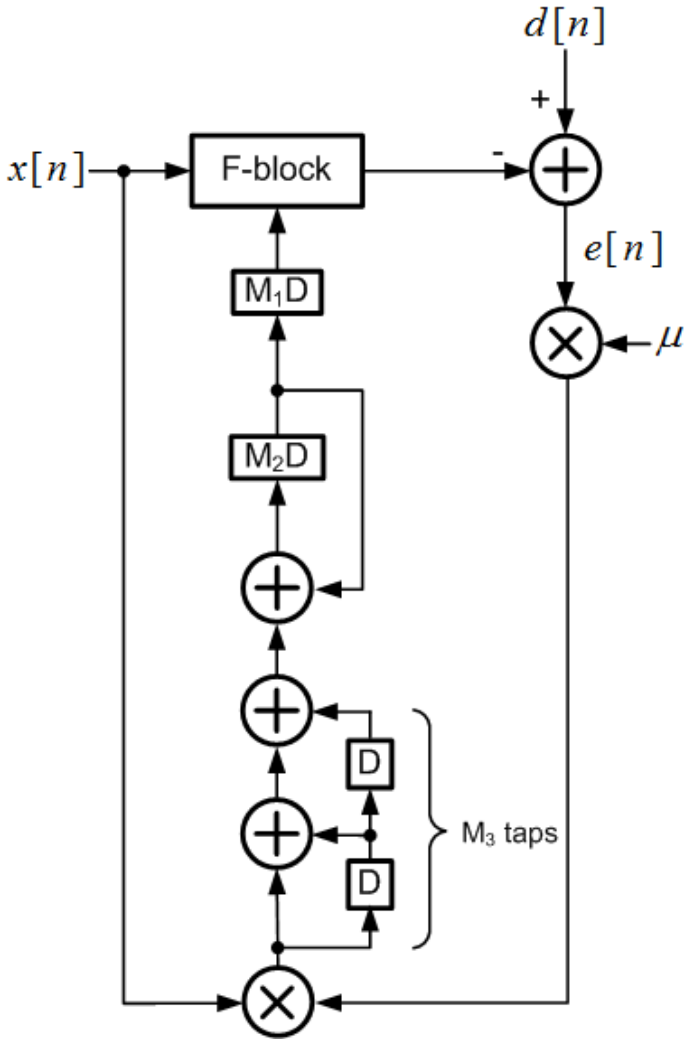
Pipelining the LMS Algorithm



$$e[n] = d[n] - \boldsymbol{W}^T[n]\boldsymbol{X}[n]$$

$$\boldsymbol{W}[n+1] = \boldsymbol{W}[n] + \mu e[n]\boldsymbol{X}[n]$$

- also has loops
- difficult to apply look-ahead directly (try it!) → need to relax the requirements of functional invariance → relaxed look-ahead



$$e[n] = d[n] - \boldsymbol{W}^T[n - M_1]\boldsymbol{X}[n]$$

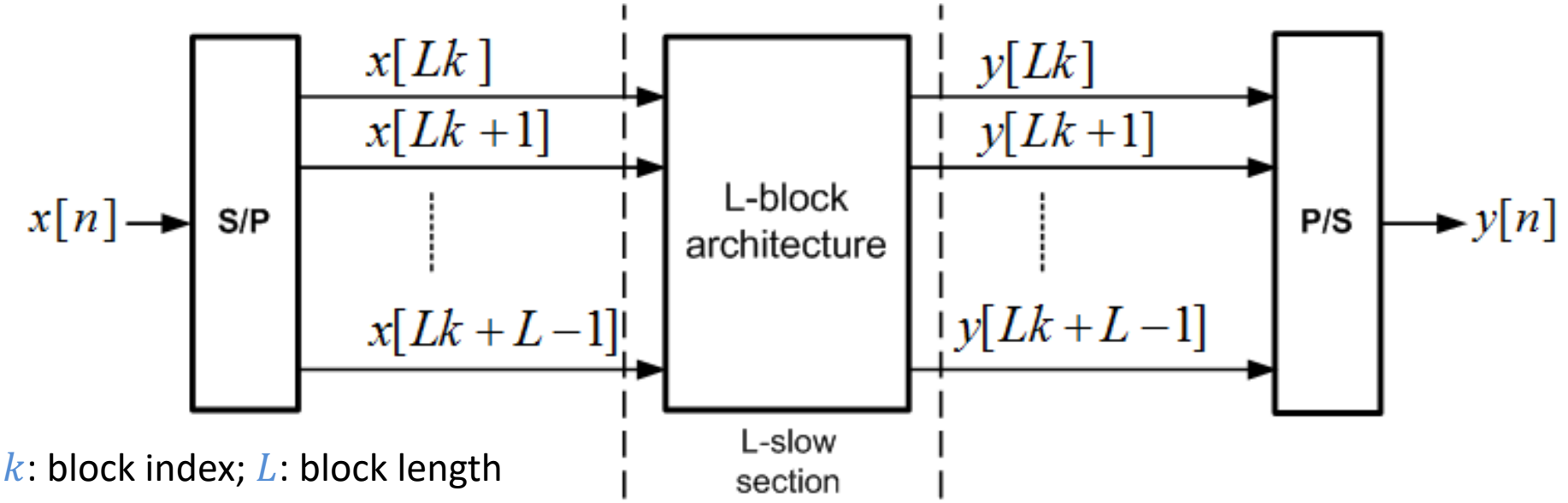
$$\mathbf{w}[n+M_2] = \mathbf{w}[n] + \mu \sum_{i=0}^{M_3-1} e^*[n-M_2+1+i]\mathbf{x}[n-M_2+1+i]$$

- M_1 (M_2) delays to pipeline outer (inner) loop
- convergence behavior is altered slightly (hence relaxed look-ahead)

Block/Parallel Processing

- improves throughput
  - unlike pipelining, does so without relying on high frequency clocks
  - useful for high-sample rate applications, e.g., optical (10+ Gb/s), chip-to-chip signaling
- reduces IPB

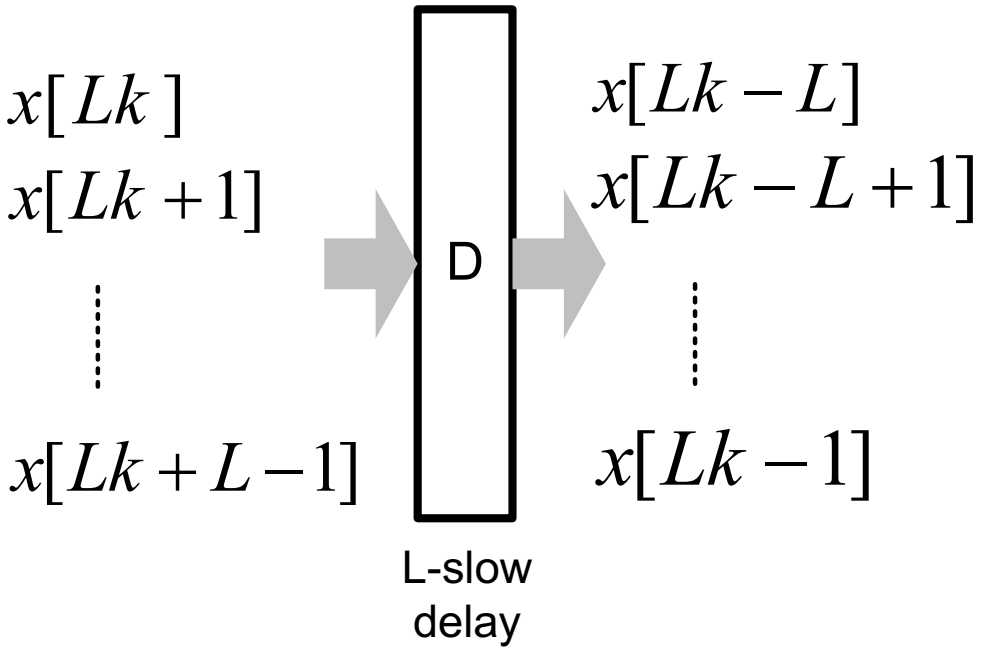
Block Architecture



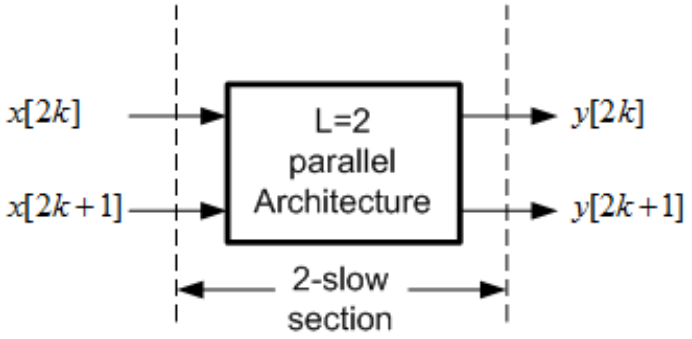
k: block index; L: block length

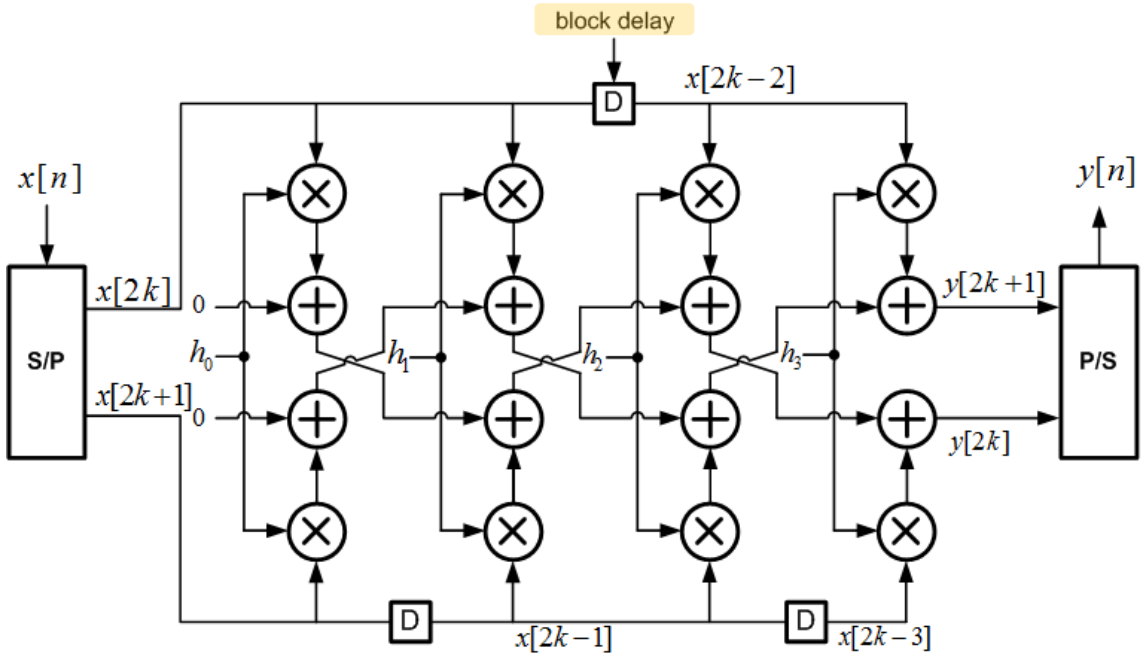
- input block $\boldsymbol{x}[k] = \left[x[Lk], x[Lk+1], \ldots, x[Lk+L-1]\right]^{\mathrm{T}}$
- output block $\boldsymbol{y}[k] = \left[y[Lk], y[Lk+1], \ldots, y[Lk+L-1]\right]^{T}$
- L-slow processor: block frequency $f_b = \frac{f_s}{L}$, i.e., L-times slower than sample frequency f_s
- L-times throughput increase without increasing clock frequency

- in a block architecture, all block delay elements are L-slow



Example: L=2, N=4, Block FIR Filter





- 4-tap FIR serial filter: $y[n] = h_0x[n] + h_1x[n-1] + h_2x[n-2] + h_3x[n-3]$
- $L = 2$, substitute n with $2k$ and $2k+1$:

$$y[2k] = h_0x[2k] + h_1x[2k-1] + h_2x[2k-2] + h_3x[2k-3]$$
$$y[2k+1] = h_0x[2k+1] + h_1x[2k] + h_2x[2k-1] + h_3x[2k-2]$$

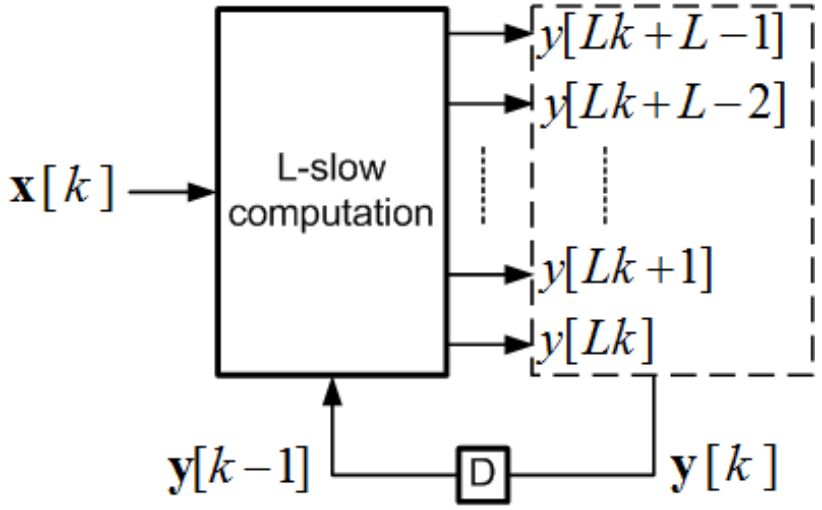
- *LN* MACs needed without computation sharing

Block Processing for DFGs with Loops

- a block IIR filter requires $\boldsymbol{y}[k]$ depend only on $\boldsymbol{y}[k-1]$ and its delayed versions

$$L = 3; N = 1$$

$$\begin{aligned} y[3k+2] &= f(x[3k+2], y[3k-1]) \\ y[3k+1] &= f(x[3k+1], y[3k-2]) \\ y[3k] &= f(x[3k], y[3k-3]) \end{aligned}$$

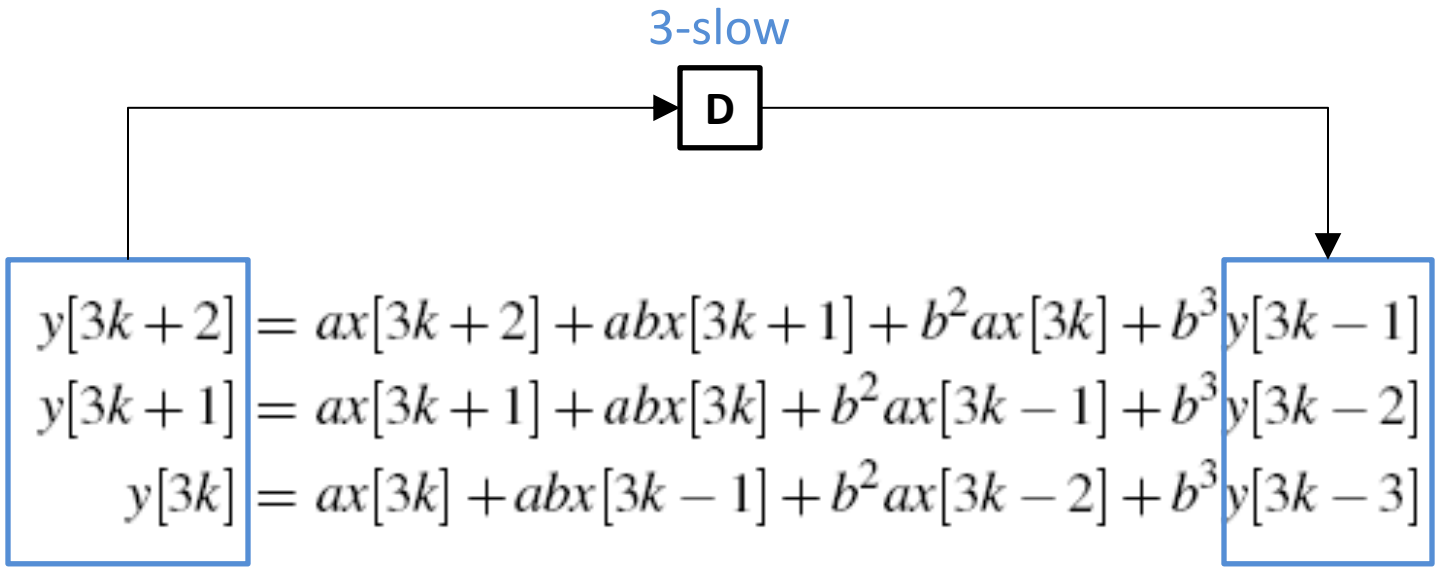


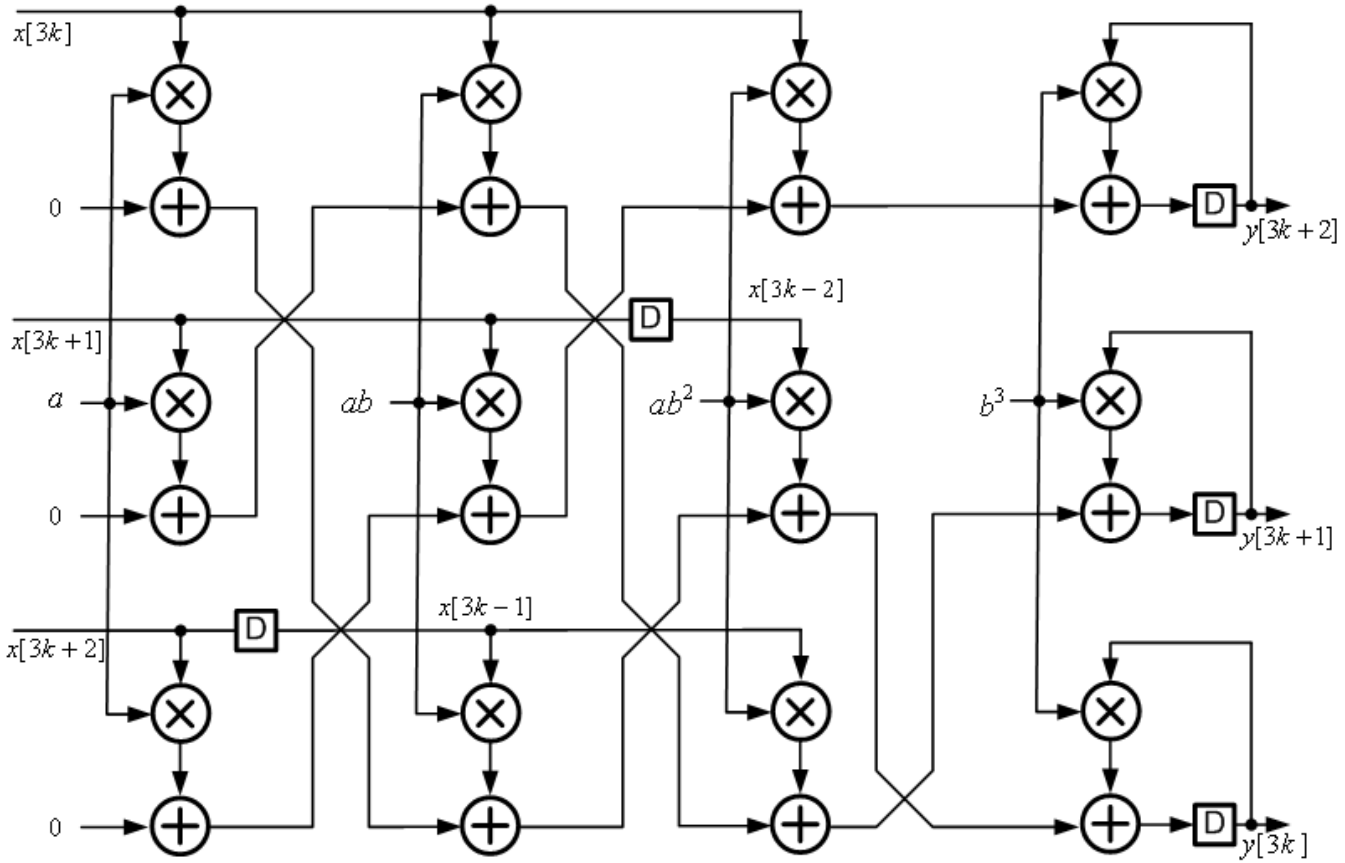
- How to get the function $f()$?

- use ($L = 3$-step) look-ahead to obtain *f()* $\quad y[n] = ax[n] + by[n-1]$

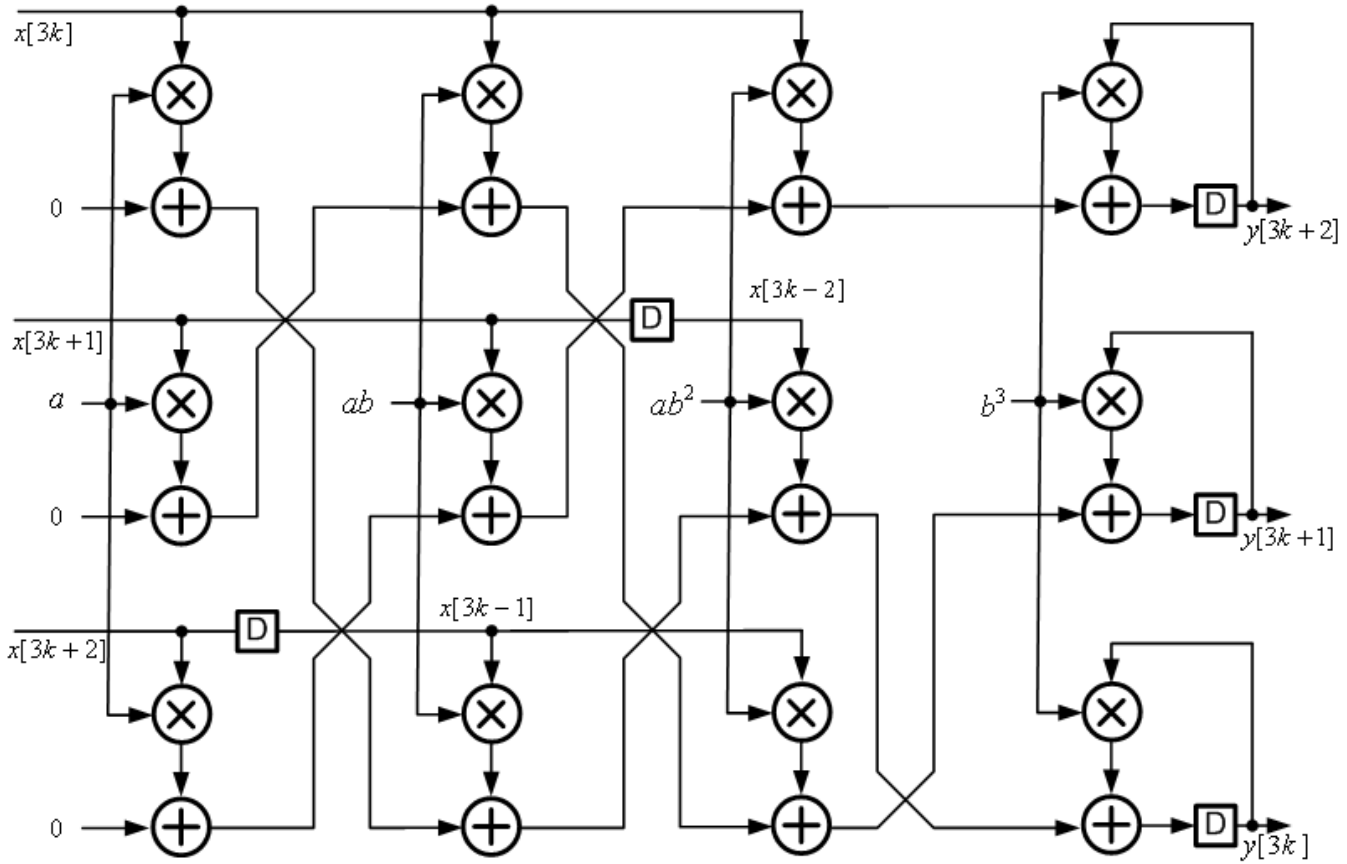
$$y[n] = ax[n] + abx[n-1] + b^2ax[n-2] + b^3y[n-3]$$

- Substitute $n = 3k+2, 3k+1$, and $3k \rightarrow$ 3-parallel IIR filter:





$$y[3k+2] = ax[3k+2] + abx[3k+1] + b^2ax[3k] + b^3y[3k-1]$$
$$y[3k+1] = ax[3k+1] + abx[3k] + b^2ax[3k-1] + b^3y[3k-2]$$
$$y[3k] = ax[3k] + abx[3k-1] + b^2ax[3k-2] + b^3y[3k-3]$$



- T_{cp} is identical (after pipelining FF section) to that of a serial architecture, i.e., $T_{cp} = T_M + T_A$
- However, L samples processed per T_{cp} seconds → L –fold speed-up is achieved

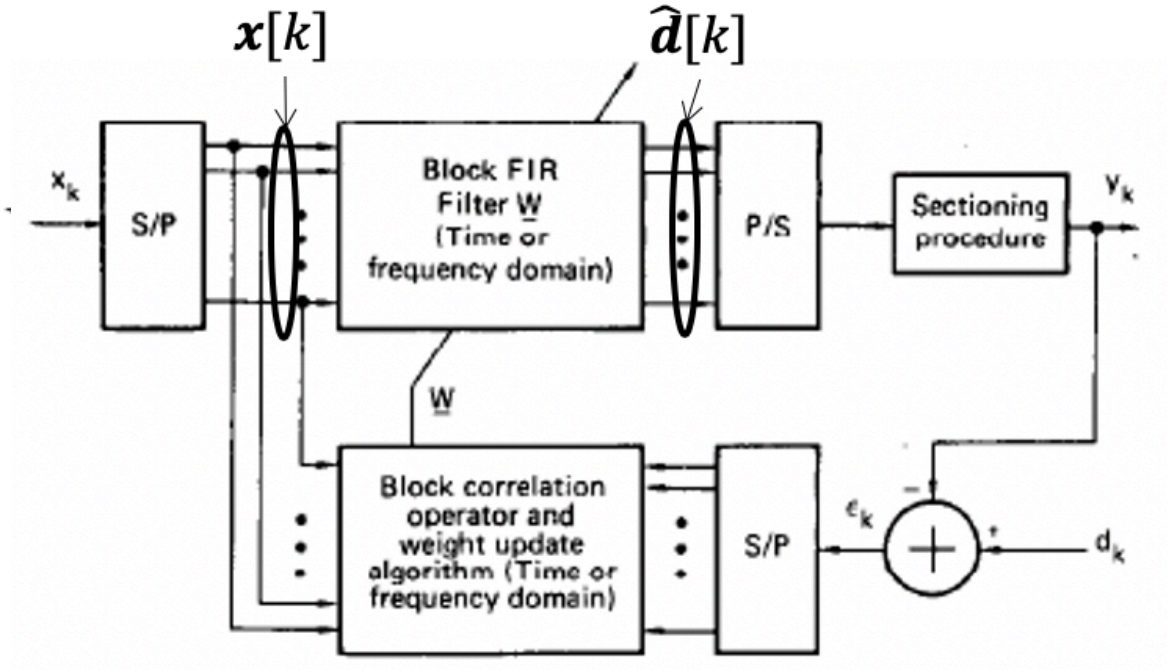
Block (Batch) LMS

- processes block of data $\boldsymbol{x}[k]$ of length L to generate a block of output $\widehat{\boldsymbol{d}}[k]$
- weights adjusted once per block → batch-mode processing
- equivalent to serial LMS when $L = 1$

Block Implementation of Adaptive Digital Filters

GREGORY A. CLARK, STUDENT MEMBER, IEEE, SANJIT K. MITRA, FELLOW, IEEE, AND SYDNEY R. PARKER, FELLOW, IEEE

IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, VOL. CAS-28, NO. 6, JUNE 1981



k: block index

$$\boldsymbol{e}[k] = \boldsymbol{d}[k] - \boldsymbol{\chi}[k]\boldsymbol{W}[k]$$

$$\boldsymbol{W}[k+1] = \boldsymbol{W}[k] + \frac{\mu}{L} \sum_{i=0:L-1} e[kL-i]\, \boldsymbol{x}_i[k]$$

$$\begin{matrix} \chi_1 \\ \chi_2 \end{matrix} \left\{ \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ \hline x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ \hline x_7 & x_6 & x_5 \\ & \vdots & \end{bmatrix} \right. \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}_{\boldsymbol{W}} = \left. \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \hline y_4 \\ y_5 \\ y_6 \\ \hline y_7 \\ \vdots \end{bmatrix} \right. \begin{matrix} \} \boldsymbol{Y}_1 \\ \} \boldsymbol{Y}_2 \end{matrix} . \quad = \widehat{\boldsymbol{d}}[k]$$

- $\boldsymbol{W}[k] = \left[w_o[k], \ldots, w_{N-1}[k]\right]^T$: same length as serial LMS; updated once per block of L samples
- update term is L-times larger hence step-size μ (same as in serial LMS) is reduced by a factor of L

k: block index

$$\boldsymbol{e}[k] = \boldsymbol{d}[k] - \boldsymbol{\chi}[k]\boldsymbol{W}[k]$$

$$\boldsymbol{W}[k+1] = \boldsymbol{W}[k] + \frac{\mu}{L} \sum_{i=0:L-1} e[kL-i]\, \boldsymbol{x}_i[k]$$

$$\begin{matrix} \chi_1 \\ \chi_2 \end{matrix} \left\{ \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ \hline x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ x_6 & x_5 & x_4 \\ \hline x_7 & x_6 & x_5 \\ & \vdots & \end{bmatrix} \right. \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}}_{\boldsymbol{W}} = \left. \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \hline y_4 \\ y_5 \\ y_6 \\ \hline y_7 \\ \vdots \end{bmatrix} \right. \begin{matrix} \} \boldsymbol{Y}_1 \\ \} \boldsymbol{Y}_2 \end{matrix} . \quad = \widehat{\boldsymbol{d}}[k]$$

- $\boldsymbol{W}[k] = \left[w_o[k], \ldots, w_{N-1}[k]\right]^T$
- $\boldsymbol{\chi}[k] = \left[\boldsymbol{x}_0[k], \boldsymbol{x}_1[k], \ldots, \boldsymbol{x}_{L-1}[k]\right]^T$
- $\boldsymbol{x}_i[k] = \left[x[kL-i], x[kL-i-1], \ldots, x[kL-i-N+1]\right]^T$

Convergence Properties

- optimum Wiener-Hopf equation identical to serial LMS: $\boldsymbol{w}_{opt} = \boldsymbol{R}^{-1}\boldsymbol{P}$
- stability bounds identical to serial LMS: $0 < \mu < \frac{2}{N\sigma_X^2}$
- convergence speed → L-times slower due to infrequent updates
- L times better accuracy (misadjustment) than serial LMS

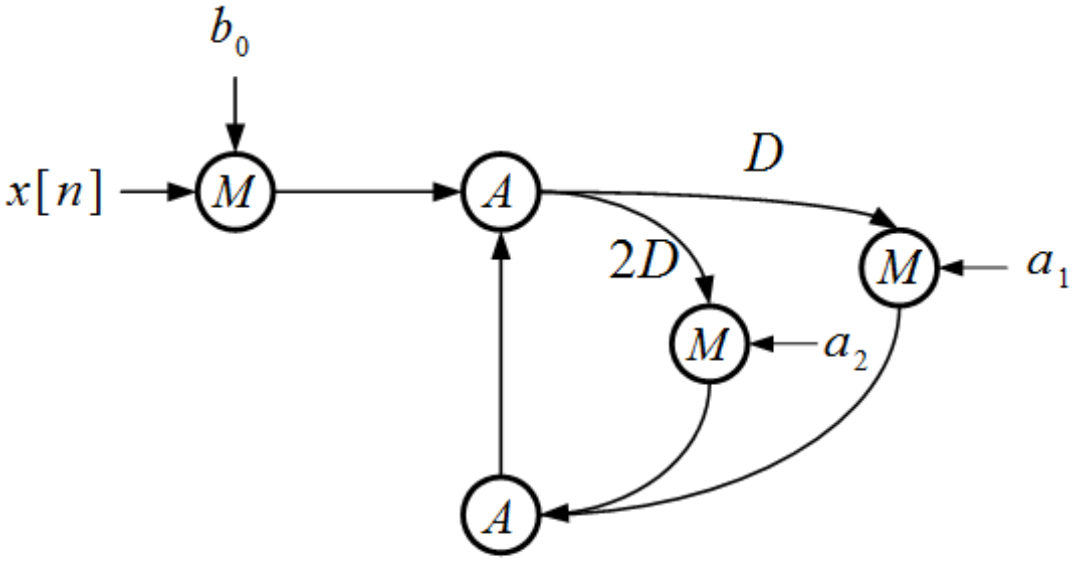
$$\eta = \frac{J(\infty) - J_{min}}{J_{min}} = \frac{\mu}{2L}\,\mathrm{tr}(R)$$

Folding and Unfolding

Unfolding

- also known as loop unrolling
- a one-to-one transform
- exposes inter-iteration precedence in a DFG
  - Unfolding by a factor *J* exposes *J* iterations
- benefits
  - generation of rate-optimal multi-core/processor schedules
  - systematic design of digit-serial architectures from bit-serial architecture
  - circuit/logic level power and speed optimization
- does not reduce IPB

Unfolding Example

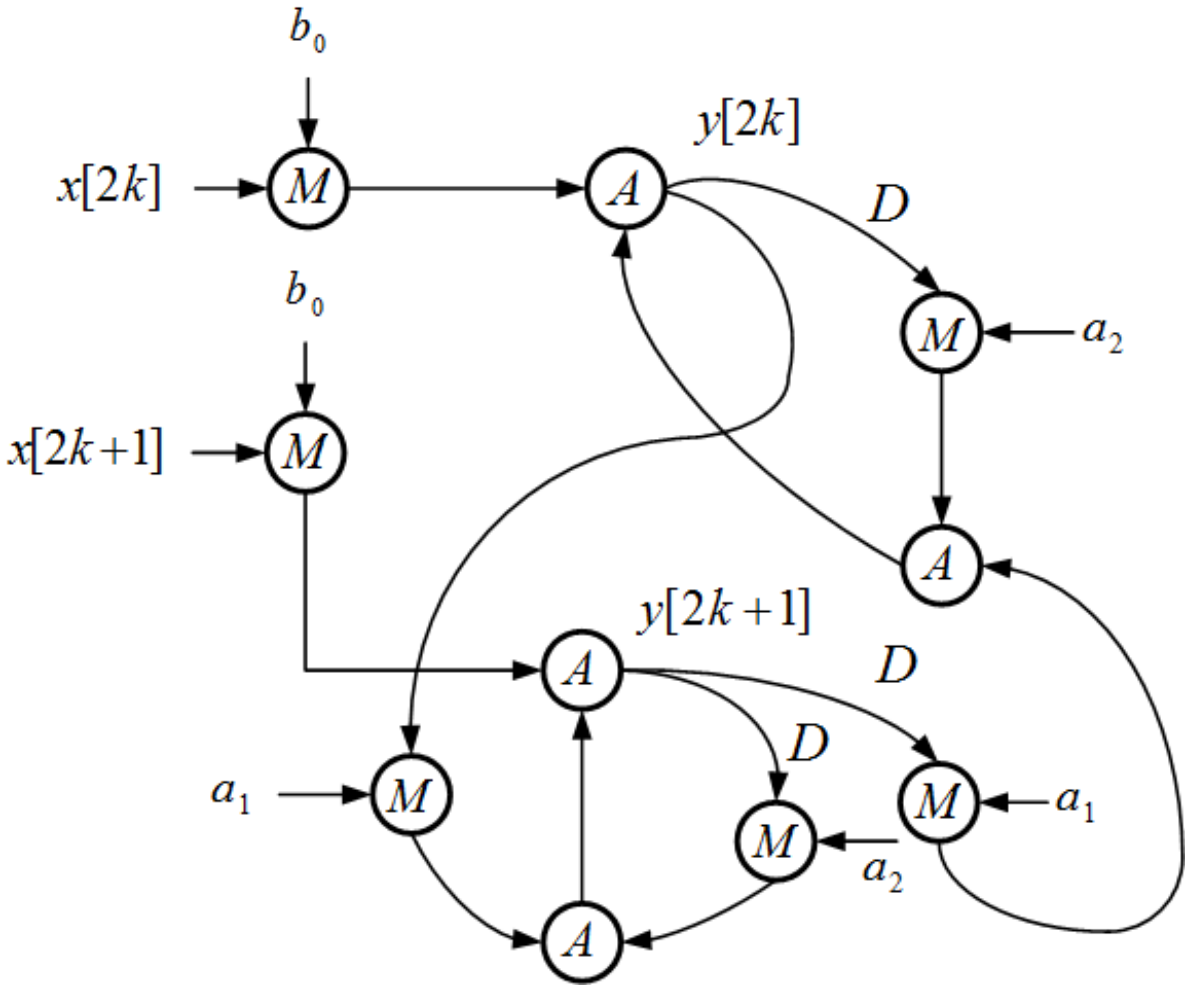


Original

- second order IIR filter: $y[n] = b_0 x[n] + a_1 y[n-1] + a_2 y[n-2]$
- substitute $n = 2k$ and $n = 2k+1$ ($J = 2$)

$$y[2k] = b_0 x[2k] + a_1 y[2k-1] + a_2 y[2k-2]$$
$$y[2k+1] = b_0 x[2k+1] + a_1 y[2k] + a_2 y[2k-1]$$

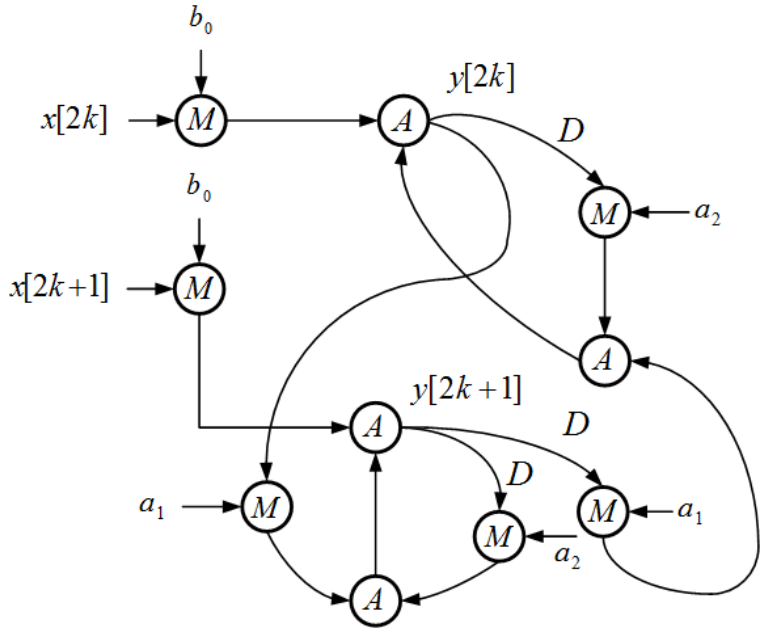
Unfolding Example (J=2)



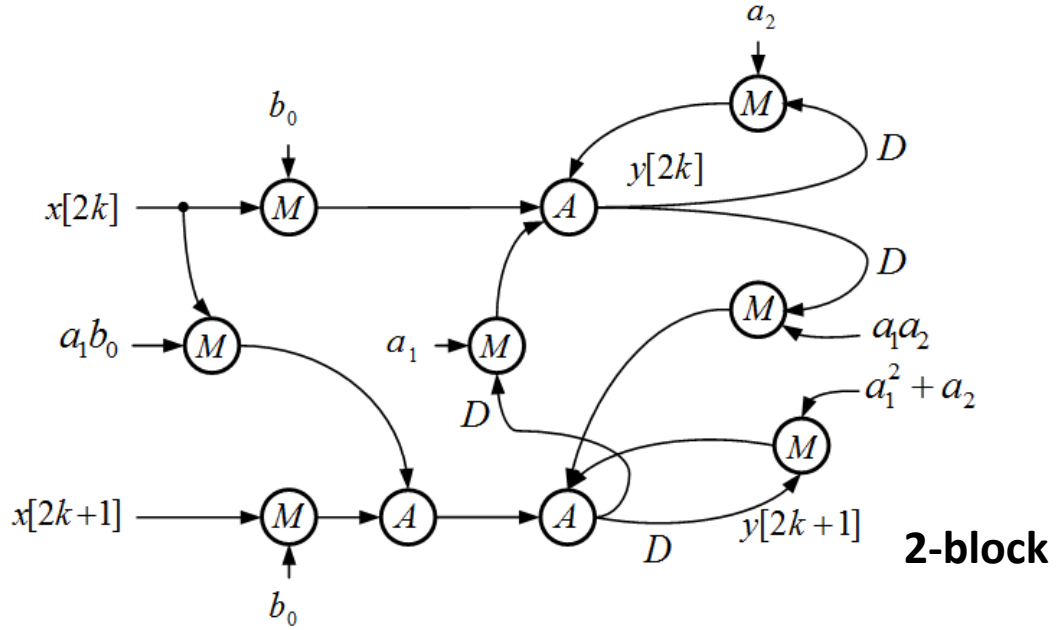
J-slow delays

$$y[2k] = b_0x[2k] + a_1y[2k-1] + a_2y[2k-2]$$
$$y[2k+1] = b_0x[2k+1] + a_1y[2k] + a_2y[2k-1]$$

Unfolding and Block Processing



2-unfolded



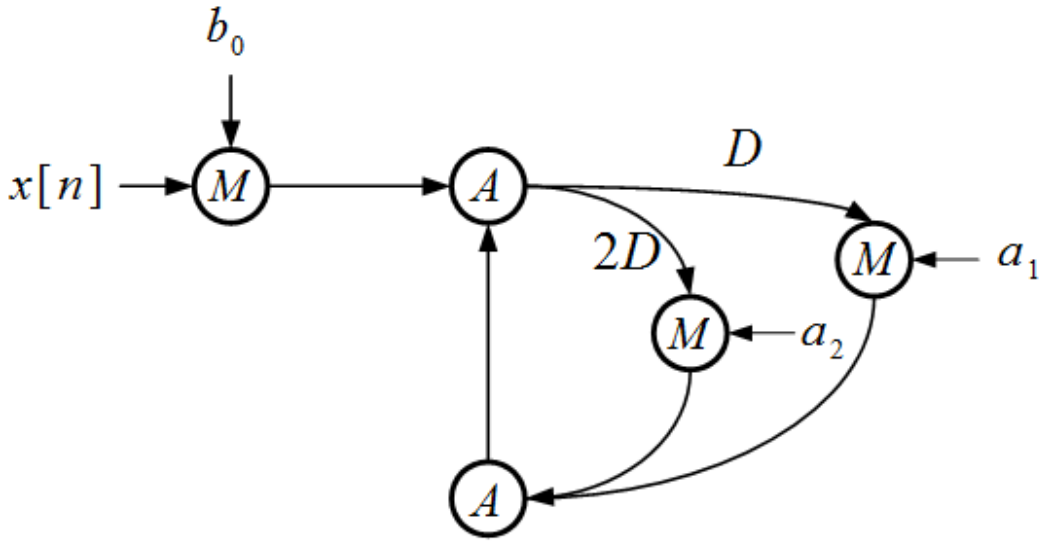
2-block

$$y[2k] = b_0x[2k] + a_1y[2k-1] + a_2y[2k-2]$$

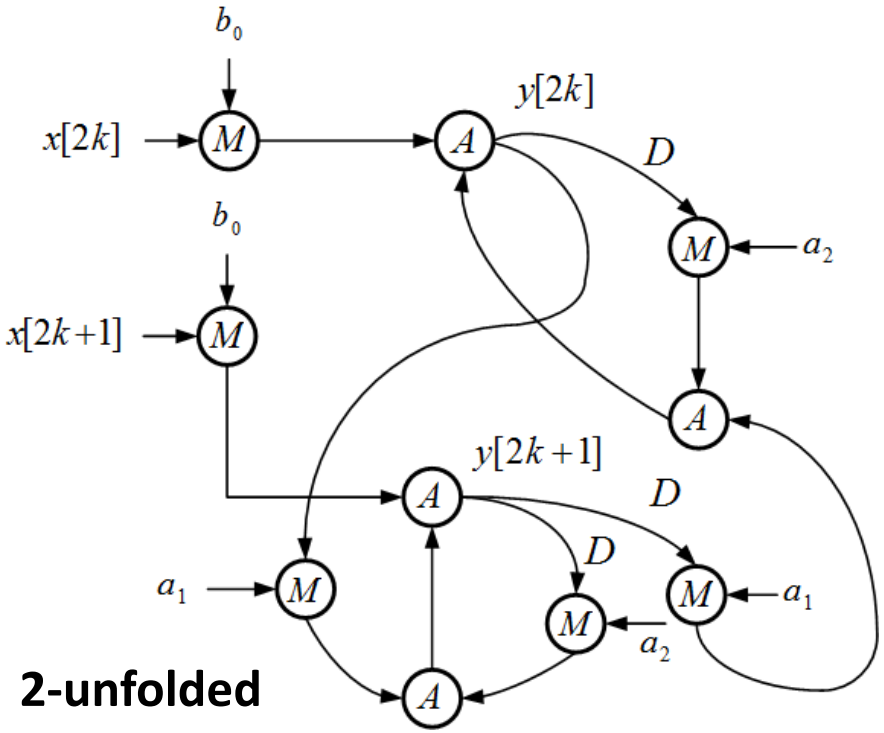
$$y[2k+1] = b_0x[2k+1] + a_1b_0x[2k] + (a_1^2 + a_2)y[2k-1] + a_1a_2y[2k-2]$$

- unfolding and block processing are *identical for non-recursive DFGs*. For recursive DFGs, unfolding computes current state from immediate past state while block processing uses look-ahead
- unfolding can achieve $IP = IPB$ *without fine-grain* pipelining

Unfolding Property 1



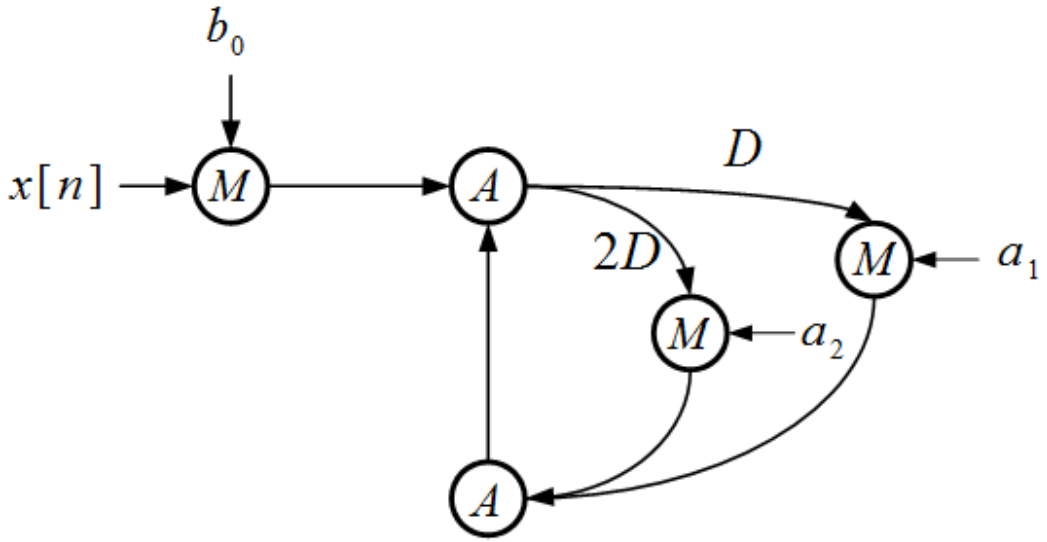
original



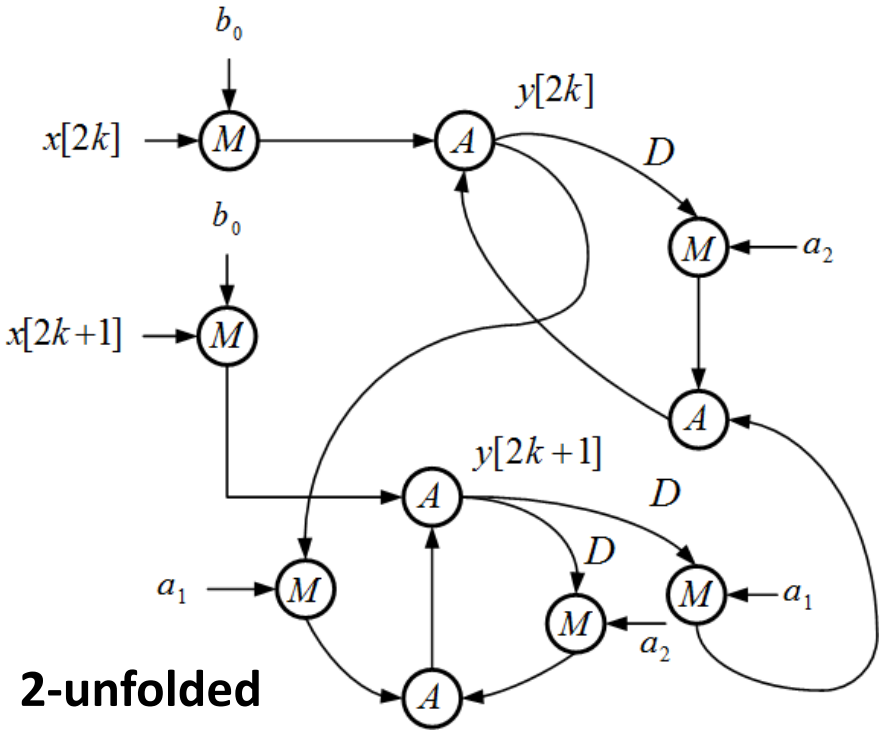
2-unfolded

- unfolding ***preserves*** *the number of delays*, and leads to a J *—fold* ***increase*** *in the number of nodes*

Unfolding Property 2



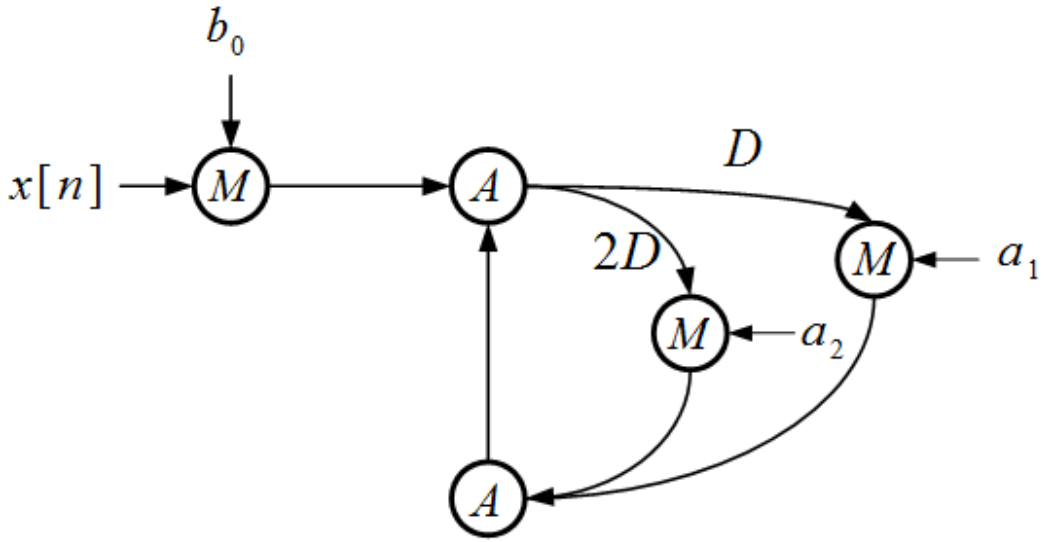
original



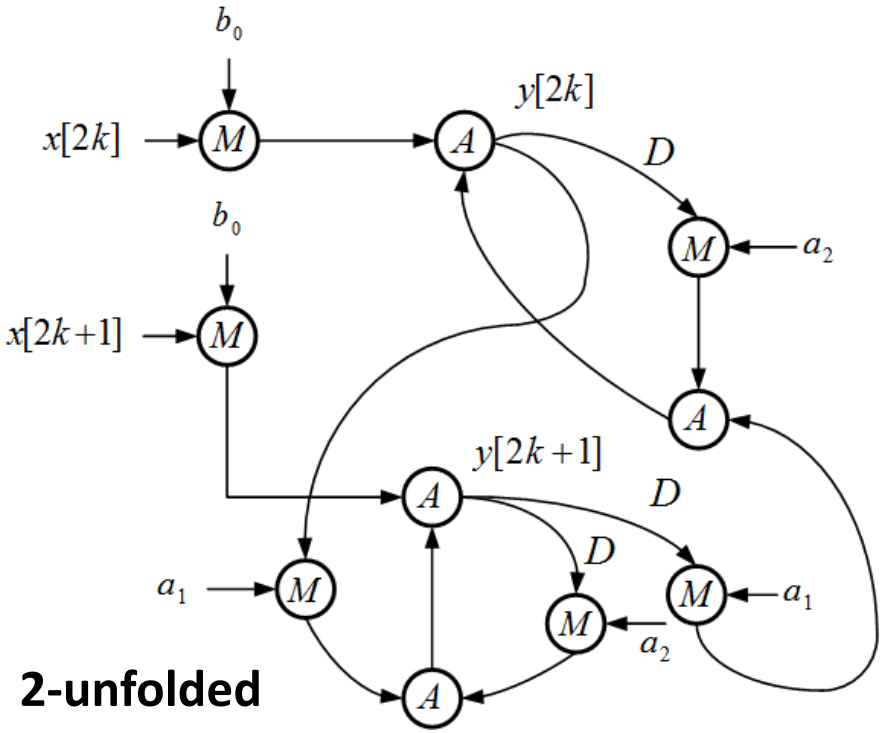
2-unfolded

- J −fold unfolding of a loop with N_D delays and N_n nodes results in $N_l = GCD(N_D, J)$ loops each with N_D/N_l delays and JN_n/N_l nodes
- J −fold unfolding increases T_{cp} by a factor of J

Unfolding Property 3



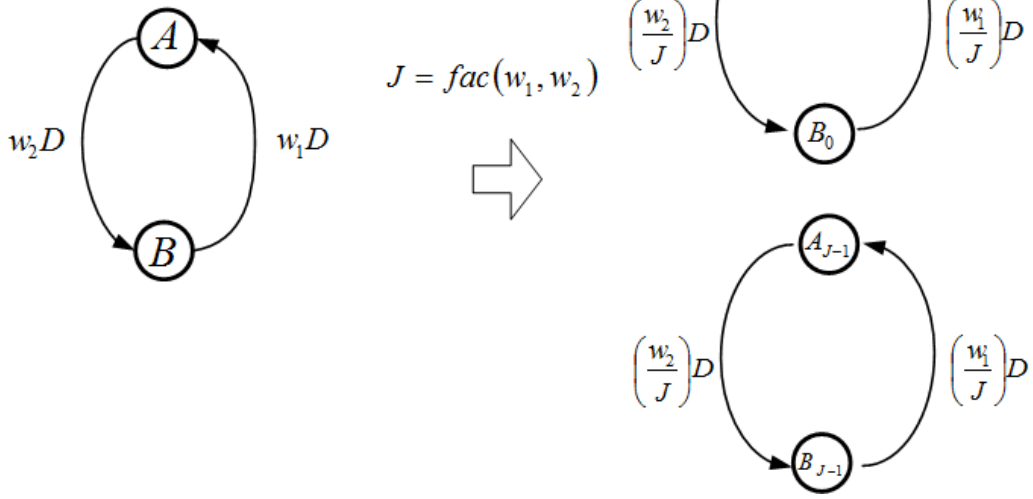
original



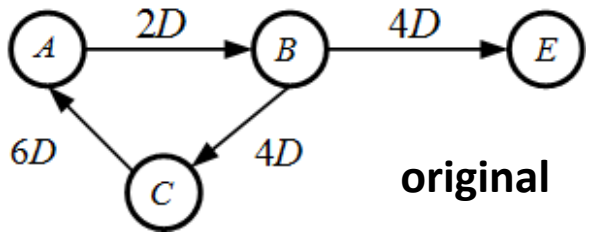
2-unfolded

- an arc/path P with weight $w(P) \geq J$ in the original DFG, will result in J paths with one or more delays in the J −fold unfolded DFG

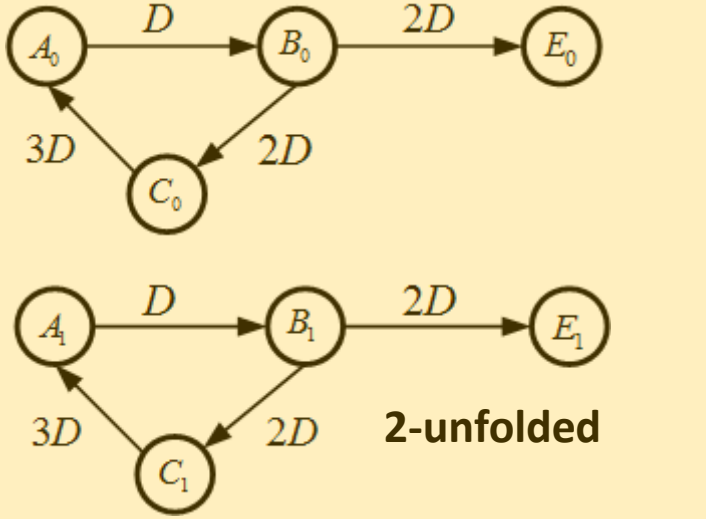
Unfolding Property 4



suitable for many-core implementation



original



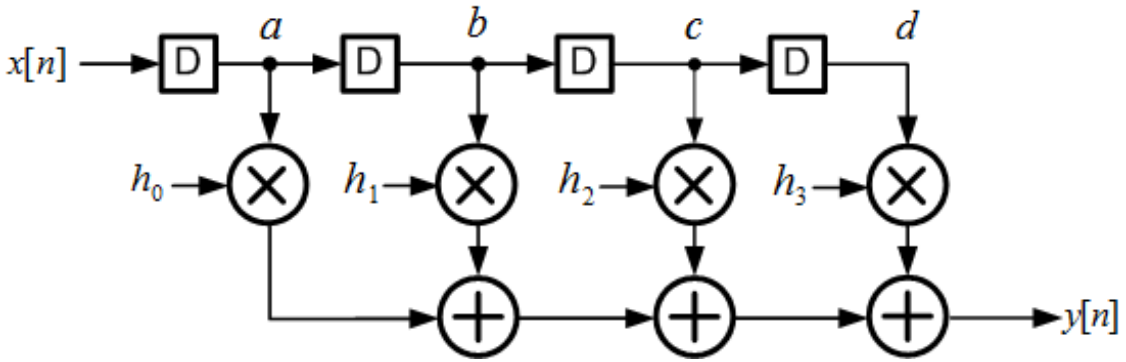
2-unfolded

- If *J is a common factor of all arc weights* in a DFG, then the *J* –fold unfolded DFG will have *J decoupled sub-DFGs* that are topologically identical to the original DFG with *all arc weights scaled down by J*

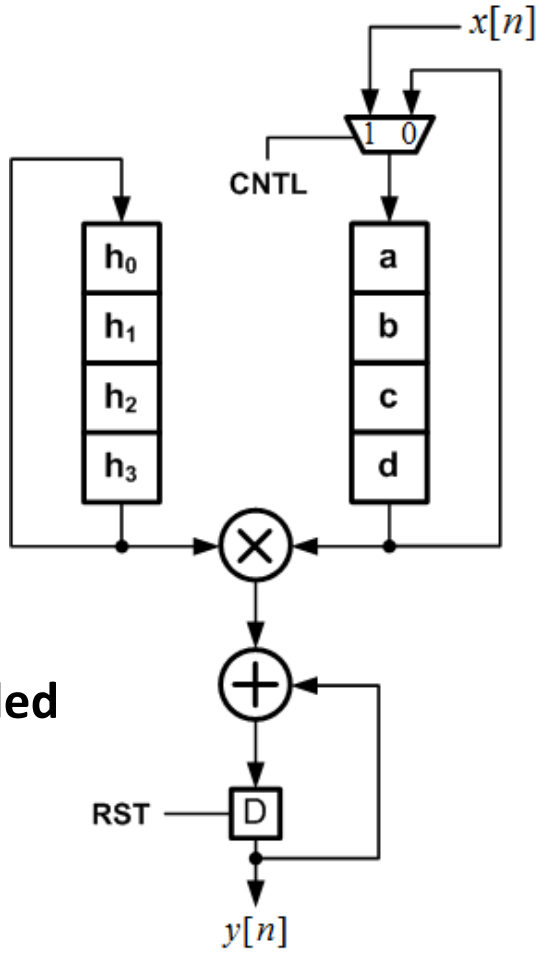
Folding

- Maps algorithmic operations to hardware
  - **Scheduling** algorithmic operations onto hardware cycles, and **binding** them to hardware units
  - Algorithmic DFG is different from hardware DFG (hDFG)
- Folding reduces area. It is a one-to-many mapping (unlike unfolding). Also referred to as multiplexing or shared-resource architecture.
- Folding by a factor J followed by unfolding by a factor J leads to a retimed version of original DFG

Folding a 4-tap FIR Filter



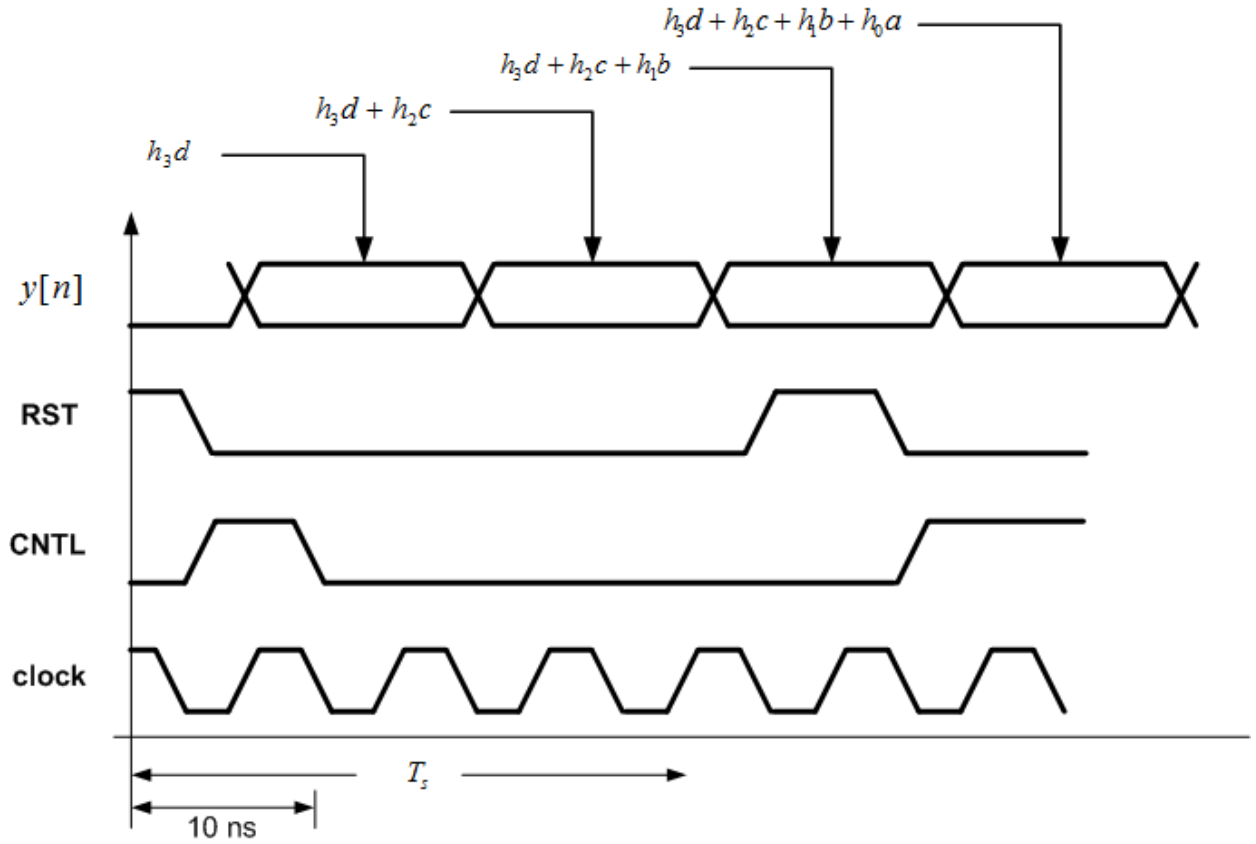
original



4-folded

- If $T_m = 7ns, Ta = 3ns, Ts = 40ns$, and $T_{cp} = 16ns$
- $J = 4$ folded architecture
  - 1 MAC unit, data, and coefficient registers

Folding a 4-tap FIR Filter: Timing Diagram



- $T_{cp} = 10ns$
- folding factor of 4 matches the hardware speed of the application assuming $T_s = 40ns$ and $T_{CLK} = 10ns$

References

- [anantha-JSSC92] A. Chandrakasan, et al., "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473-484, April 1992. [parhi-att] "Algorithm transformation techniques for concurrent processors," Proceedings of the IEEE, vol. 77, no. 12, pp. 1879-1895, December 1989.
- [anantha-CAD95] A. P. Chandrakasan et al., "Optimizing power using transformations," IEEE Transactions on CAD, vol. 14, no. 1, pp. 12-31, January 1995.
- [leiserson] C. Leiserson and J. Saxe, "Optimizing synchronous systems," Journal of VLSI and Computer Systems, vol. 1, no. 1, pp. 41-67, January 1983.
- [parhi-bip-sla] K. K. Parhi and D. G. Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters - Part I,II: Pipelining using scattered look-ahead and decomposition," *IEEE Transactions on Acoustics, Speech and Signal Processing,* vol. 38, no. 4, pp. 358-375, April 1991.
- [shanbhag-lms] N. R. Shanbhag and K. K. Parhi, "Relaxed look-ahead pipelined LMS adaptive filters and their application to ADPCM coder," *IEEE Transactions on Circuits and Systems,* vol. 40, no. 12, pp. 334-343, December 1993.
- [blaauw-JSSC12] D. Jeon et al., "A super-pipelined subthreshold energy-efficient 240 MS/s FFT core in 65nm CMOS," IEEE Journal of Solid-State Circuits, vol. 47, no. 1, pp. 23-34, January 2012.

Course Web Page

https://courses.grainger.illinois.edu/ece598nsg/fa2020/
https://courses.grainger.illinois.edu/ece498nsu/fa2020/

http://shanbhag.ece.uiuc.edu