ISA: Integrated Systems Architecture

Part 1

Algorithm to architecture mapping

Guido Masera, Politecnico di Torino, 2018-2019

Content

- Algorithm to architecture mapping
- Amdahl’s law
- DSP systems
- Digital filters
- Pipelining
- Parallel processing
- Low power architectures
- Iteration bound
- Retiming
- Unfolding
- Folding
- Universal techniques
- Decomposition
- Algebraic transformations
- Look-ahead transform
- Pipeline interleaving

Algorithm to Architecture Mapping

- Many applications require real-time processing with specific speed constraints

- *Data, audio and video (de)compression*
- *Ciphering & deciphering (primarily for secret key ciphers)*
- *Error correction coding*
- *Digital modulation & demodulation (for modems, wireless communication, and disk drives)*
- *Adaptive channel equalization for copper lines and optical bers*
- *Multipath combiners in broadband wireless access networks*
- *Computer graphics and video rendering*
- *Multimedia (e.g. MPEG, HDTV)*
- *Pattern recognition*

- We need to design HW Architectures for specified algorithm constraints (ASIC)
- Area-Speed-Power Tradeoffs
- Additional constraints: latency, flexibility

Algorithm to Architecture Mapping

Does it make sense to consider dedicated hardware architectures?

Dedicated architectures outperform program-controlled processors by orders of magnitude (wrt throughput and
energy eciency) in many transformatorial systems where data streams get processed in fairly regular ways.

but

Dedicated architectures can not rival the agility and economy of processor-type designs in applications where the computation is primarily reactive, very irregular, highly data-dependent, or memory-hungry.

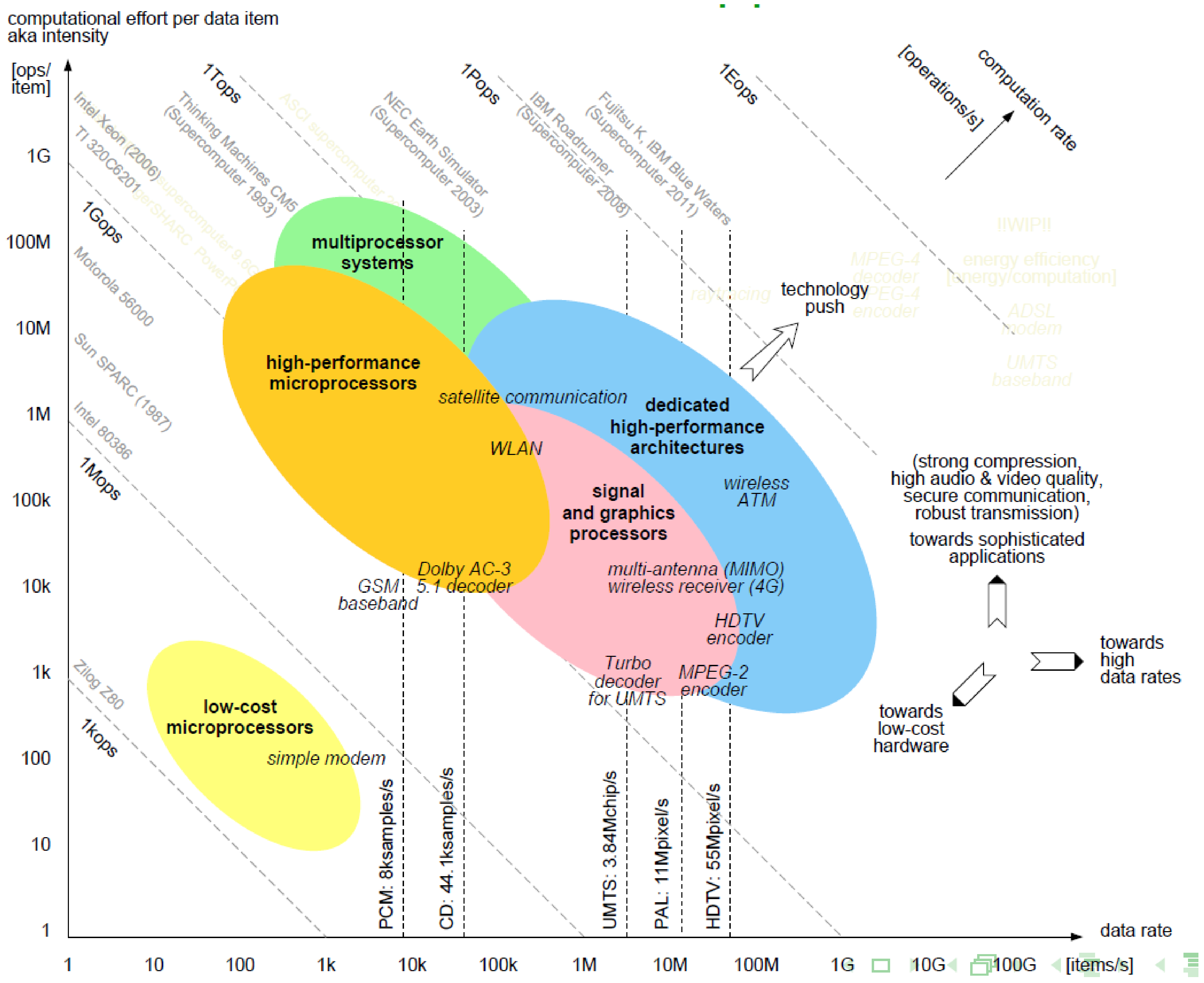
The architectural antipodes

	Hardware architecture	
	General purpose	Special purpose
Algorithm	any, not known a priori	fixed, must be known
Architecture	instruction set processor	dedicated, no single pattern
Execution model	fetch-load-execute-store “instruction-oriented”	process data item and pass on “dataflow-oriented”
Datapath	ALU(s) plus memory	customized design
Controller	with program microcode	typically hardwired
Performance indicator	instructions per second, run time of benchmarks	data throughput, can be anticipated analytically
Strengths	highly flexible, immediately available, routine design flow, low up-front costs	room for max. performance, highly energy-efficient, lean circuitry

Before embarking in ASIC design, find out

- Does an architecture dedicated to the application at hand make sense
- or is a program-controlled general-purpose processor more adequate?

Computational needs of various applications



From Hubert Kaeslin,
Microelectronics Design Center
ETH Zurich

Amdahl’s law

In making a design trade-off, favor the frequent case over the infrequent case

Example: Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st

Frequent case is often simpler and can be done faster than the infrequent case

Example: overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow

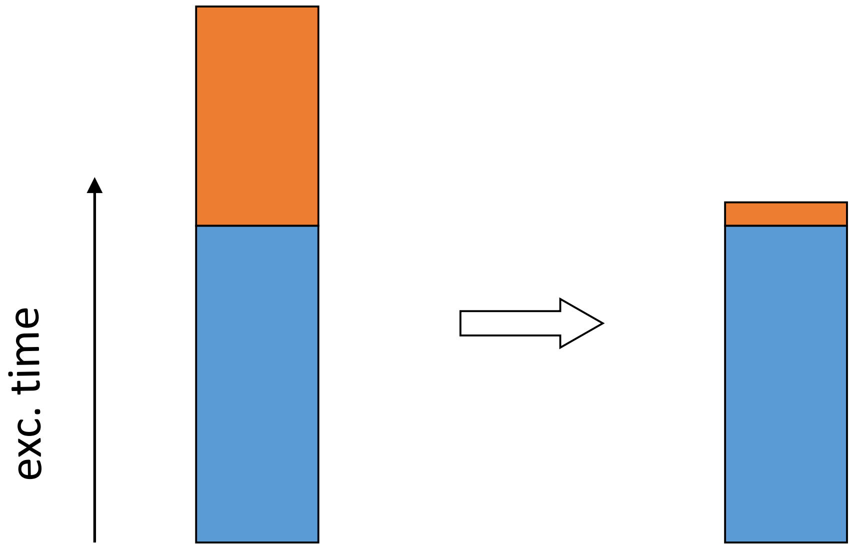
May slow down overflow, but overall performance improved by optimizing for the normal case

What is frequent case and how much performance improved by making case faster => Amdahl’s Law

Amdahl's Law

$$t_{new} = t_{old}\left[(1 - f_e) + \frac{f_e}{s_e}\right]$$

$$s_o = \frac{t_{old}}{t_{new}} = \frac{1}{\left[(1 - f_e) + \frac{f_e}{s_e}\right]}$$



- f_e: fraction of the enhanced architecture
- s_e: speedup of the enhanced architecture
- s_o: speedup of the overall architecture

Amdahl's Law: example

Floating point instructions improved to run 2 times faster, but only 10% of actual instructions are FP

$$t_{new} = t_{old}\left[(1-10\%)+\frac{10\%}{2}\right] = 0.95\ t_{old}$$

$$s_o = \frac{t_{old}}{t_{new}} = \frac{1}{0.95} = 1.053$$

Numeric signals and DSP systems

Numeric signal, *x(tn)*

- Discrete-time: $\{tn\}=\{n \cdot T;\ n=0, 1, 2, \ldots\}$
- Quantized: $x(tn) \in \{m \cdot q;\ m=0, 1, 2, \ldots\}$, q is the quantization step

Linear time-invariant system (LTI)

- Mapping $\mathcal{T}$: $x(n \cdot T) \to y(n \cdot T)$, with the following properties:
  - linear: $\sum_j a_j x_j \rightarrow \sum_j a_j y_j$
  - time-invariant: $x(n \cdot T-k \cdot T) \to y(n \cdot T-k \cdot T)$
- Causality implies that output at time $n \cdot T$ does not depend on future inputs: $y(n \cdot T)= \mathcal{T}[\ \{x(k \cdot T);\ k \leq n\}\ ]$

Numeric signals and DSP systems

The behavior of a causal LTI system can be described by means of a finite difference linear equation with constant coefficients

$$y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k)$$

If initial conditions, *y(-1), y(-2), …* and *x(-1), x(-2), …* are null, the answer to inputs *x(n)* for *n≥0* is given as

$$y(n) = x(n) * h(n) = \sum_{k=0}^{n} x(k)h(n-k) = \sum_{k=0}^{n} h(k)x(n-k)$$

Where *h(n)* if the impulse response, and * indicates the convolution

IIR and FIR

IIR (Infinite Impulse Response):

- $h(n)$ has infinite length ($N>0$ and $a_k \neq 0$ for at least one k value)
- $h(n)$ is found by applying $x(n)=\delta(n)$

  $h(0)=b_0,\ h(1)=a_1 h(0)+b_1,\ \ldots,\ h(M)=a_1 h(M-1)+\ldots+a_N h(M-N)+b_M, \ldots$

FIR(Finite Impulse Response):

- $h(n)$ has finite length ($N=0$, non recursive equation)
- $h(n)=b_n$ with $0 \leq n < M$ (and $h(n)=0$ otherwise)

z transform

$$X(z) = \sum_n x(n) z^{-n}$$

equivalent to the Laplace transform for continuous signals

By setting $|z^{-1}| = 1$ (unit circle), we get the Fourier transform of the discrete-time signal

$$z = e^{-j\theta}, \theta = 2\pi f T \rightarrow X(\omega)\text{=}\sum_k x(k) e^{-j\omega k T}$$

Properties of the z transform

On the unit circle, it is linear with respect to *f* (period *1/T*)

Linear

Time shift corresponds to the increment of the z exponent

$$x(n-m) \ \rightarrow X(z)z^{-m}$$

Convolution is mapped to a product

$$y(n) = \sum_{k=0}^{n} x(k)h(n-k) \rightarrow Y(z) = \sum_{n=0}^{\infty}(x*h)\, z^{-n} = X(z)H(z)$$

The finite difference equation becomes

$$Y(z) = \sum_{k=1}^{N} a_k Y(z)\ z^{-k} + \sum_{k=0}^{M} b_k X(z)\, z^{-k}$$

z transform of the system response

The ratio *Y(z)/X(z)* is the z transform of the impulse response *h(n)*

$$\frac{Y(z)}{X(z)} = H(z) = \frac{\sum_{k=0}^{M} b_k \, z^{-k}}{1 - \sum_{k=1}^{N} a_k \, z^{-k}}$$

$$\frac{Y(z)}{X(z)} = H(z) = b_0 \frac{\prod_{k=1}^{M}\left(1 - r_k z^{-k}\right)}{\prod_{k=1}^{N}(1 - p_k z^{-k})}$$

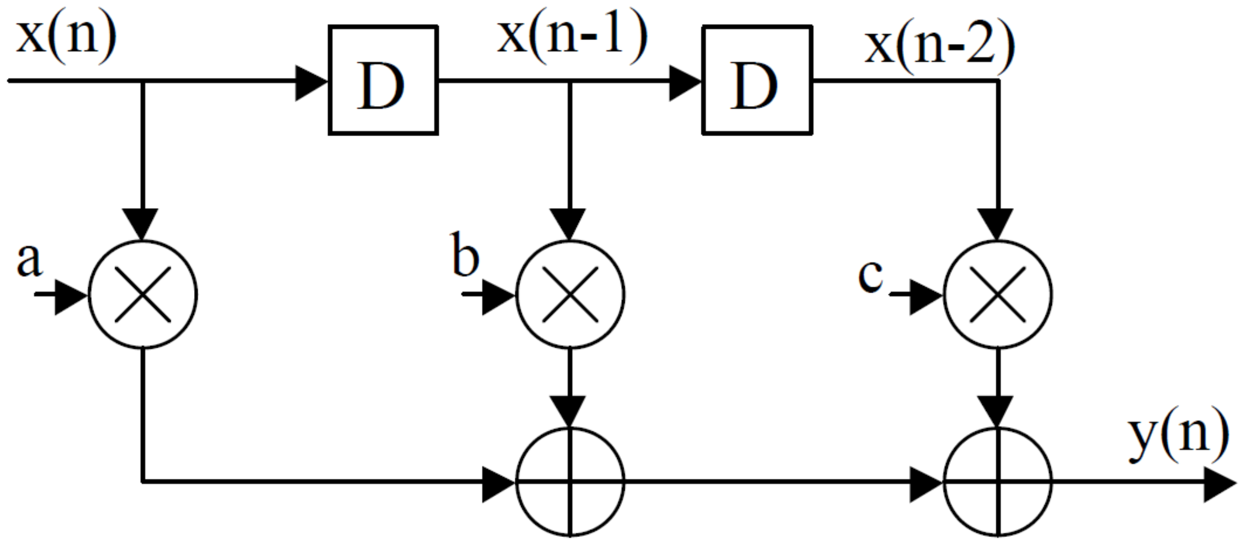
Where r_k are the roots of the numerator (zeros) and p_k and the roots of the denominator (poles)

Representation of DSP systems

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

<u>Graphical Representation Method 1: Block Diagram</u>

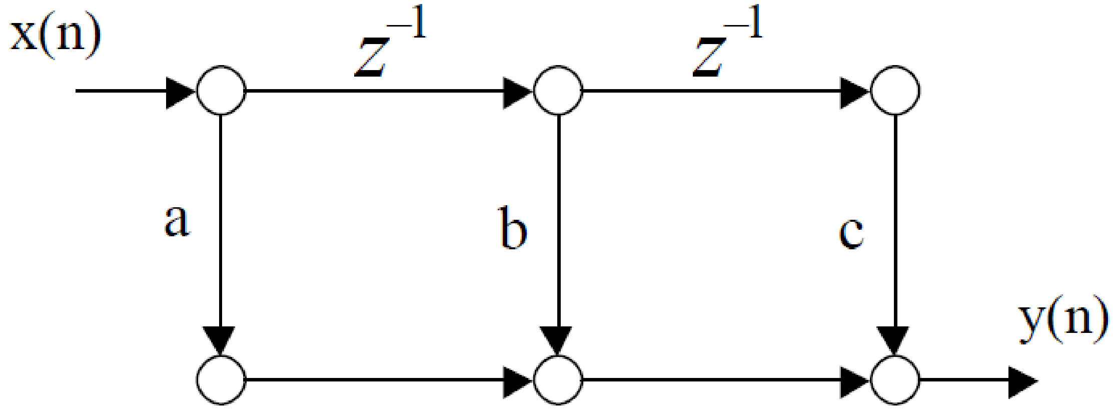
- functional blocks connected with directed edges,
- data flow from input to output



Representation of DSP systems

<u>Graphical Representation Method 2: Signal-Flow Graph (SFG)</u>

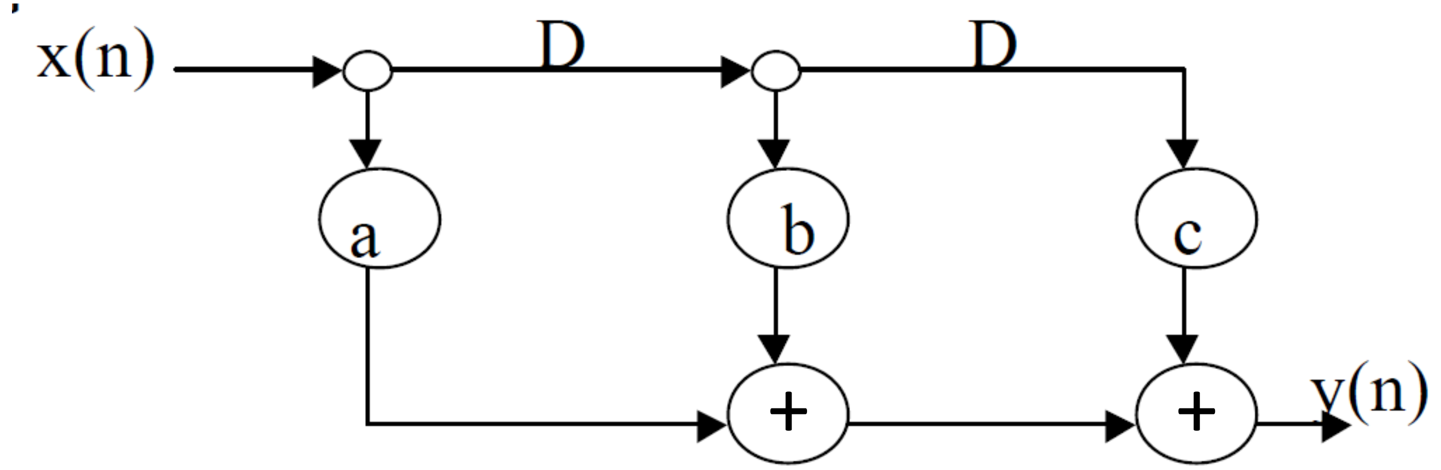
- Nodes sum all incoming signals
- Directed edge (j, k) denotes a linear transformation from the input signal at node j to the output signal at node k
- Usually used for linear time-invariant DSP systems representation
- Linear SFGs support transformations like graph reversal or transposition



Representation of DSP systems

<u>Graphical Representation Method 3: Data-Flow Graph (DFG)</u>

- Nodes represent computations (or functions or subtasks)
- Directed edges represent data paths (data communications between nodes)
- Each edge has a nonnegative number of delays associated with it



Representation of DSP systems

A 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

Computational structures

The *H(z)* is usually derived from a set of input specifications, mainly related to the modulus and phase behavior

Given a certain *H(z)*, we can find lots of possible implementations (computational structures) that are fully equivalent in terms of I/O behaviour, as long as all operations use infinite-precision arithmetic

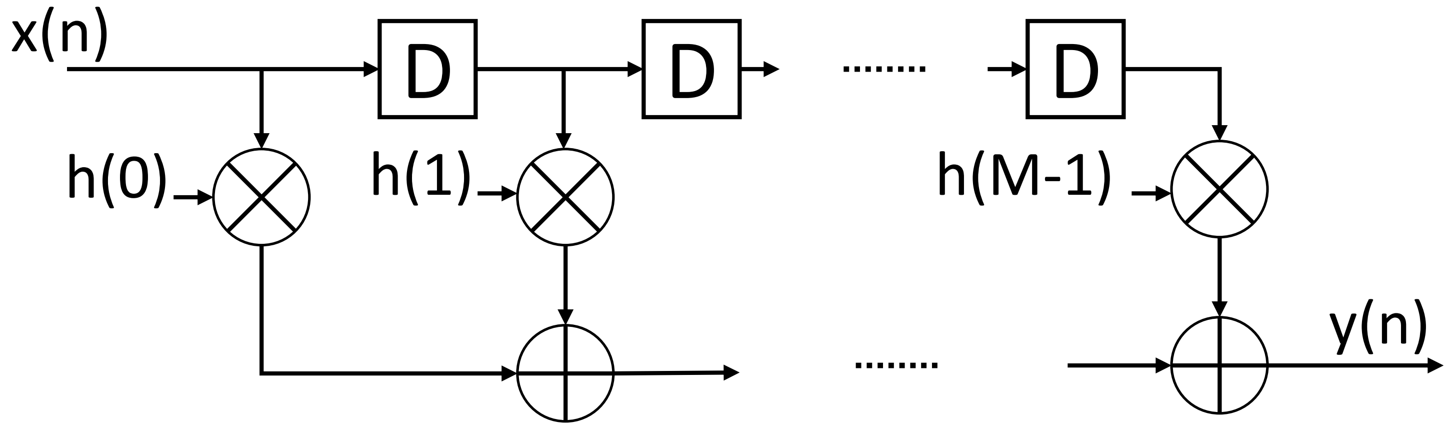
These computational structures may show differences in terms of complexity and/or processing speed

The adoption of finite-precision arithmetic introduces quantization errors and thus differences among alternative structures

Computational structures: FIR case

Direct form:

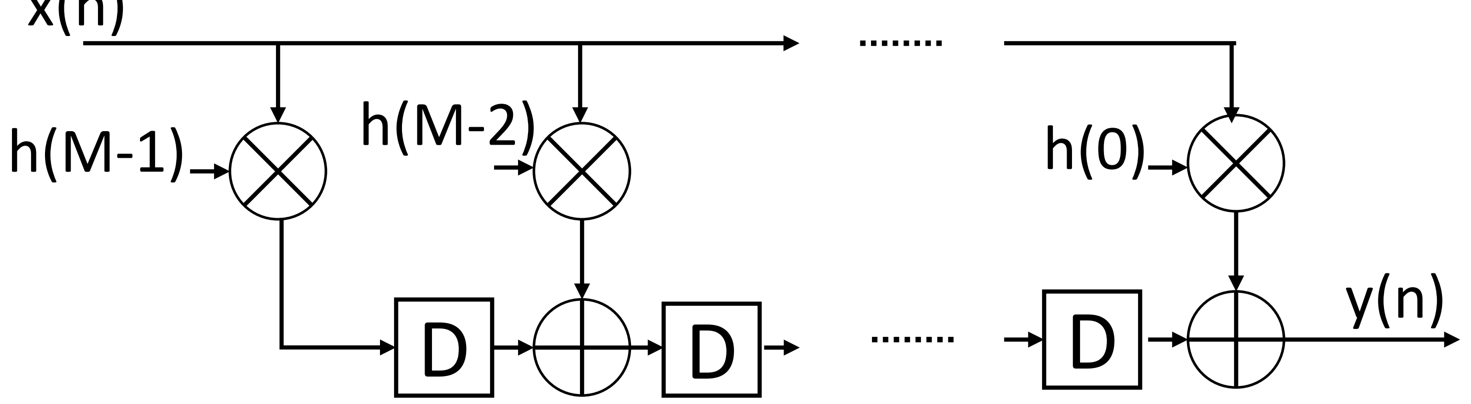
$$t_{cp} = t_{CKQ} + t_M + (M-1)t_A + t_{su}$$



Transposed form

(or data broadcast structure):

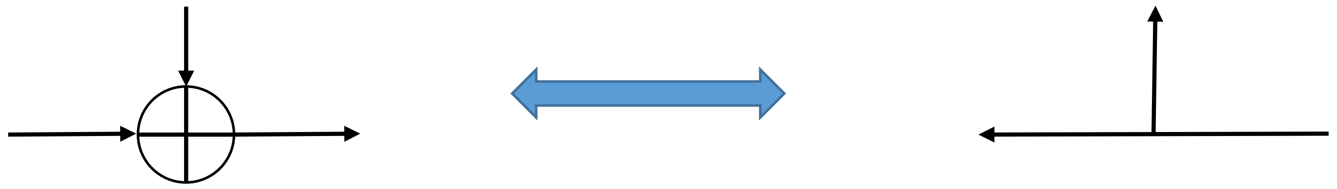
$$t_{cp} = t_{CKQ} + t_M + t_A + t_{su}$$



Transposition form

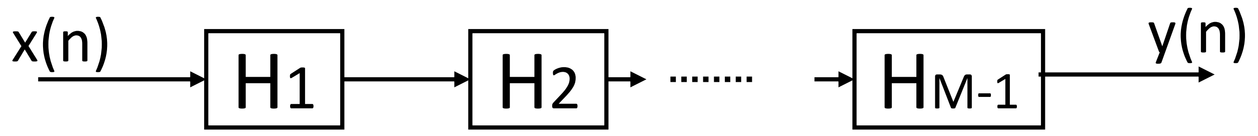
Any SFG *G* can be transformed into the equivalent transposed form *Gt* by means of a few simple steps:

1. For each arc in *G*, invert the direction
2. Exchange input and output ports
3. For each arc in *G*, keep the same gain (multiplication factor or z^{-1}) in *Gt*
4. Transform sum nodes into fork nodes and viceversa:



Cascade and additive forms (FIR case)

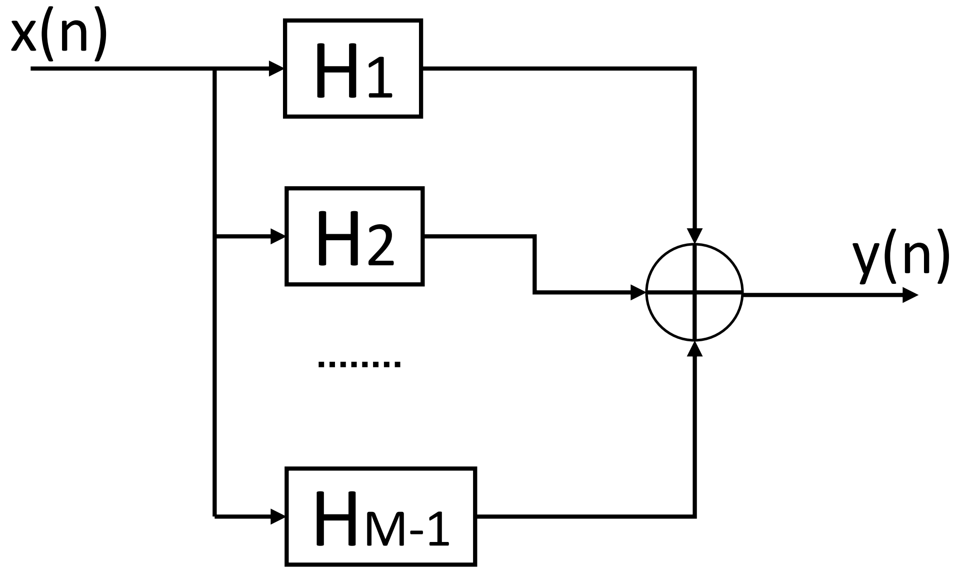
Cascade form:



$$H(z) = G_0 \cdot H_1(z) \cdot H_2(z) \cdot \cdots H_{M-1}(z)$$

$$H_i(z) = \left(1 - \frac{r_i}{z}\right)$$

Additive form:



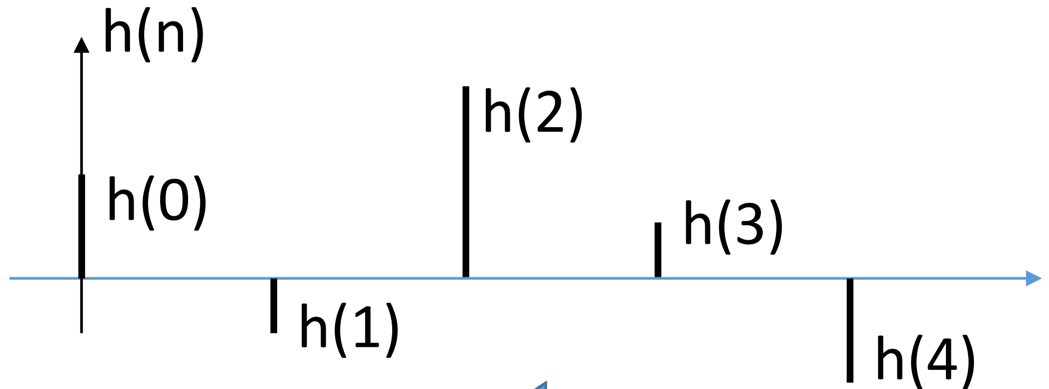
$$H(z) = H_1(z) + H_2(z) \cdots + H_{M-1}\,(z)$$

Linear phase in FIR filters

Usually, we desire an *H(ω)* with

- constant modulus,
- linear phase

in pass band



The condition

$$\mathrm{H}(\omega) = e^{-j\alpha\omega}\widetilde{H}(\omega) \qquad \tan^{-1}\frac{Im\{H(\omega)\}}{Re\{H(\omega)\}} = -\alpha\omega + c$$

implies

- either $Im\{\widetilde{H}(\omega)\}$=0, achieved if *h(n)* has even simmetry
- or *Re*$\{\widetilde{H}(\omega)\}$=0, achieved if *h(n)* has odd simmetry

Linear phase in FIR filters

Given the impulse response $h(0), h(1), \dots h(M-1)$, we indicate as S the position of the central element:

- with odd M, $S=(M-1)/2$
- with even M, S is between $M/2-1$ and $M/2$

$h(n)$ shows even symmetry if

- $h(M/2+k) = h(M/2-1-k)$ $\quad k = 0, \dots, M/2-1$ $\quad$ with M even
- $h(S+k) = h(S-k)$ $\quad k = 0, \dots, S$ $\quad$ with M odd

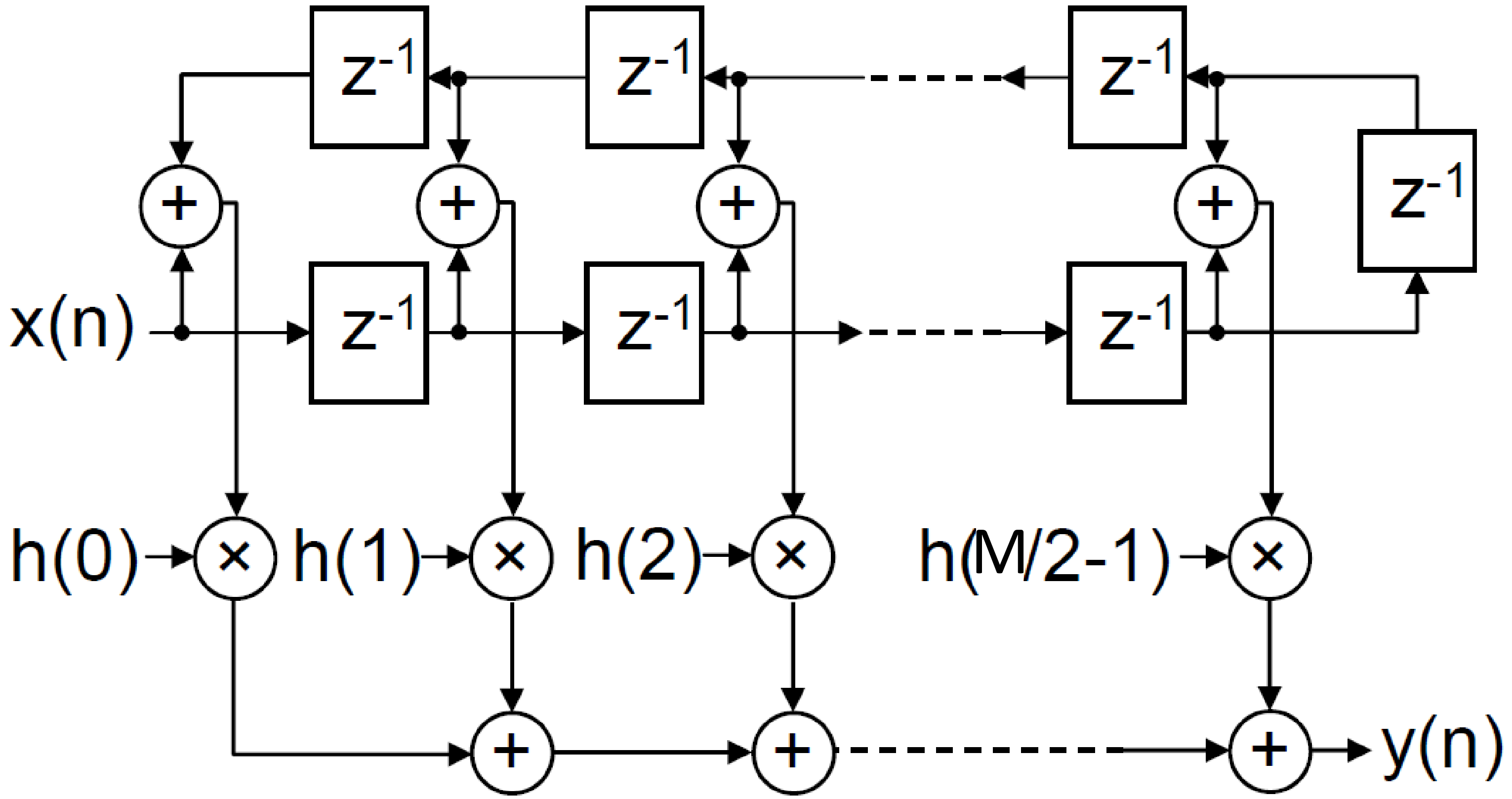
$h(n)$ shows odd symmetry if

- $h(M/2+k) = -h(M/2-1-k)$ $\quad k = 0, \dots, M/2-1$ $\quad$ with M even
- $h(S+k) = -h(S-k)$ $\quad k = 0, \dots, S$ $\quad$ with M odd

Linear phase in FIR filters

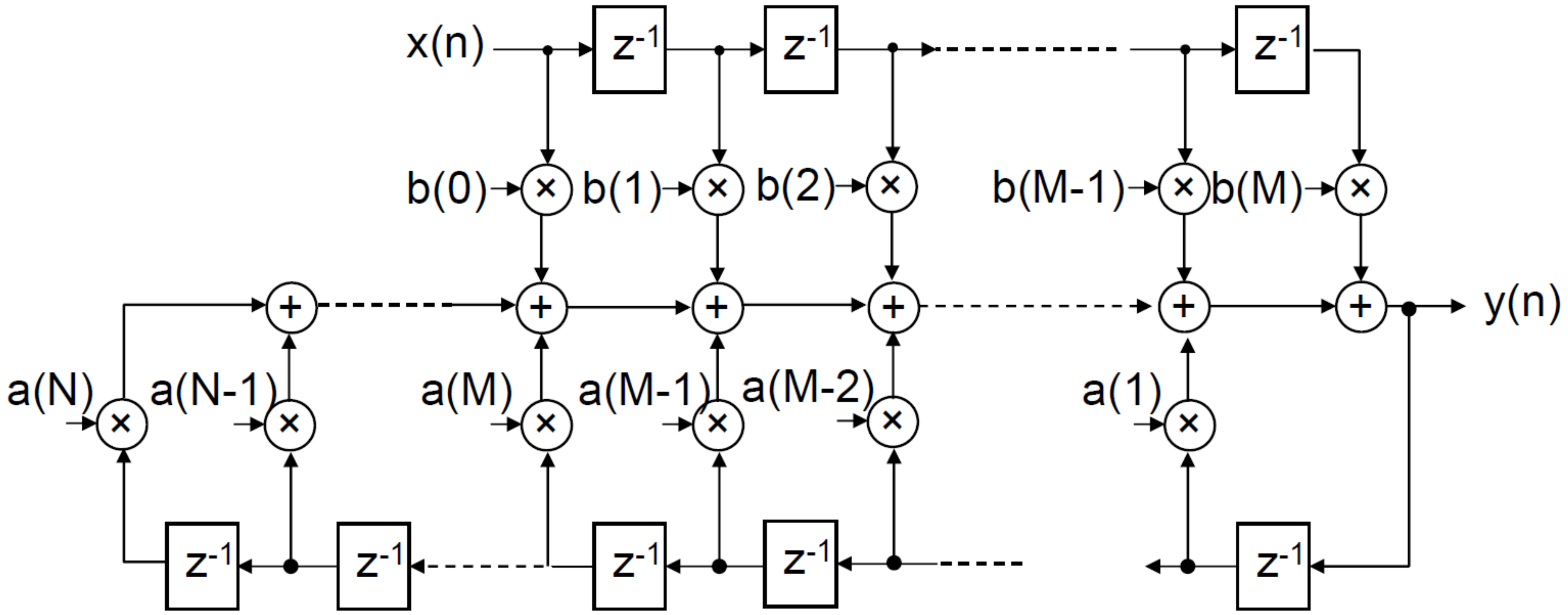
The even or odd symmetry in the impulse response *h(n)* can be exploited to reduce the number of multipliers in the direct form

Example (even symmetry, M even):



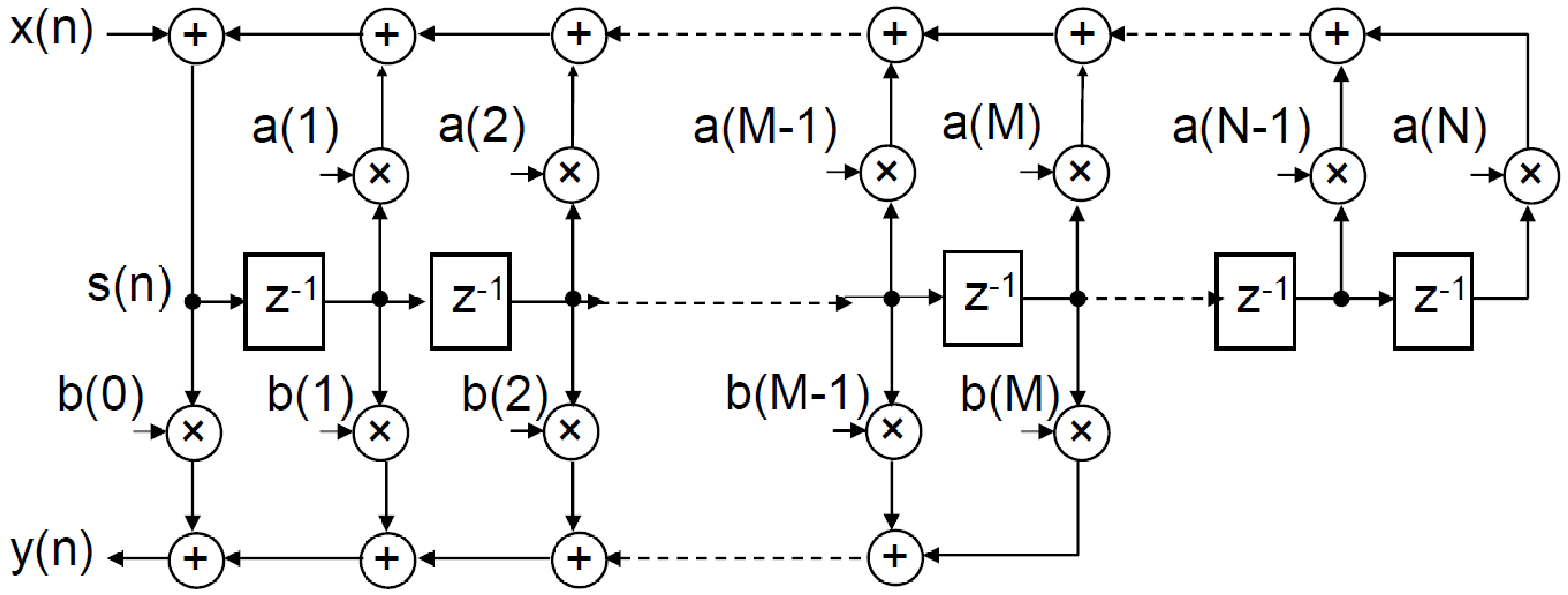
Computational structures: IIR case

Direct form I: $y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k)$



Computational structures: IIR case

Direct form II: $y(n) = \sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k)$



Computational structures: IIR case

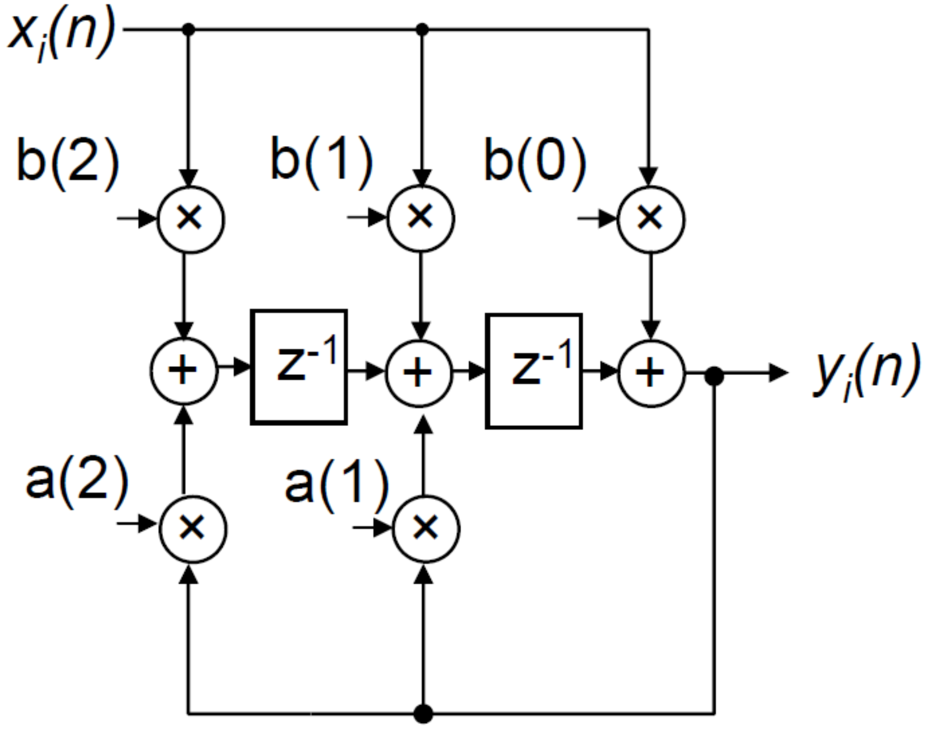
Transposed forms can be obtained for direct forms I and II

Cascade forms:

$$H(z) = \frac{\sum_{k=0}^{M} b_k \, z^{-k}}{1 - \sum_{k=1}^{N} a_k \, z^{-k}}$$

$$H(z) = \prod_{i=1}^{Q} \frac{b_{i,0} + b_{i,1} z^{-1} + b_{i,2} z^{-2}}{1 + a_{i,1} z^{-1} + a_{i,2} z^{-2}}$$

Additive forms …



Computational structures

In the practical implementation of a computational structures, the quantization of coefficients tend to modify the ideal position of zeros and poles of *H(z)*.

This affects the behaviour of the system and its stability too. The effect of finite precision arithmetic is weaker for systems with disperse zeros/poles than in the case of grouped zeros/poles.

In the FIR case, zeros tend to be disperse and therefore the direct form is the most used computational structure

In the IIR case, zeros and poles are grouped and the sensitivity to quantization is high: cascade forms are then preferred, as they allow for the independent tuning of each biquad stage.

Figures of merit for hardware architectures

- Cycles per data item *Cdi*, number of computation cycles between releasing two subsequent data items (e.g. samples)
- Longest path delay, or critical path delay, *tcp* , the lapse of time required for data to propagate along the longest path. A circuit cannot function correctly unless $tcp < Tck$
- Time per data item *T* , the lapse of time between releasing two subsequent data items (e.g. in ns/sample, ms/frame, or s/computation): $T = Cdi * Tck > Cdi * tcp$
- Data throughput $th = 1/T$ (expressed in pixel/s, sample/s, frame/s…)
- Latency *L* , number of computation cycles from a data item entering a circuit until the pertaining result becomes available

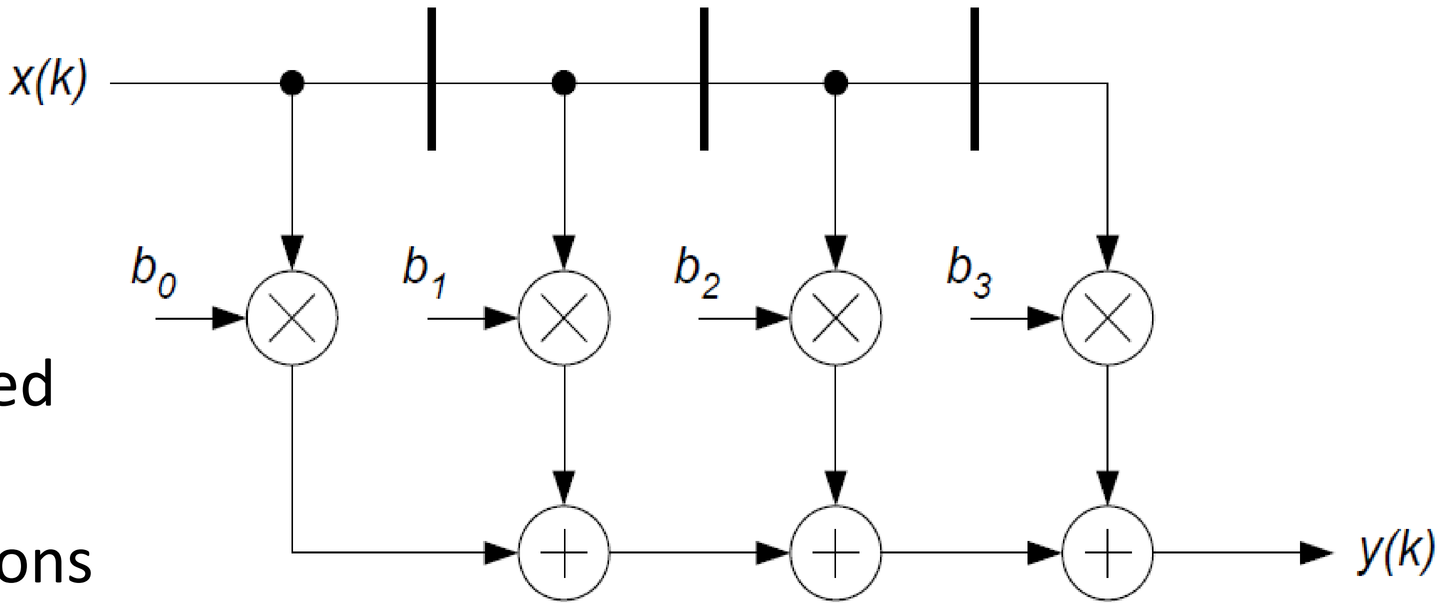
Figures of merit for hardware architectures

- Circuit size or complexity *A*, expressed in mm2, GE (gate equivalent) or number of elementary components
- Size-time product *AT* , the hardware resources spent to obtain a given throughput: *AT = A / th* (we also use the throuput to area ratio, TAR)
- Energy per data item *E* , the amount of energy dissipated for a given computation on a data item (e.g. in pJ/MAC, nJ/sample, or in mWs/frame). Can also be understood as power-per-throughput ratio *E = P / th* (measured in mW/Mbits or W/GOPS)

  Energy/data item = energy per second/data item per second = power/throughput

- Energy-time product *ET*: indicates how much energy gets spent for achieving a given throughput (synonym “energy-per-throughput ratio"): *ET = E/th = P/th^2* (e.g. in nJ/datablock/s or mWs2/videoframe)

Example



Approximations

✓Interconnect delays neglected (overly optimistic)

✓Delays of arithmetic operations summed up (sometimes pessimistic)

✓Glitching ignored (optimistic)

- A = 3Areg + 4Am + 3Aa
- Cdi = 1
- tcp = treg + tm + 3ta

- AT = (3Areg + 4Am + 3Aa)(treg + tm + 3ta)
- L = 0
- E = 3Ereg + 4Em + 3Ea

Pipelining

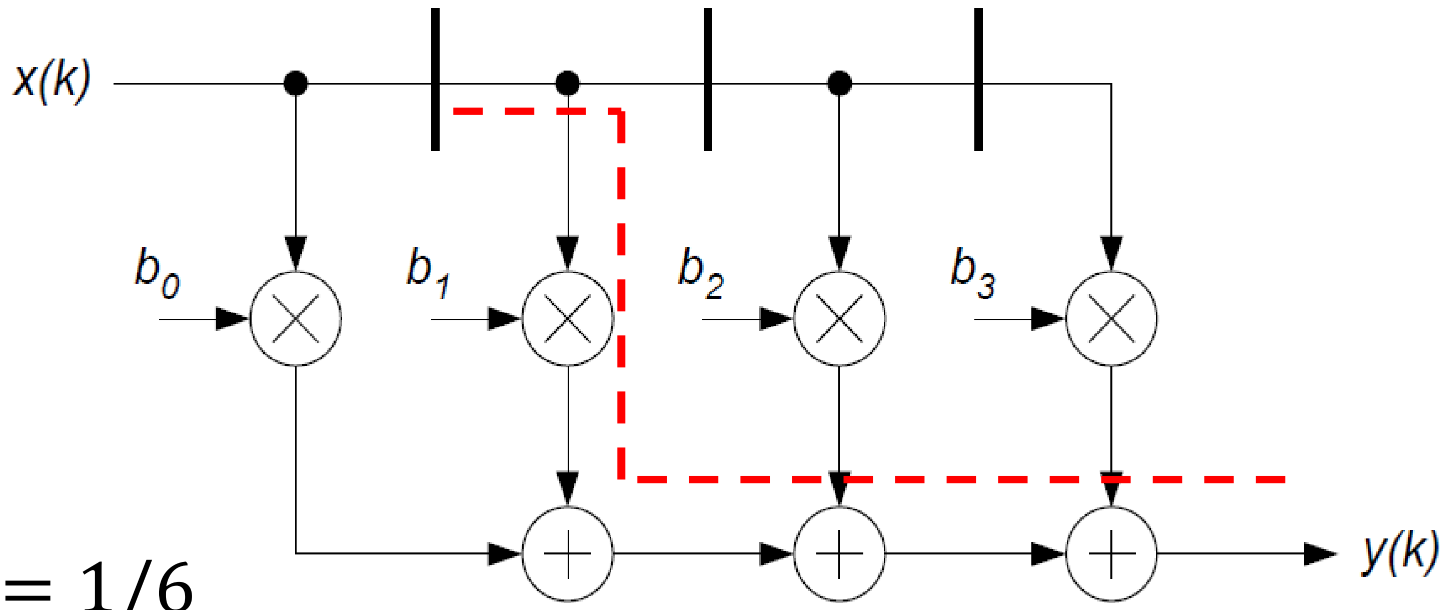
- Comes from the idea of a water pipe: continue sending water without waiting the water in the pipe to be out
- Leads to a reduction in the critical path by increasing the clock speed

Isomorphic implementation:

critical path - - - - - -

tcp = Tm+3*Ta = 3+3*1=6 u.t.

Tck>Tm+3*Ta th<= $\frac{1}{3Ta+Tm} = 1/6$

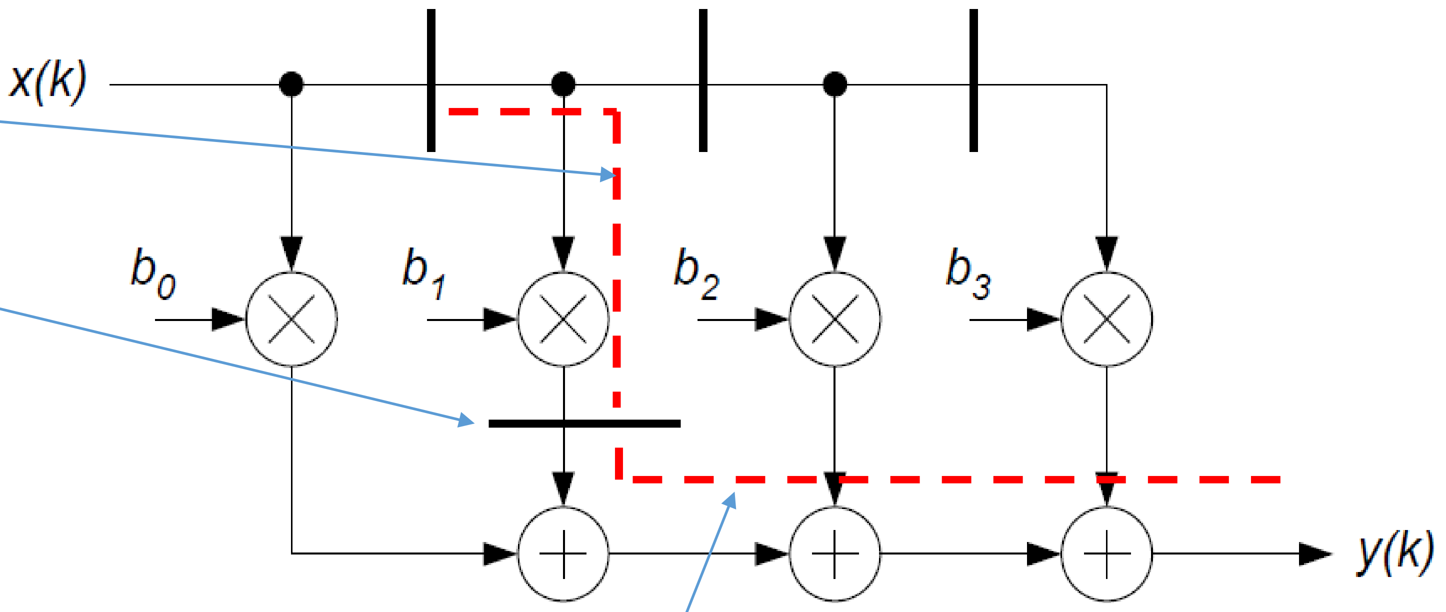


Pipelining

Segment 1 (delay=Tm=3 u.t.)

Pipelining register

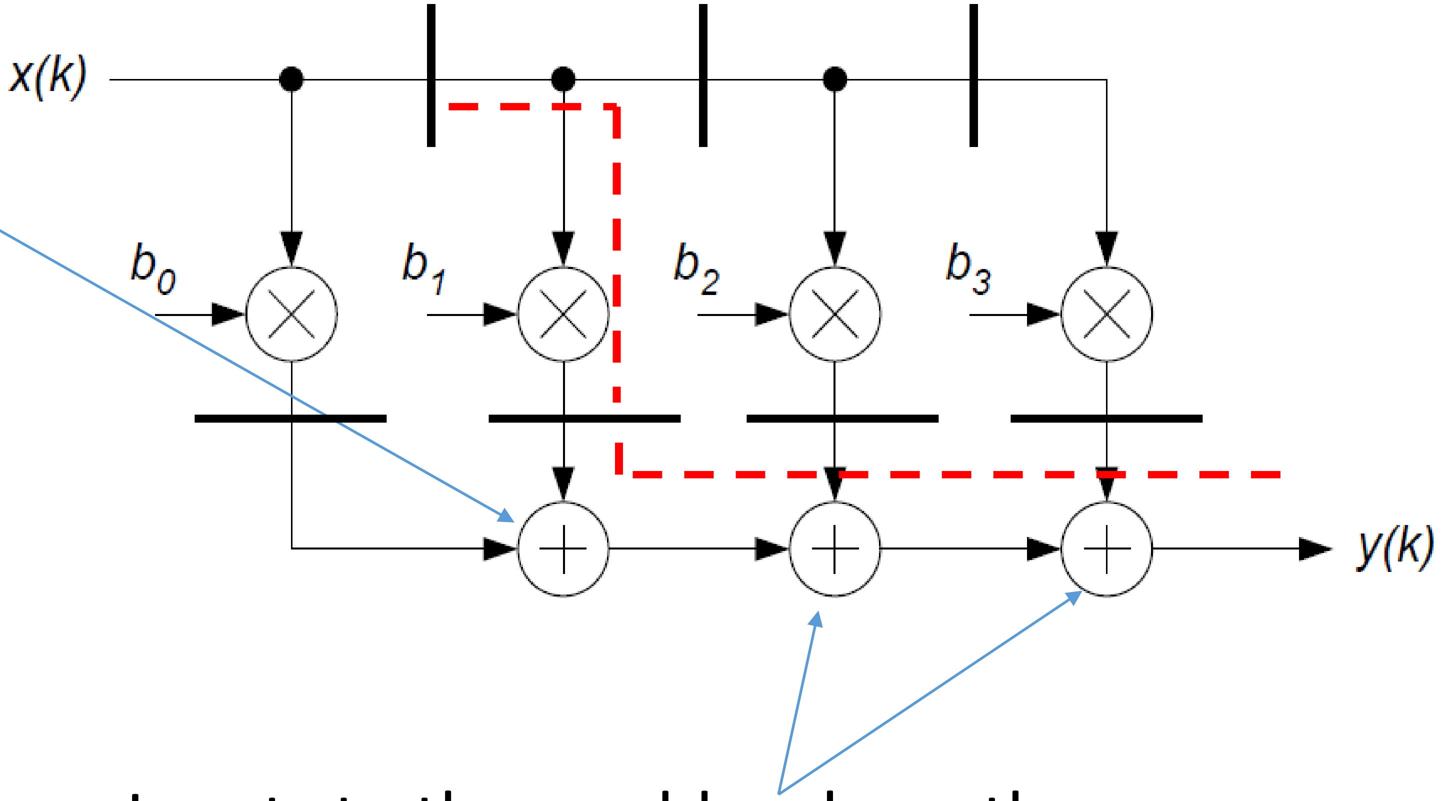
However, more pipelining registers are needed to have synchronous inputs to each graph node!



Segment 2 (delay=3Ta=3 u.t.)

Pipelining

Inputs to this adder arrive after the same number of cycles as in the original DFG



tcp = Tm=3*Ta = 3 u.t.

Tck>Tm th<= $\frac{1}{Tm} = 1/3$

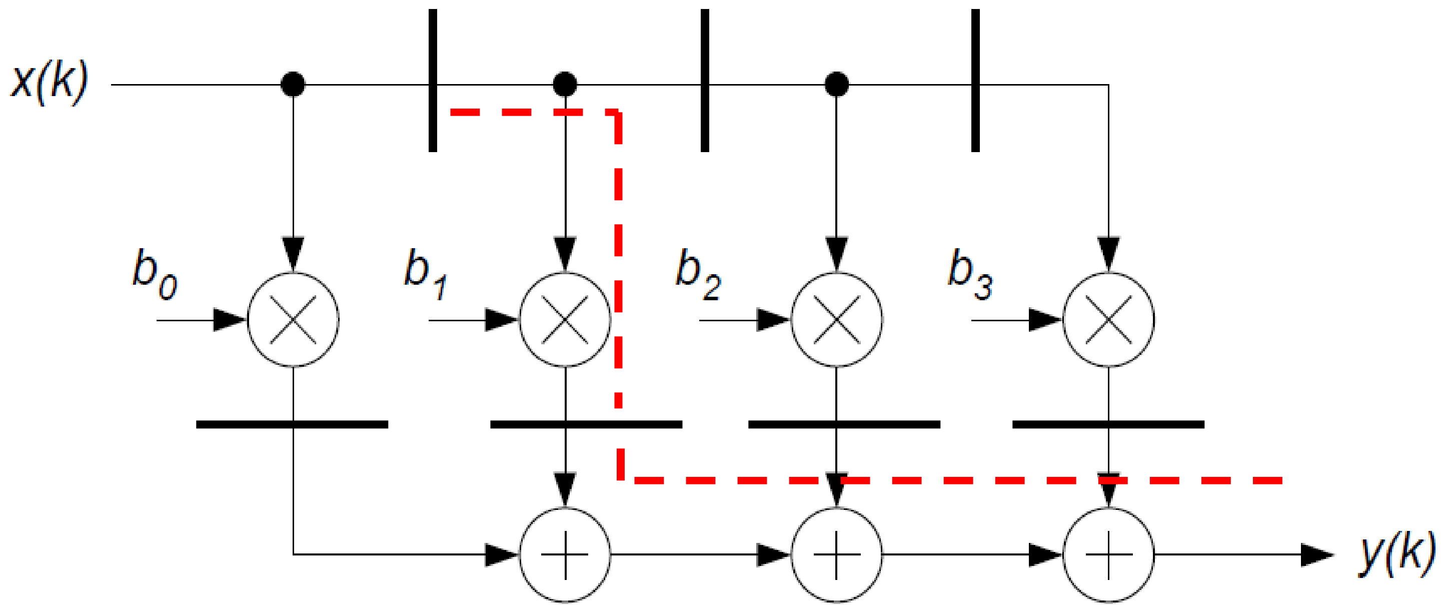
Inputs to these adders have the same relative delay as in the original DFG

Pipelining: formal method

Formal method to facilitate the suitable placing of pipelining latches:

The pipelining registers can only be placed across any feed-forward cutset of the graph

- Cutset: a set of edges of a graph such that if these edges are removed from the graph, the graph becomes disjoint
- Feedforward cutset: a cutset where all edges have the same direction



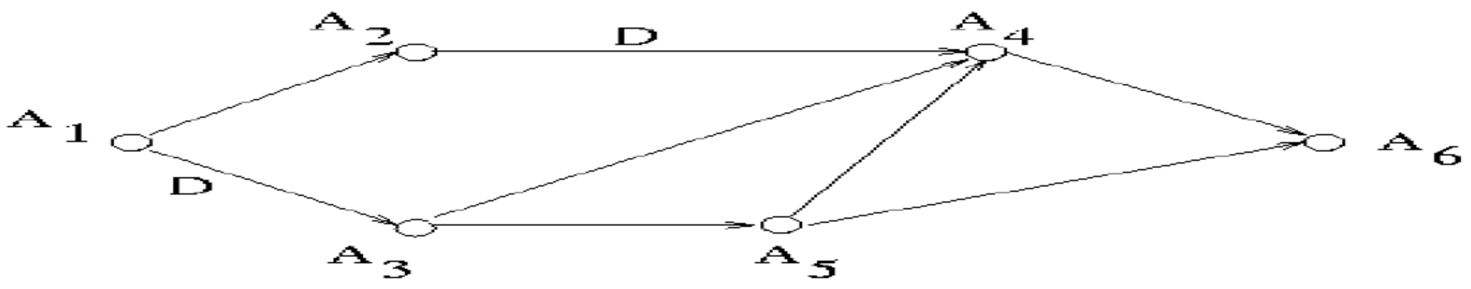
Pipelining: formal method

Example:

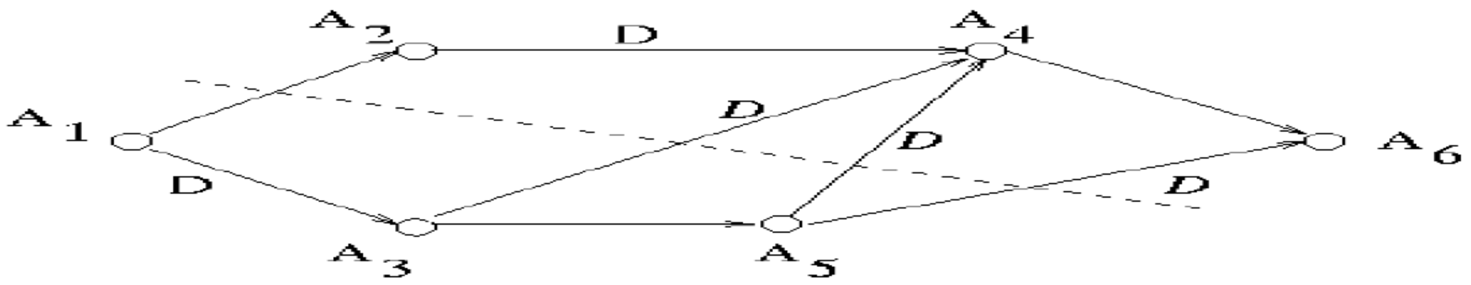
(a) Original DFG

(b) Dashed line -> feedforward cutset and additional registers

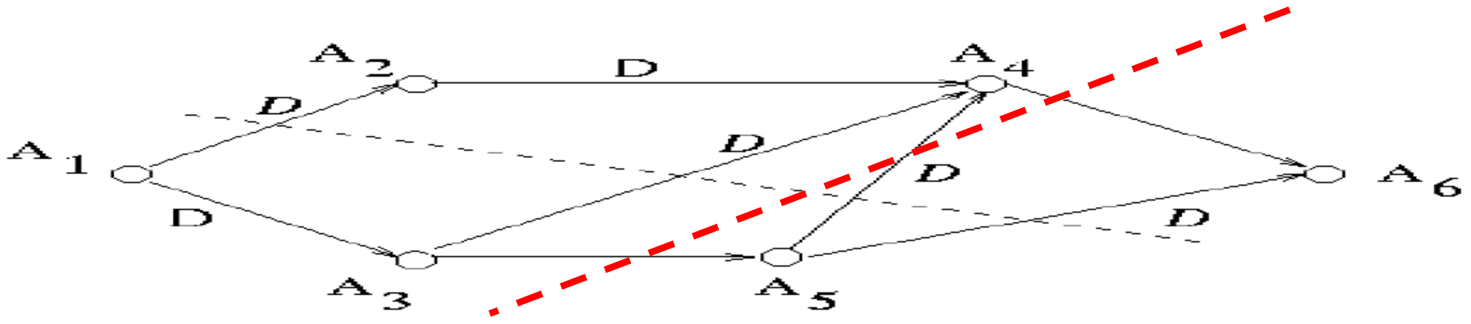
(c) Red dashed line -> cutset (not a feedforward cutset)



(a)



(b)

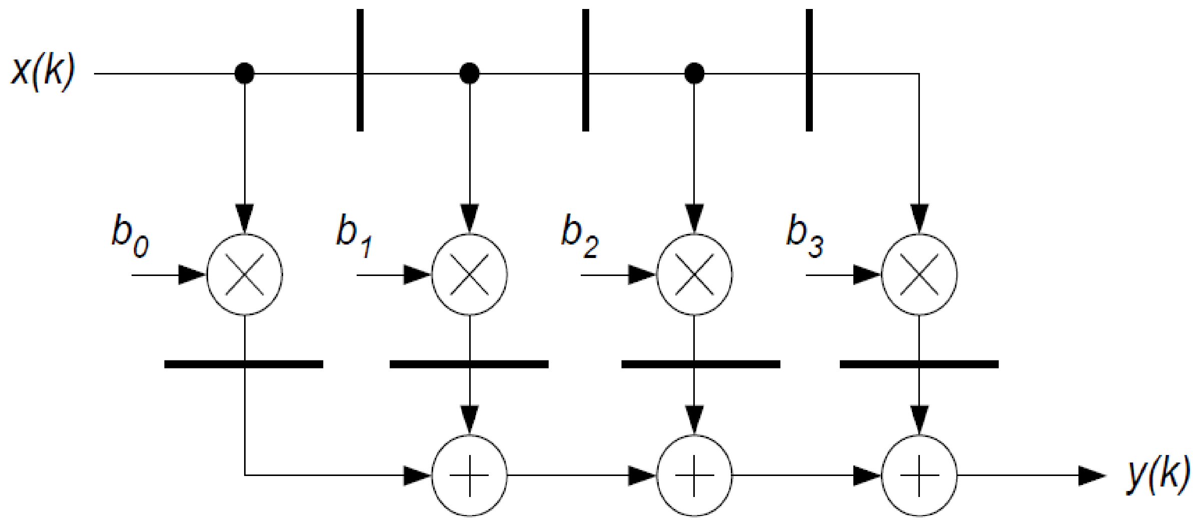


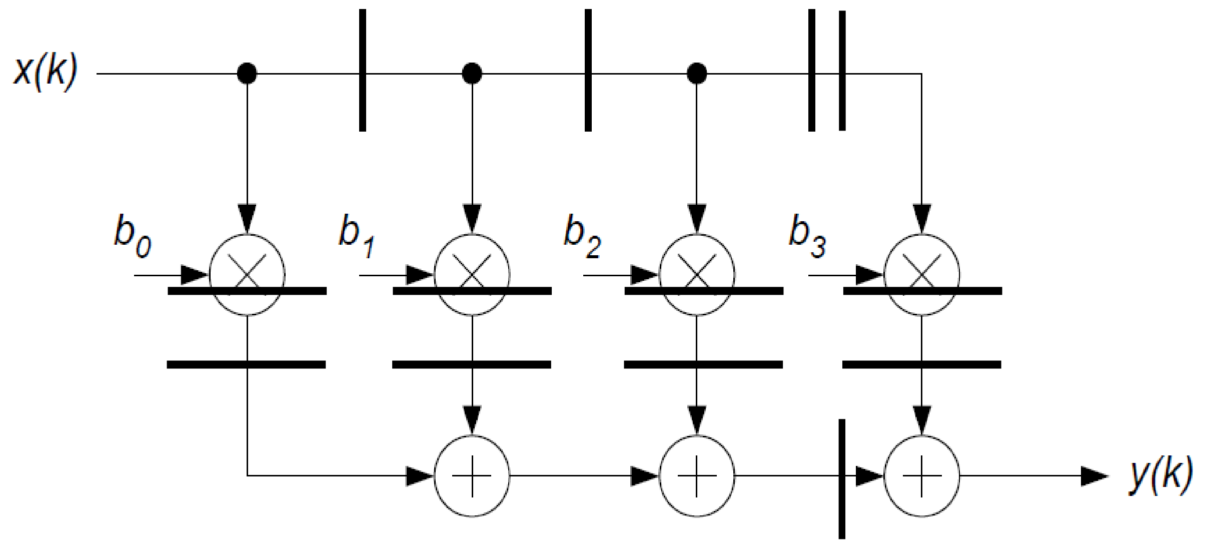
(c)

Fine grain pipelining

Sometimes, pipelining registers can be inserted within processing nodes, to further increase the clock speed.

Example: by breaking the multiplier into 2 smaller units with processing times of 2 and 1 u.t., respectively, the clock period can be reduced from 3 to 2 u.t.





Limits of pipelining

Ideally, *p* levels of pipelining provide a *p*-fold throughput increasing at the cost of little extra complexity (only pipelining registers)

AT product decreases dramatically

However, large *p* (e.g. fine grain pipelining) implies diminished speed-up, *AT* increase and excessive power dissipation

Theoretical bound:

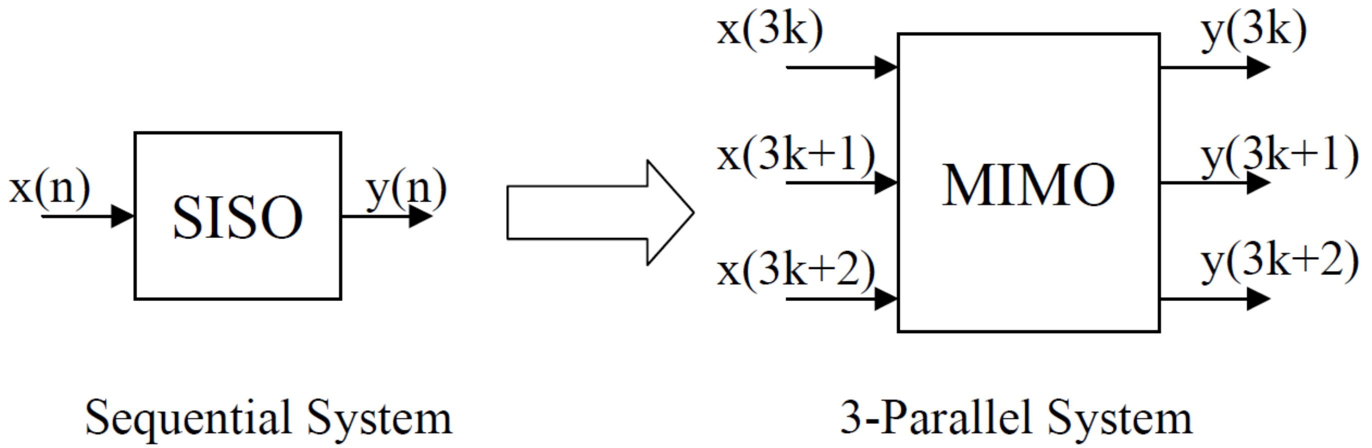
$$T_{ck} \geq t_{cp} = \min(t_{comb}) + t_{reg} = t_{nand2} + t_{su} + t_{CKQ}$$

12 or so FO4 inverter delays per stage is close to practical limit

Parallel processing

A brute force approach to performance: allocate multiple hardware resources (processing nodes) working concurrently on more data

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

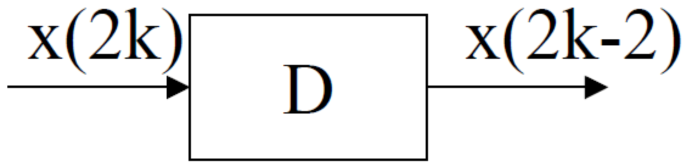


A MIMO system with degree of parallelism *P* receives *P* samples per cycle (at times *Pk, Pk+1, … Pk+P-1*) and generates *P* sample per cycle

Parallel processing

In a *P*-parallel DFG, any register introduces a delay of *P* cycles.

Example:



Given a DFG, the *P*-parallel version is not obtained by simply replicating the original DFG!

Two approaches:

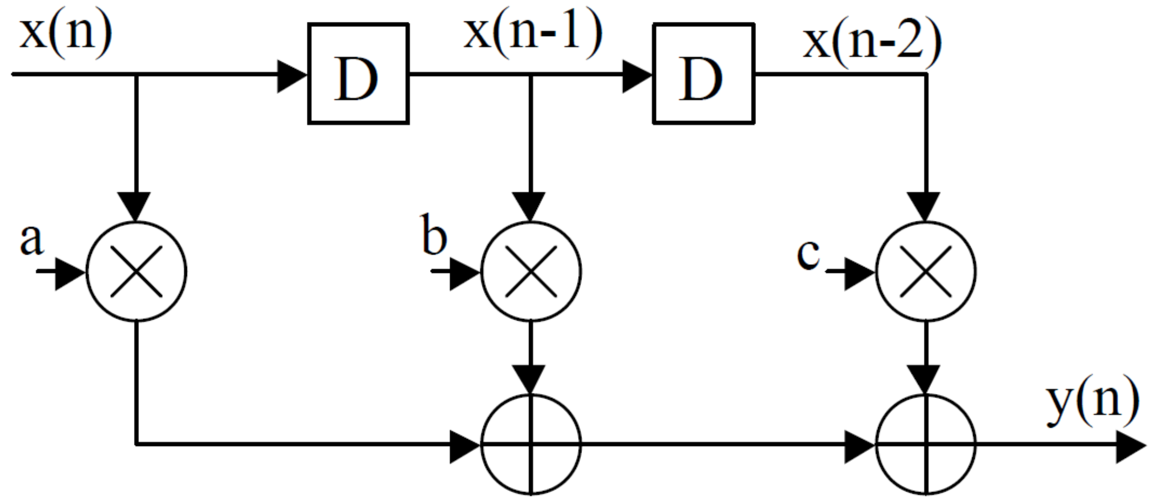
1. Rewrite equations for the *P*-parallel version
2. Follow a set of rules to draw the *P*-parallel DFG

Parallel processing: approach nr. 1

Start by writing the discrete-time equation of the DFG

Example (*P=2*):

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



Then, write *P* concurrent equations

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

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

Parallel processing: approach nr. 1

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

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

In this case, there are two streams: even numbered and odd numbered samples.

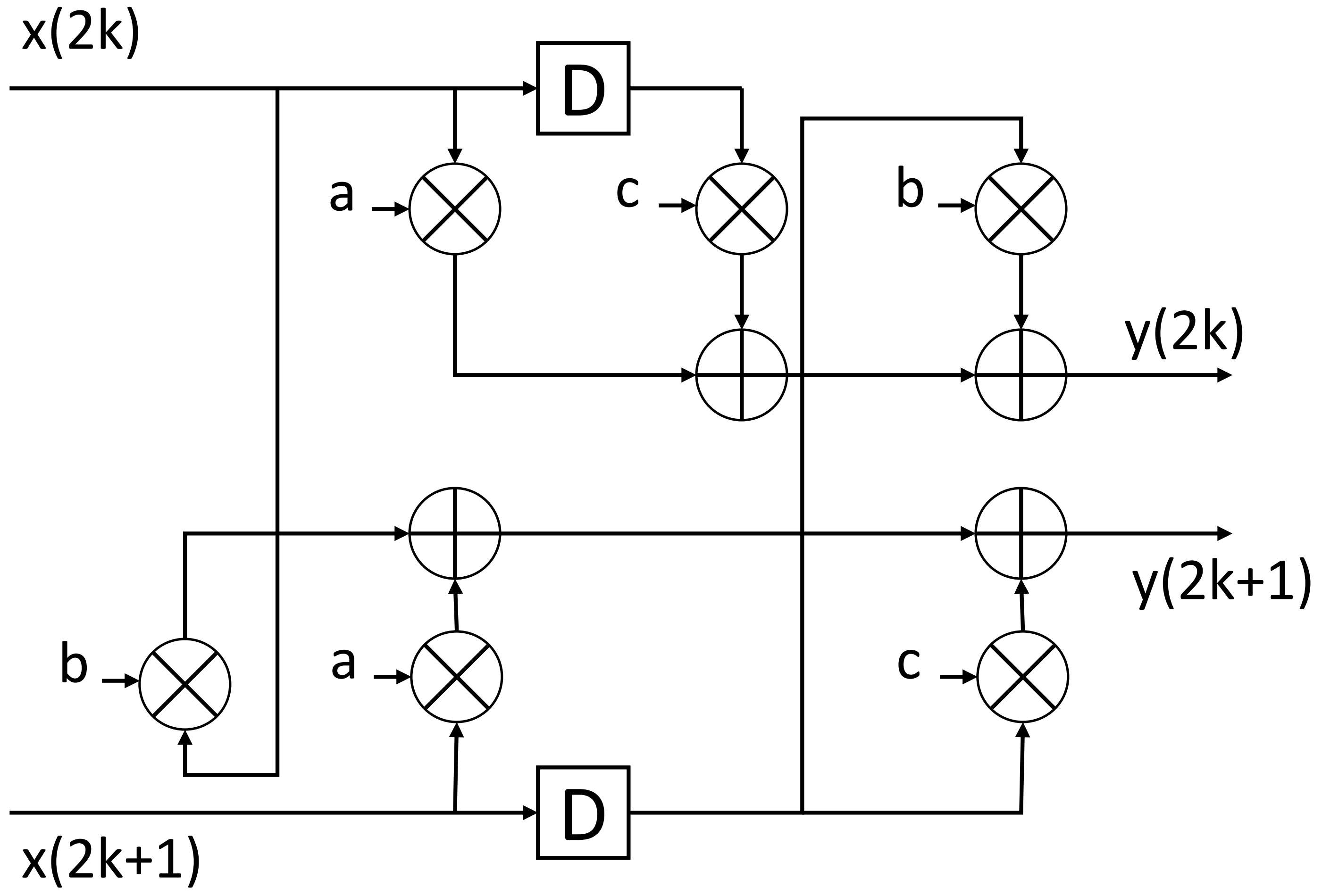
Finally, the *P*-parallel DFG can be drawn from the equations, by taking into account the concept of block delay:

e.g. x(2k-2) is the output of a register receiving x(2k) and x(2k-1) is the output of a register receiving x(2k+1)

Parallel processing: approach nr. 1

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

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



Parallel processing: approach nr. 2

To draw the *P*-parallel version (*Gp*) of a DFG (*G*), follow the following rules:

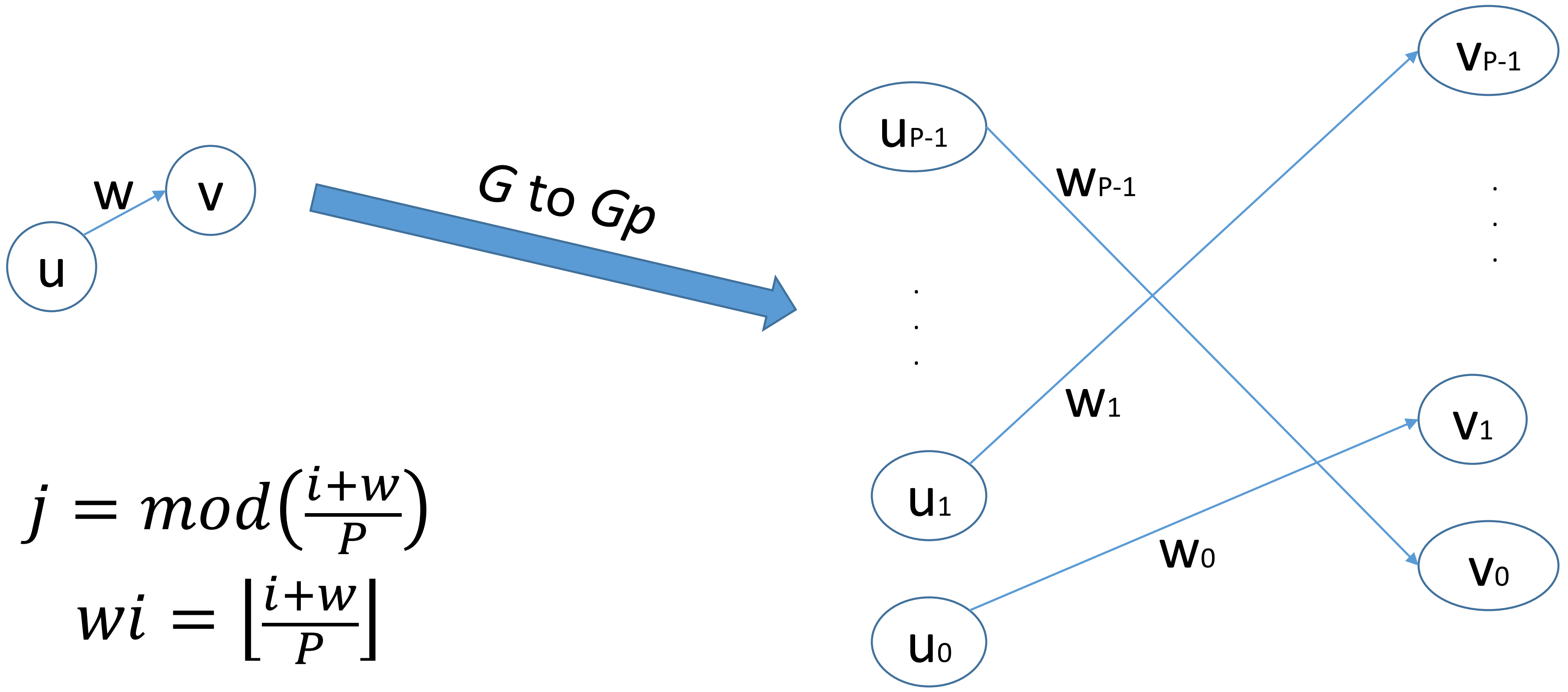
1. For each node *u* in *G*, allocate *P* nodes (*uj*) of the same type as *u* (*j=0, 1, … P-1*)
2. Given the arc *e* in *G*, connecting nodes *u* to node *v* with *w* registers, allocate in *Gp* P arcs, going from *ui* to *vj* with *wi* registers, according to

$$j = mod\left(\frac{i+w}{P}\right)$$

$$wi = \left\lfloor\frac{i+w}{P}\right\rfloor$$

i=0, 1, … P-1

Parallel processing: approach nr. 2



$$j = mod\left(\frac{i+w}{P}\right)$$

$$wi = \left\lfloor\frac{i+w}{P}\right\rfloor$$

Effect of parallel processing

Ideally, the conversion of a DFG into its *P*-parallel version produces the following effects:

1. The complexity is increased by a factor *P* (all nodes are replicated, the global number of registers does not change)
2. The critical path delay does not change (sometimes, it increases)
3. The throughput is increased by a factor *P*:

$$th=P/tcp$$

4. The *AT* product (or the TAR) does not change

Pipelining and Parallel Processing for Low Power

When sample speed does not need to be increased, these techniques can be used for lowering the power consumption

Propagation delay Tpd of CMOS circuit

$$t_{pd} = \frac{C_{ch} V_{DD}}{k(V_{DD} - V_{th})^2}$$

Power consumption (dynamic) in CMOS circuit

$$P_d = C_{tot} V_{DD}^2 f_{CK}$$

- c_{ch} capacitance charged or discharged along the critical path
- V_{th} threshold voltage
- c_{tot} total capacitance of the circuit

Pipelining for Low Power

Initially, the sequential architecture has clock period and power dissipation

$$T_{CK} = \frac{1}{f_{CK}} \qquad P_{seq} = C_{tot} V_{DD}^2 f_{CK}$$

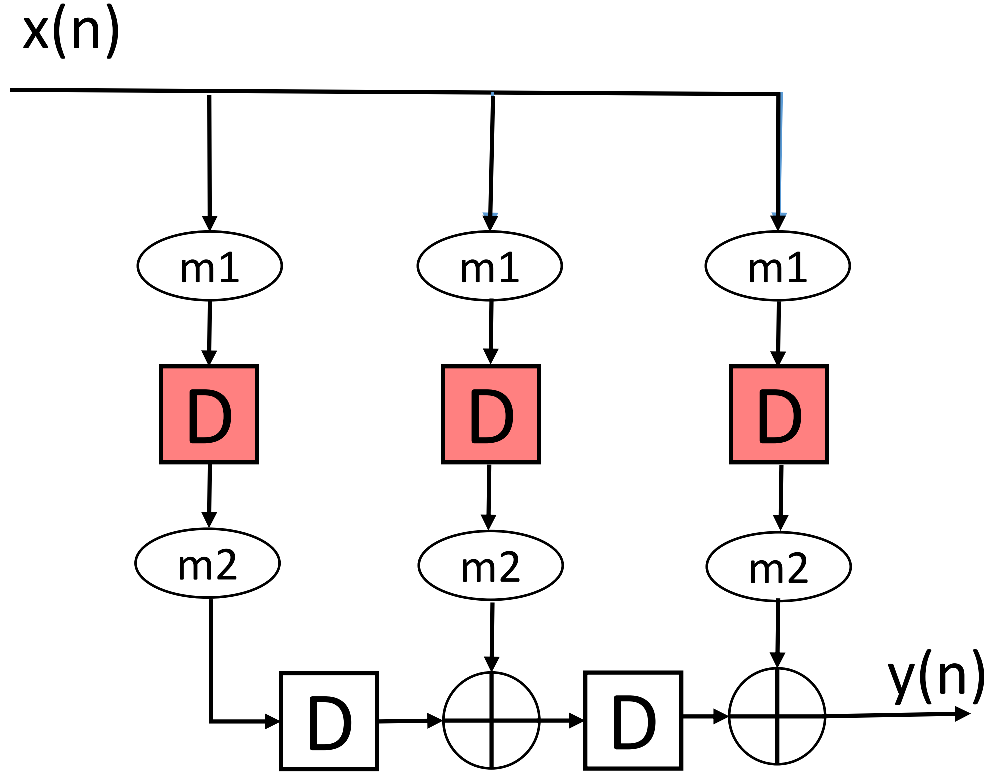
With *M*-level pipelining, the critical path is reduced by *1/M*, and the capacitance to be charged/discharged in a single clock cycle is also reduced by *1/M.*
If the same f_{CK} is maintained, only a fraction (*1/M*) of the original capacitance is charged/discharged in the same amount of time. This implies that the supply voltage can be reduced by a factor β (0<β <1):

$$P_{pipe} = C_{tot} \beta^2 V_{DD}^2 f_{CK} = \beta^2 P_{seq}$$

$$t_{seq} = \frac{C_{ch} V_{DD}}{k(V_{DD} - V_{th})^2} = t_{pipe} = \frac{C_{ch}/M \beta V_{DD}}{k\left(\beta V_{DD} - V_{th}\right)^2} \rightarrow M(\beta V_{DD} - V_{th})^2 = \beta (V_{DD} - V_{th})^2$$

Pipelining for Low Power: example

FIR filter, length 3, fine grain pipelining:



Cm=5 Ca   Cm1=3 Ca   Cm2=2 Ca

Vt=0.6 V   VDD=5 V

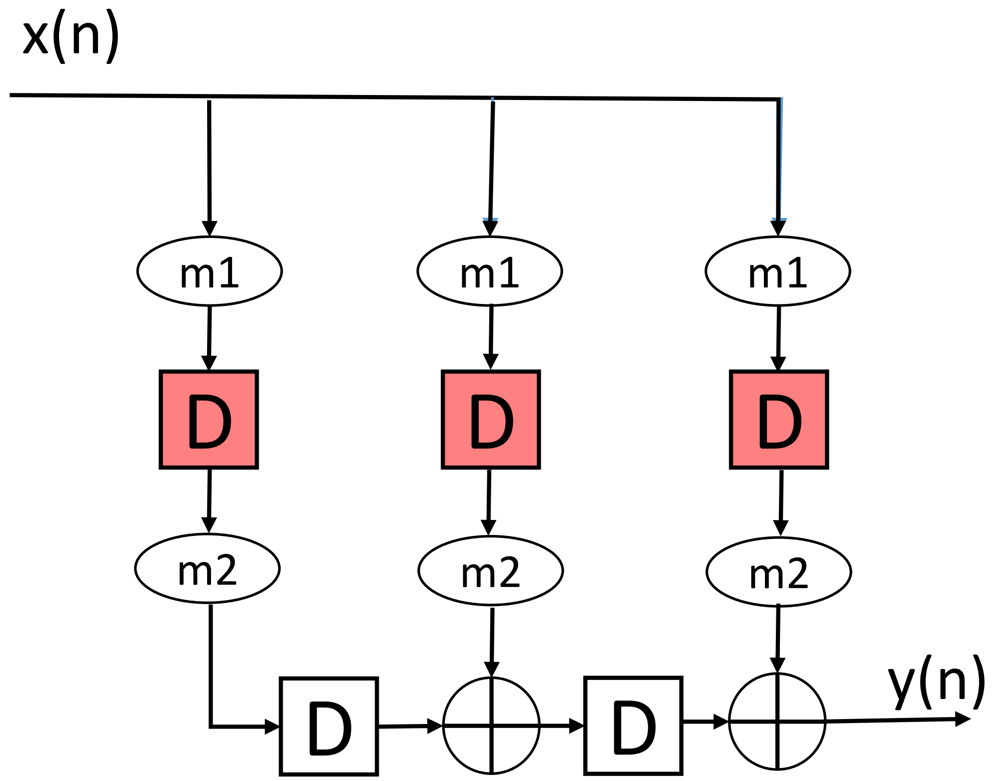
With <u>no pipelining</u>:

Critical path has Cd=Cm+Ca=6 Ca

$$P_{np} = V_{DD}^2 C_L f \qquad T_{np} = \frac{C_d V_{DD}}{k(V_{DD} - V_{th})^2}$$

Pipelining for Low Power: example

FIR filter, length 3, fine grain pipelining:



$$P_{np} = V_{DD}^2 C_L f \qquad T_{np} = \frac{C_d V_{DD}}{k(V_{DD} - V_{th})^2}$$

With <u>pipelining</u> (M=2):

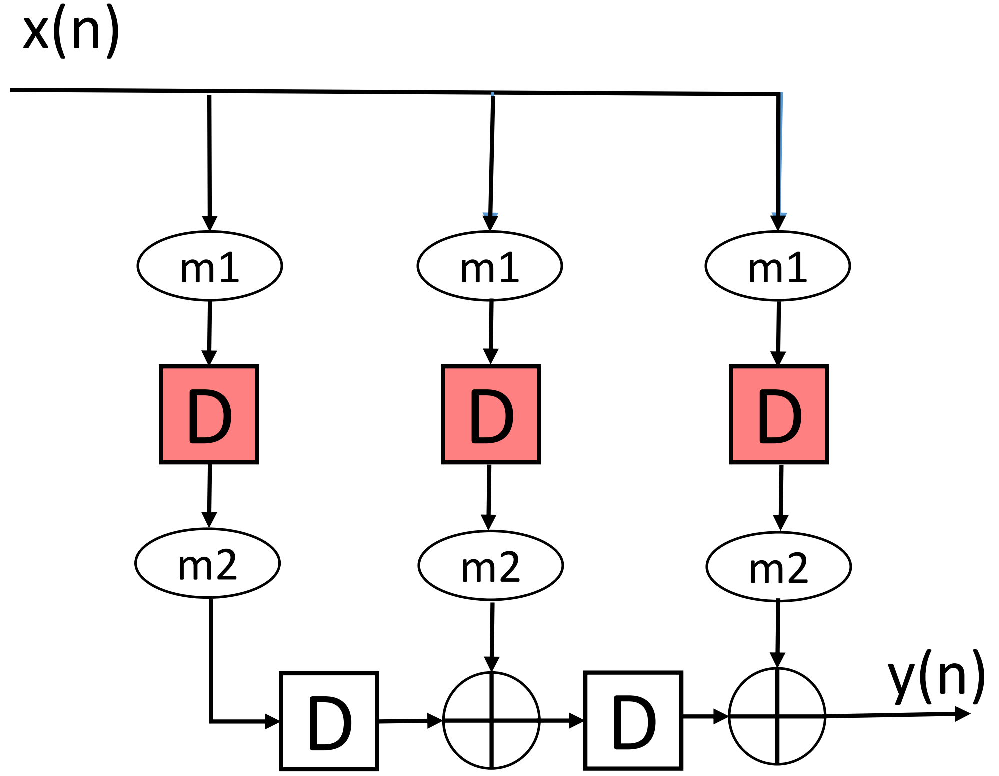
CL/M=3 Ca = Cm1 = Cm2+Ca

$$P_{pi} = \alpha^2 V_{DD}^2 C_L f$$

$$T_{pi} = \frac{{}^{1}/_{M}\, C_d \alpha V_{DD}}{k(\alpha V_{DD} - V_{th})^2} = T_{np}$$

Pipelining for Low Power: example

FIR filter, length 3, fine grain pipelining:



$$P_{pi} = \alpha^2 V_{DD}^2 C_L f$$

$$T_{pi} = \frac{{}^{1}/_{M}\, C_d \alpha V_{DD}}{k(\alpha V_{DD} - V_{th})^2} = T_{np}$$

α =0.6

$$\alpha V_{DD} = 3\ V$$

$$\frac{P_{pi}}{P_{np}} = \alpha^2 = 36\%$$

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}). The supply voltage can be reduced by a factor β, since there is more time to charge the same capacitance

$$t_{par} = Lt_{seq} = \frac{C_{ch}\beta V_{DD}}{k(\beta V_{DD} - V_{th})^2} \rightarrow L(\beta V_{DD} - V_{th})^2 = \beta(V_{DD} - V_{th})^2$$

Parallel processing for Low Power

Given the factor β, the power dissipated by the parallel system is

$$P_{par} = LC_{tot}\beta^2 V_{DD}^2 f_{CK}/L = \beta^2 P_{seq}$$

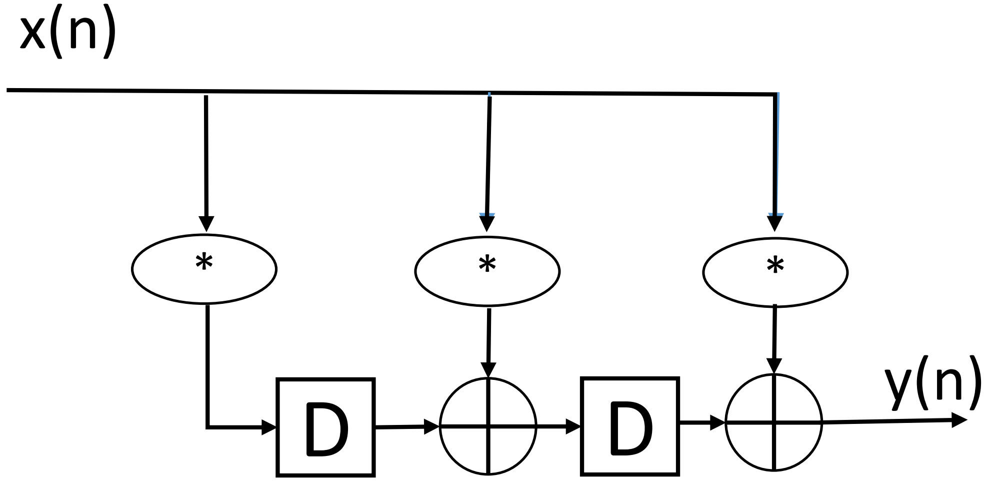
We can decide β to get the same throughput as the sequential architecture:

$$T_{seq} = \frac{C_{ch}V_{DD}}{k(V_{DD} - V_{th})^2} \qquad T_{par} = L\,T_{seq} = \frac{C'_{ch}\,\beta\,V_{DD}}{k(\beta\,V_{DD} - V_{th})^2}$$

From $T_{par} = L\,T_{seq}$, we obtain β and scale V_{DD} to save power

Parallel processing for Low Power: example

FIR filter with 3 taps, L=2:



Cm=5 Ca   Cm1=3 Ca   Cm2=2 Ca

Vt=0.6 V   VDD=5 V

<u>Sequential version</u>:

Critical path has $C_{ch} = C_m + C_a = 6\ C_a$

$P_{seq} = V_{DD}^2 C_L f \qquad T_{seq} = \dfrac{C_{ch} V_{DD}}{k(V_{DD} - V_{th})^2}$

Parallel processing for Low Power: example

Parallel version (L=2):

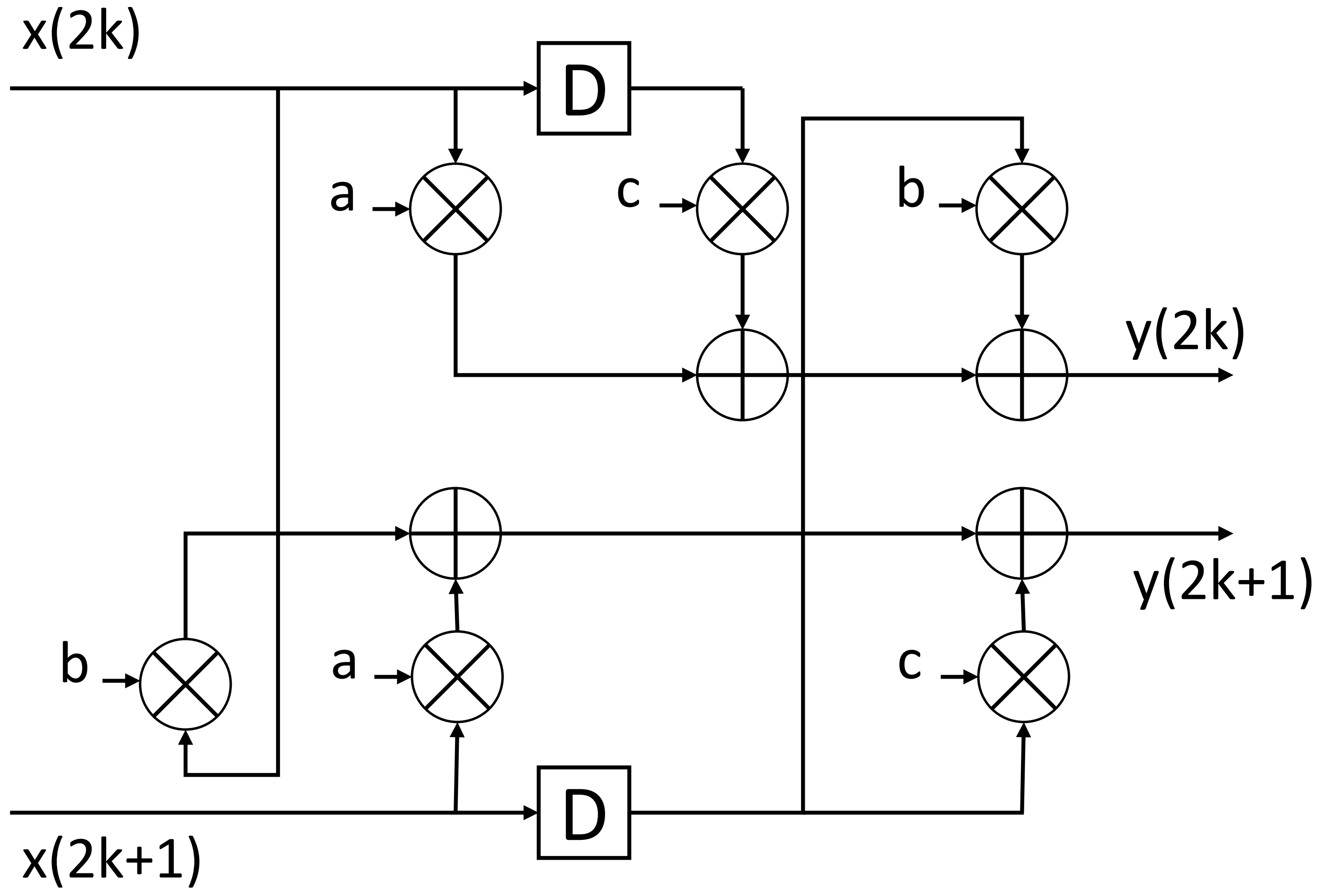
Critical path has

$$C'_{ch} = C_m + 2C_a = 7\ C_a$$

$$T_{par} = L\ T_{seq}$$

$$\frac{7\ C_a\ \beta\ V_{DD}}{k(\beta\ V_{DD} - V_{th})^2} = 2\,\frac{6\ C_a V_{DD}}{k(V_{DD} - V_{th})^2}$$

β = 0.67 $\frac{P_{par}}{P_{seq}} = \beta^2 = 45\%$



Loop bound

Assume the execution times of multiplier and adder are *Tm* and *Ta*

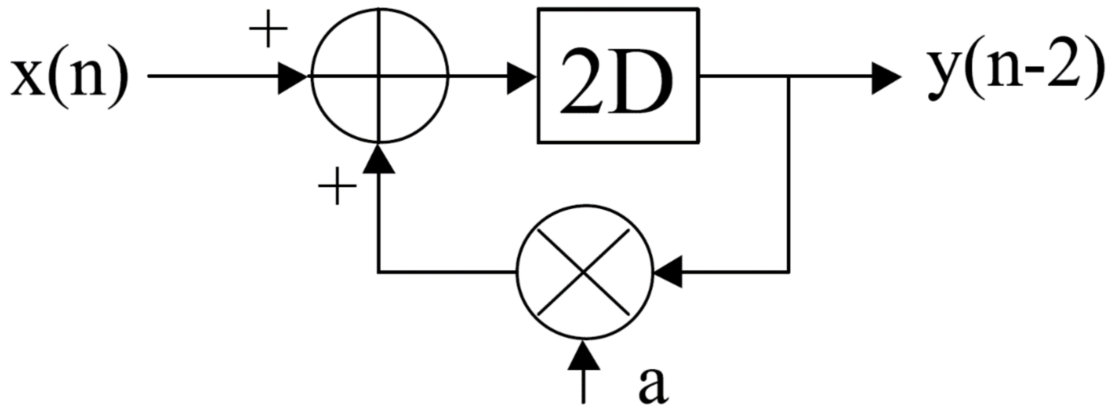
Then, the clock period for this example is Tck>*Tm+Ta*

And the throughput is *th<1/(Tm+Ta)*

y(n)=a*y(n-2)-x(n)

Loop bound:

lb= $\frac{Ta+Tm}{2}$



Loop bound and iteration bound

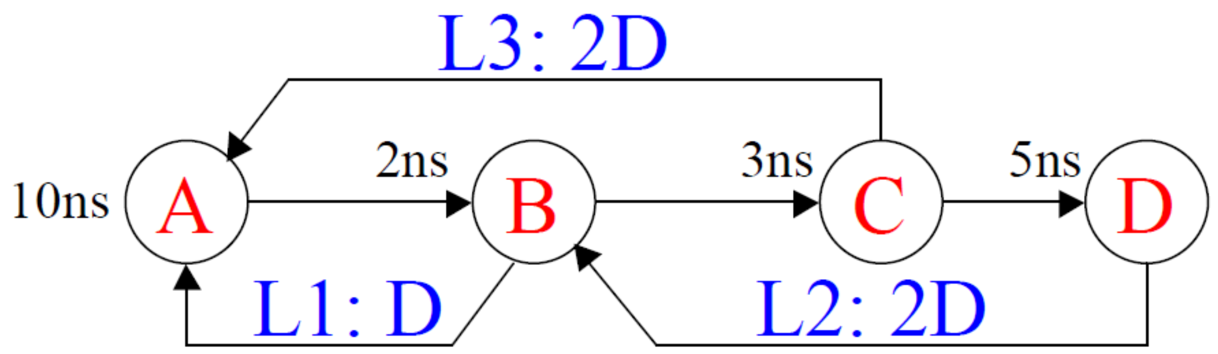
In general, the loop bound is defined as the ratio between the computation time of the loop and the number of registers located along the loop

For a DFG with multiple loops, we define the iteration bound as the largest loop bound:

$$T_\infty = \max_j \left\{ \frac{T_j}{W_j} \right\}$$

where *Tj* is the computation time for loop *j* and *Wj* is the number of registers along loop *j*

Example



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

$$T_{\infty} = \max\{T_{L1}, T_{L2}, T_{L3}\} = 12\ ns$$

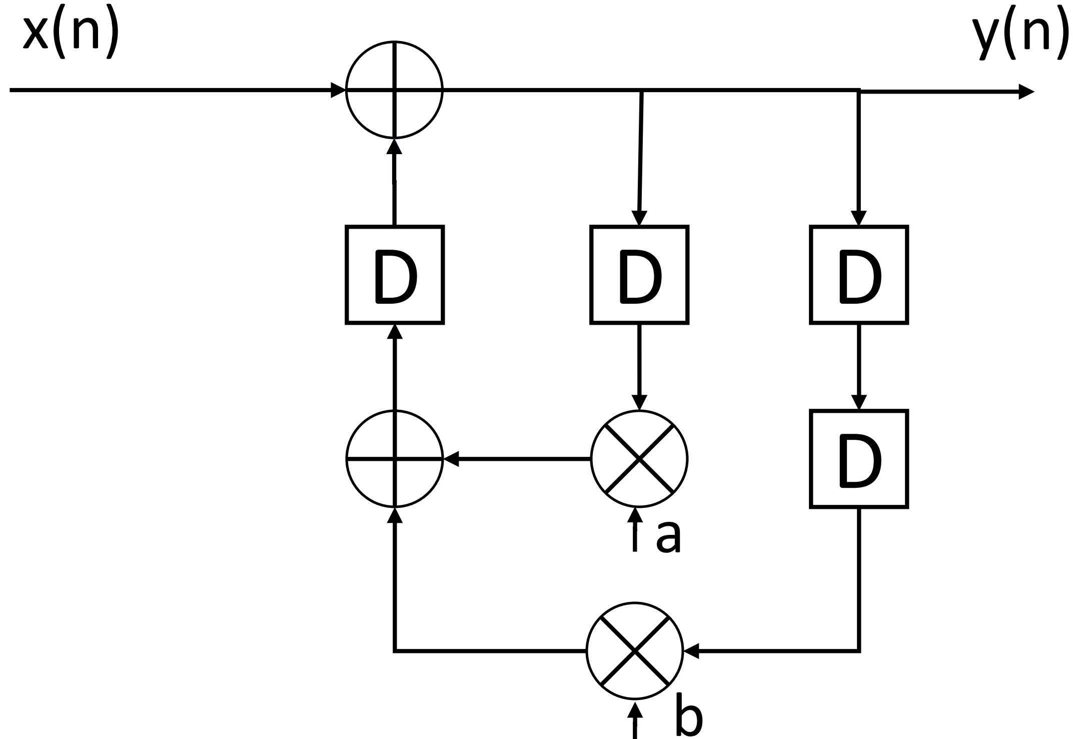
Tck >10+2+3+5=20 ns for this DFG

However, a DFG transformation exists which results in an improved implementation, with *Tck* as low as 12 ns

Retiming

It is about moving the registers in a DFG without changing the system behavior. Retiming can be exploited to reduce the length of the critical path and thus improving the performance

Example:



$$y(n) = x(n) + ay(n-2) + by(n-3)$$

If we implement this structure, performance is computed as

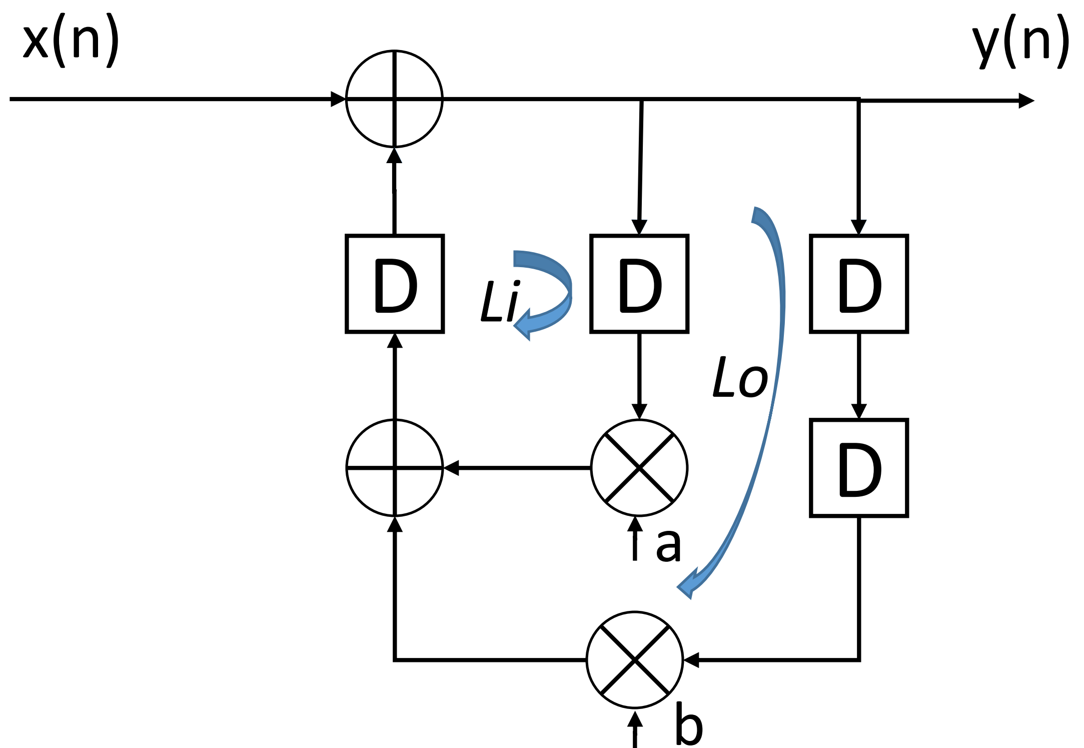
$$t_{cp} = T_A + T_M = 3 \qquad th = \frac{1}{T_A + T_M} = \frac{1}{3}$$

$(T_M = 2 \qquad T_A = 1)$

Retiming: example

Let us derive the iteration bound for this recursive DFG.

There are two loops: an inner loop (*Li*) and an outer loop (*Lo*)



Li: $\frac{T_M+2T_A}{2}$

Lo: $\frac{T_M+2T_A}{3}$

The iteration bound is

$T_\infty = \frac{T_M+2T_A}{2}$=2

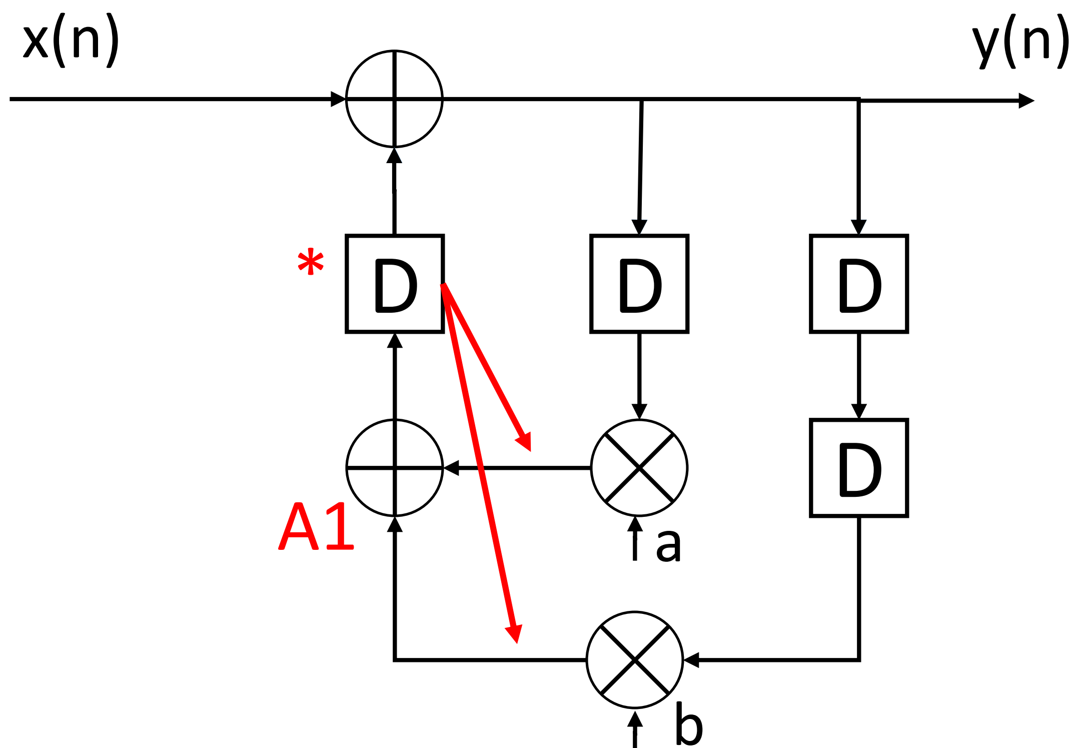
There must be an universal technique able to provide an architecture with

$$t_{cp} = 2 \qquad th = \frac{1}{2}$$

Retiming: example

We look for a different location of the registers that

1. does not modify the I/O behavior
2. achieves the performance bound



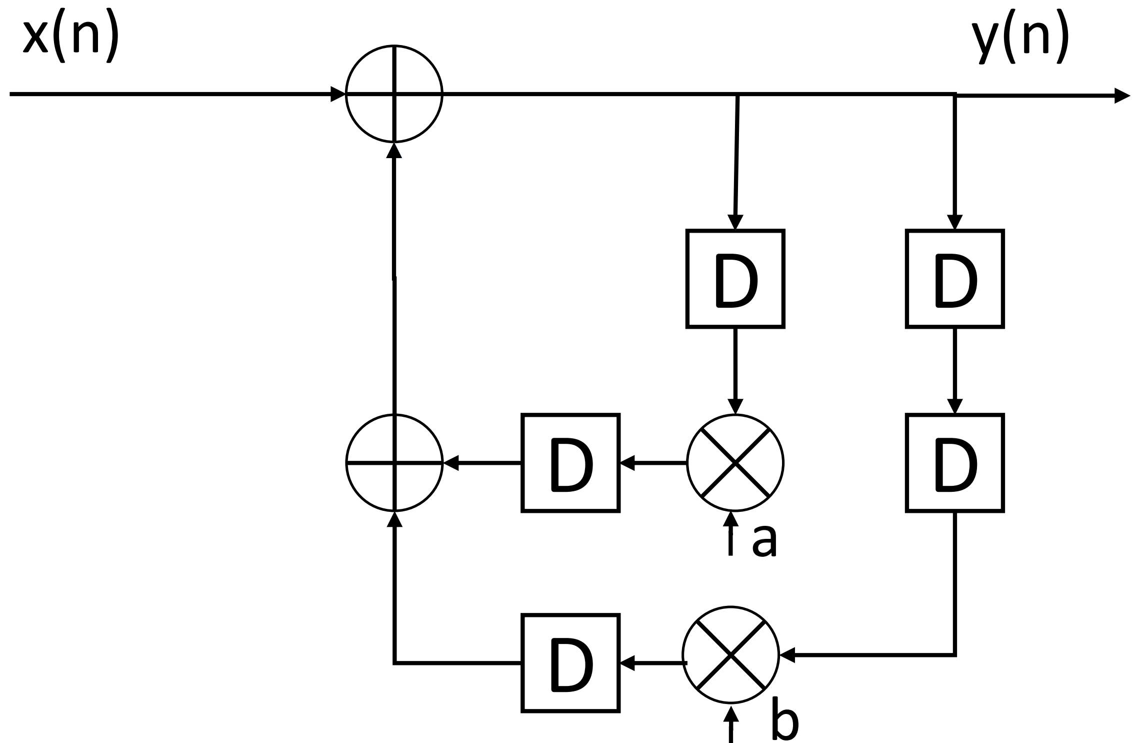
Method 1: <u>direct inspection</u>

We move register * to the inputs of the adder A1

By doing so, we split the critical paths going through multipliers and adder A1

Retiming: example

Retimed DFG has the same topology as the initial one, but different positions of registers (and potentially a different number of registers)



The performance for the retimed DFG is:

$$t_{cp} = 2T_A = T_M = 2 \qquad th = \frac{1}{2}$$

and the behavior is unaffected:

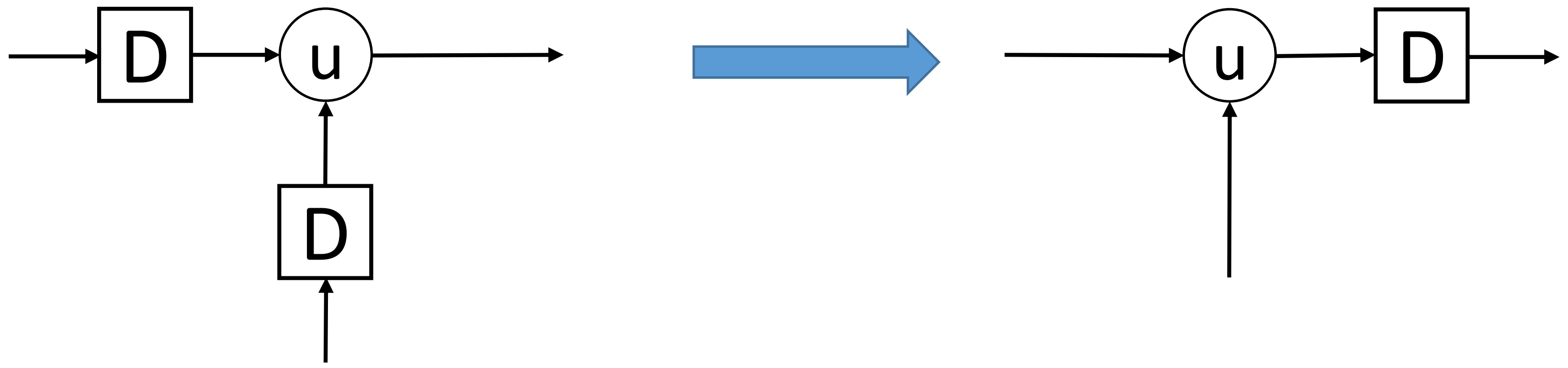
$$y(n) = x(n) + ay(n-2) + by(n-3)$$

Retiming of a node

For a given node u, the retiming operation implies:

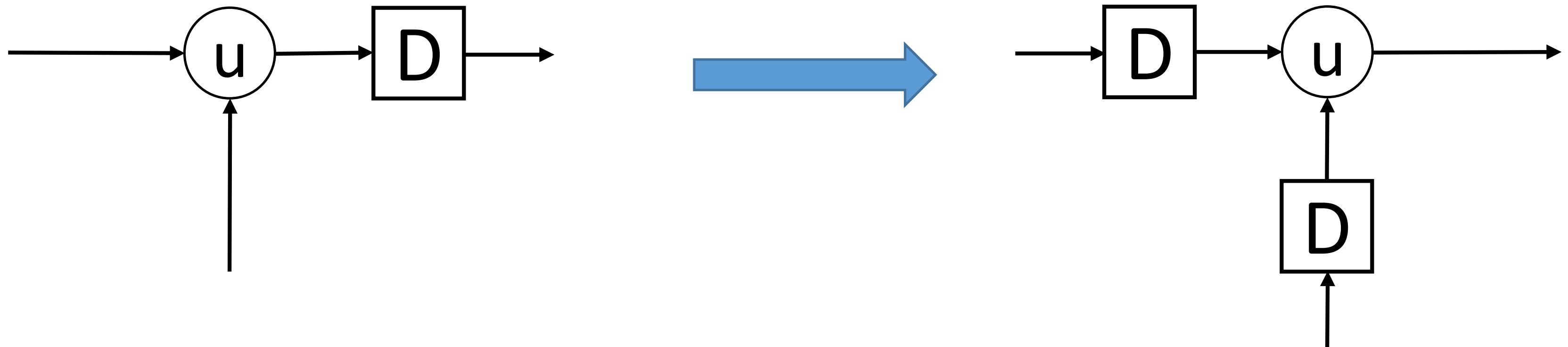
either a shift of *k* registers from input arcs to output arc:

$$r_u = k$$



or a shift of *k* registers from output arc to input arcs:

$$r_u = -k$$

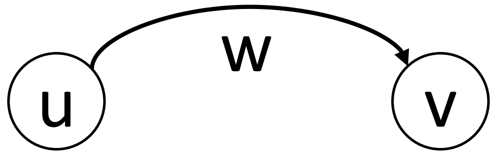


Method 2: formal retiming

Given a DFG, we derive tcp and iteration bound.

If $t_{cp} = T_\infty$, there is no room for improving the speed performance.

If $t_{cp} > T_\infty$, for each edge *e*, with *w* registers, we impose one of the following constraints:



Nodes u and v are along a potential critical path if the sum of their delay is larger than the target clock period

$$w_r(e) = w(e) + r_u - r_v \geq 0$$

(if the edge is not part of any potential critical path)

$$w_r(e) = w(e) + r_u - r_v \geq 1$$

(if the edge is part of a potential critical path)

By imposing these constraints on a DFG with *m* edges, we obtain a set of *m* conditions, which must be satisfied to shorten the critical path delay.

Method 2: example

The solution of the set of constraints provides the values of *r* to be applied to every edge to obtain the retimed DFG.

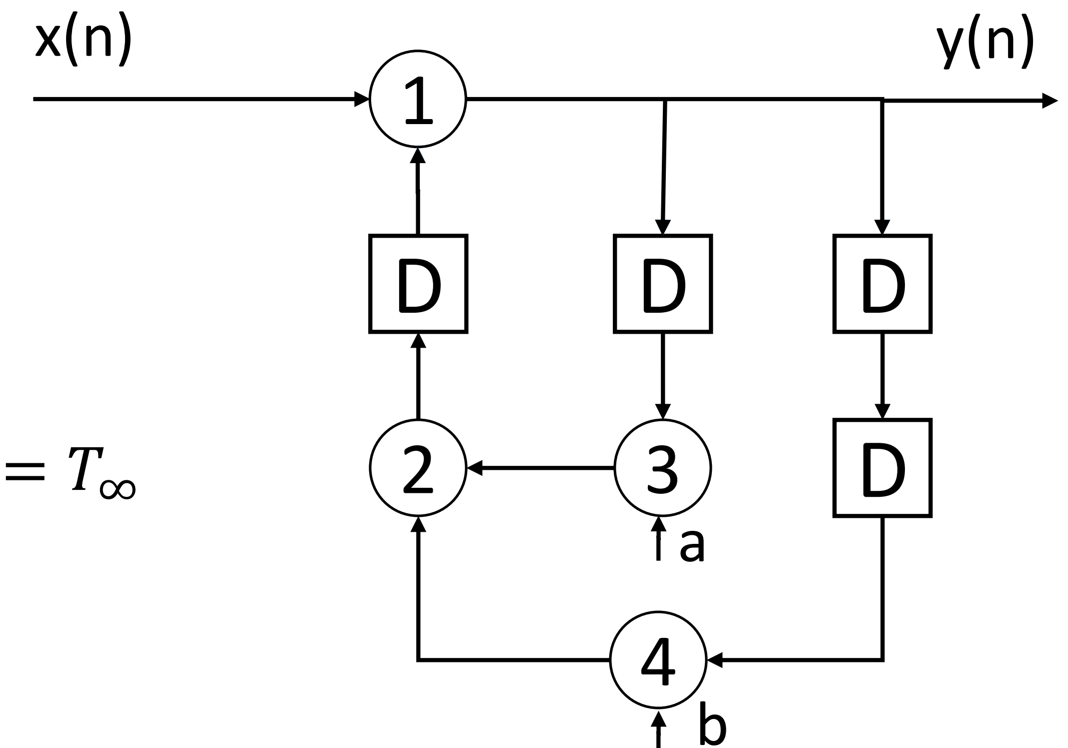
We now use method 2 for the same example already solved with method 1.

For edge e_{21}, we have constraint
$w_r(e_{21}) = w(e_{21}) + r_2 - r_1 \geq 0$

$t_1 + t_2 = 2 = T_\infty$

that can be written as
$1 + r_2 - r_1 \geq 0$ or

$r_1 - r_2 \leq 1$



Method 2: example

$$w_r(e_{21}) = w(e_{21}) + r_2 - r_1 \geq 0$$
$$w_r(e_{32}) = w(e_{32}) + r_3 - \boxed{r_2 \geq 1}$$
$$w_r(e_{42}) = w(e_{42}) + r_4 - r_2 \geq 1$$
$$w_r(e_{13}) = w(e_{13}) + r_1 - r_3 \geq 1$$
$$w_r(e_{14}) = w(e_{14}) + r_1 - r_4 \geq 1$$

$t_3 + t_2 = 3 > T_\infty$

$$\Downarrow$$

$1+r_2 - r_1 \geq 0$

$r_3 - r_2 \geq 1$

$r_4 - r_2 \geq 1$

$1+r_1 - r_3 \geq 1$

$2+r_1 - r_4 \geq 1$

$$\Rightarrow$$

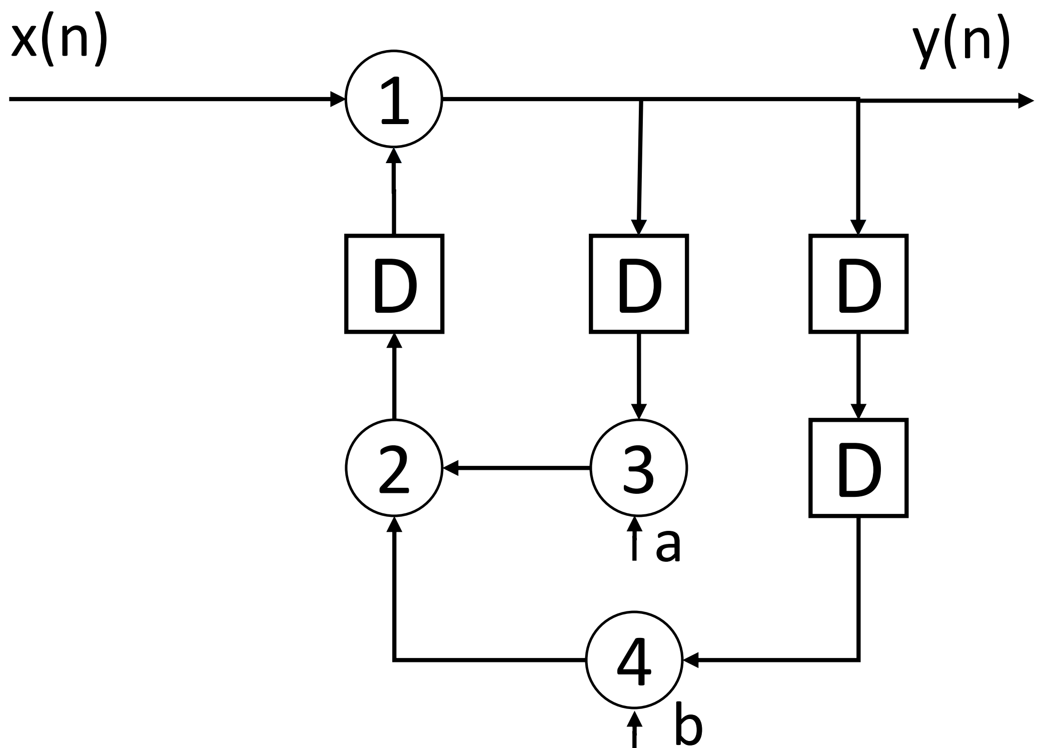
$1 \geq r_2 - r_1$

$r_3 - r_2 \geq 1$

$r_4 - r_2 \geq 1$

$r_1 - r_3 \geq 0$

$1+r_1 - r_4 \geq 0$



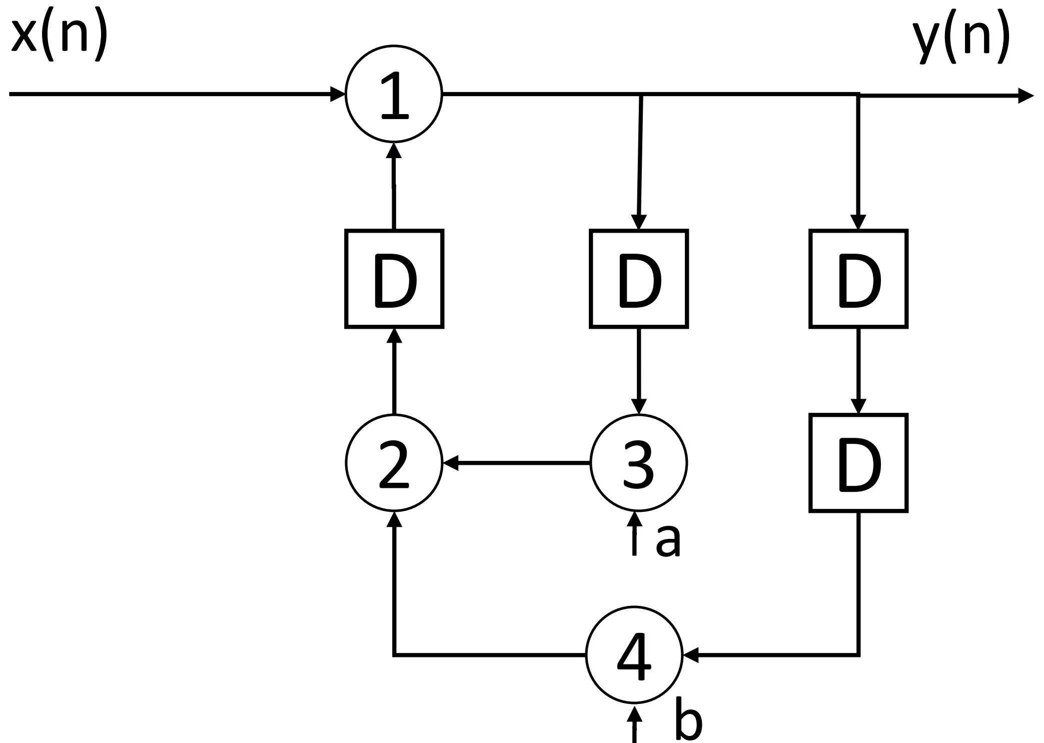
Method 2: example

$$1 \geq r_2 - r_1$$
$$r_3 - r_2 \geq 1$$
$$r_4 - r_2 \geq 1$$
$$r_1 - r_3 \geq 0$$
$$1 + r_1 - r_4 \geq 0$$

This procedure leads to a mathematical problem that can have no solution (the bound cannot be achieved by means of retiming), one or multiple solutions

In this case, one possible solution is:
r(1) = 0; r(2) = -1; r(3) =r(4) = 0
(solution found with method 1).

Another solution is given by:
r(1) = 0; r(2) = -1; r(3) = 0; r(4) = 1



Method 3: cut-set retiming

This is a simpler method, useful to quickly find a local solution.

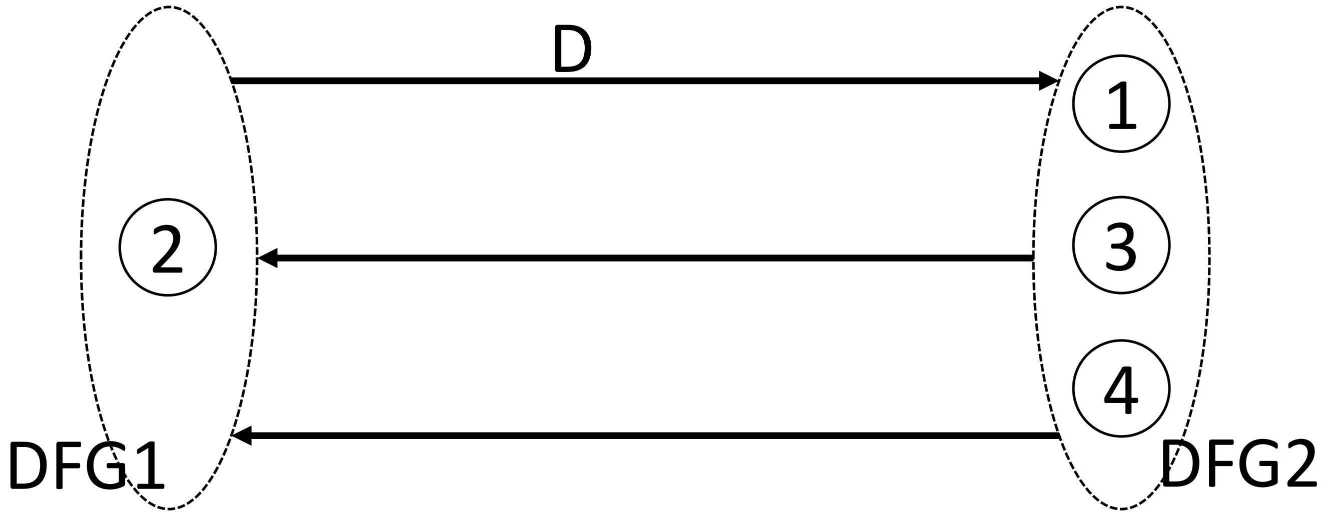
First, we identify a cut-set in the DFG that breaks the critical path
Then, we insert/remove registers along the arcs in the cut-set.
Let us call *G1* and *G2* the two sub-graphs generated by removing the arcs of the cut-set from the DFG:

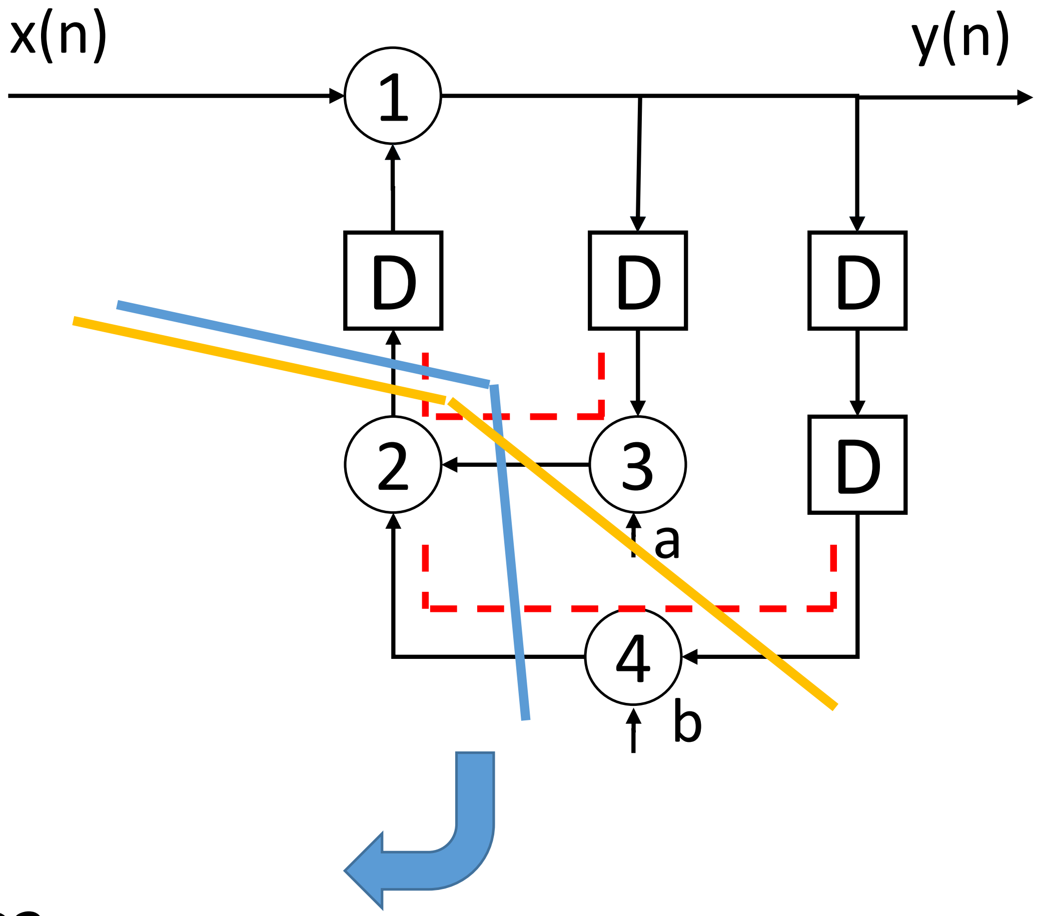
- we can increment by *k* the number of registers on each arc going from *G1* to *G2,*
- provided that we decrement by *k* the number of registers on each arc going in the opposite direction (from *G2* to *G1*)

(*k* is positive or negative integer)

Cut-set retiming: example

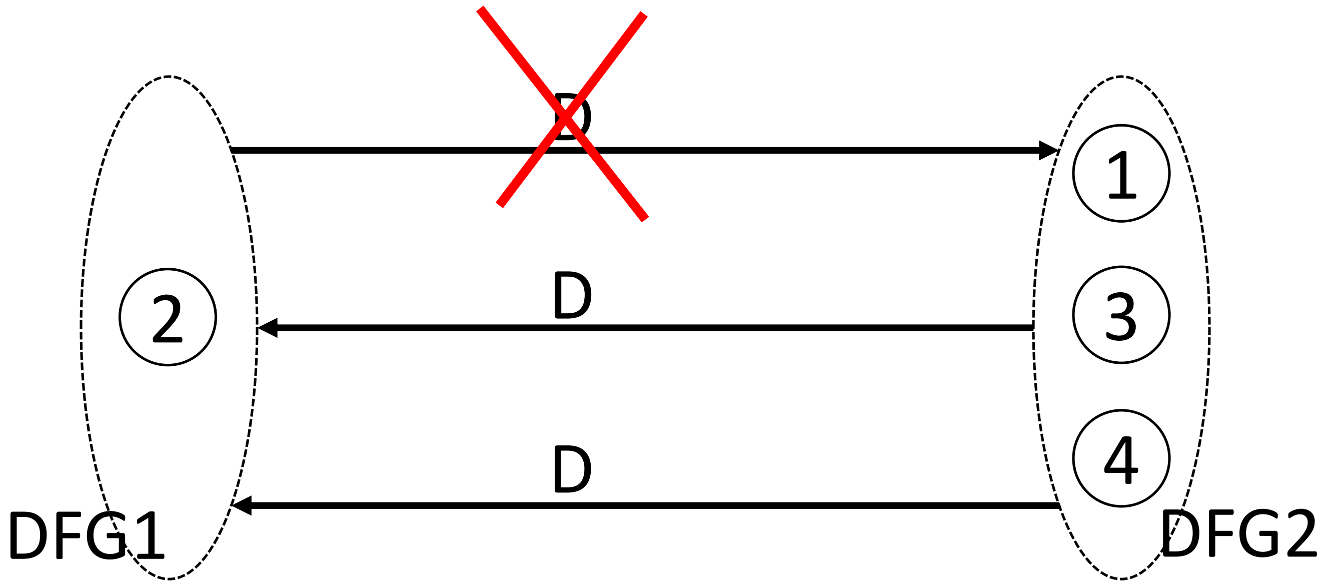
Critical path: - - - -
Cut-set 1: ▬
Cut-set 2: ▬
With cut-set 1, we have DFG1 formed by node 2 and DFG2 formed by nodes 1,3,4.
Arcs between the two sub-graphs are:

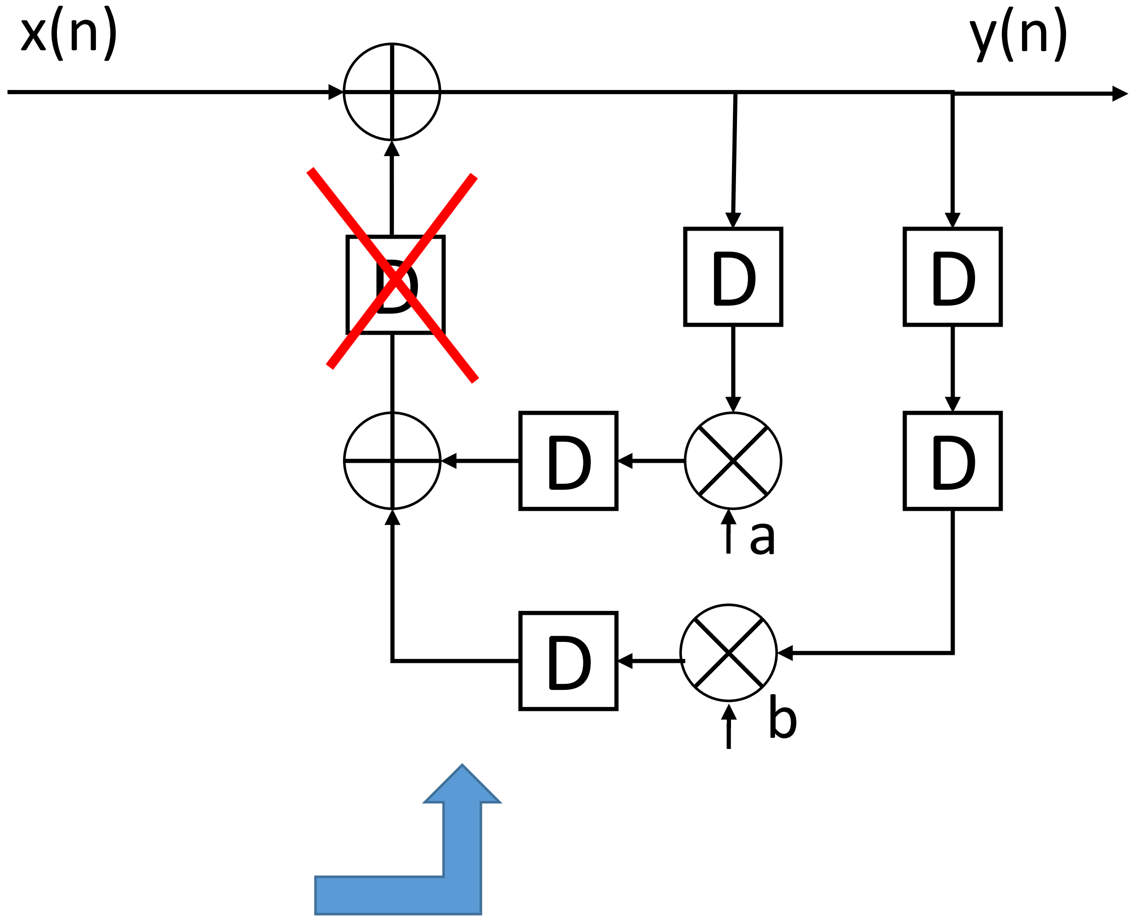




Cut-set retiming: example

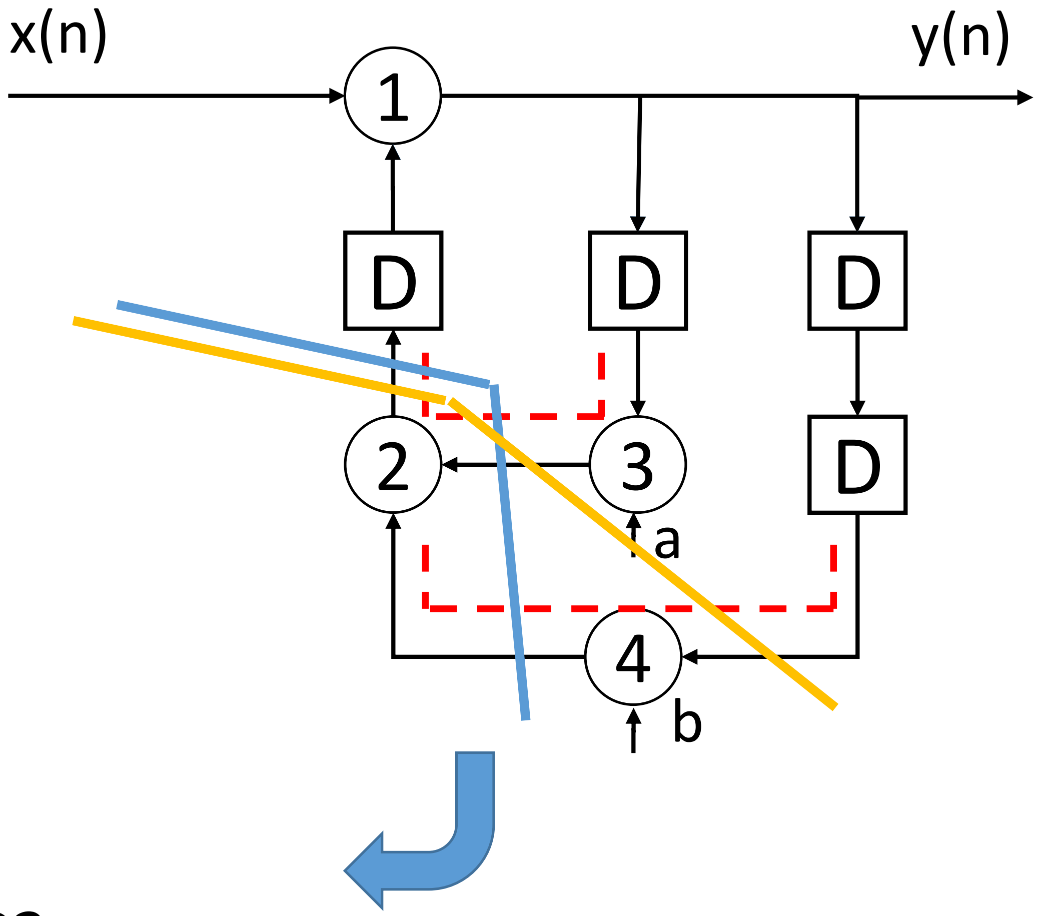
By setting k=-1, we remove the register from the arc connecting node 2 to node 1 and insert two registers along the arcs entering node 2

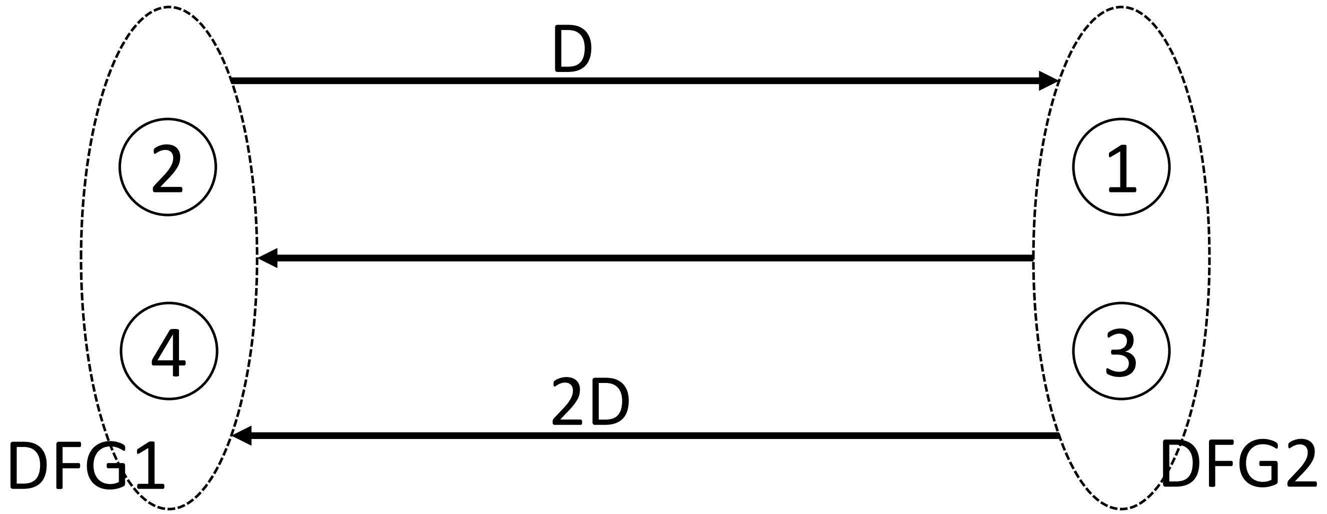




Cut-set retiming: example

With cut-set 2, we have DFG1 formed by nodes 2 and 4, and DFG2 formed by nodes 1 and 3. However, in this case, we are not able to break the critical path between nodes 2 and 4.





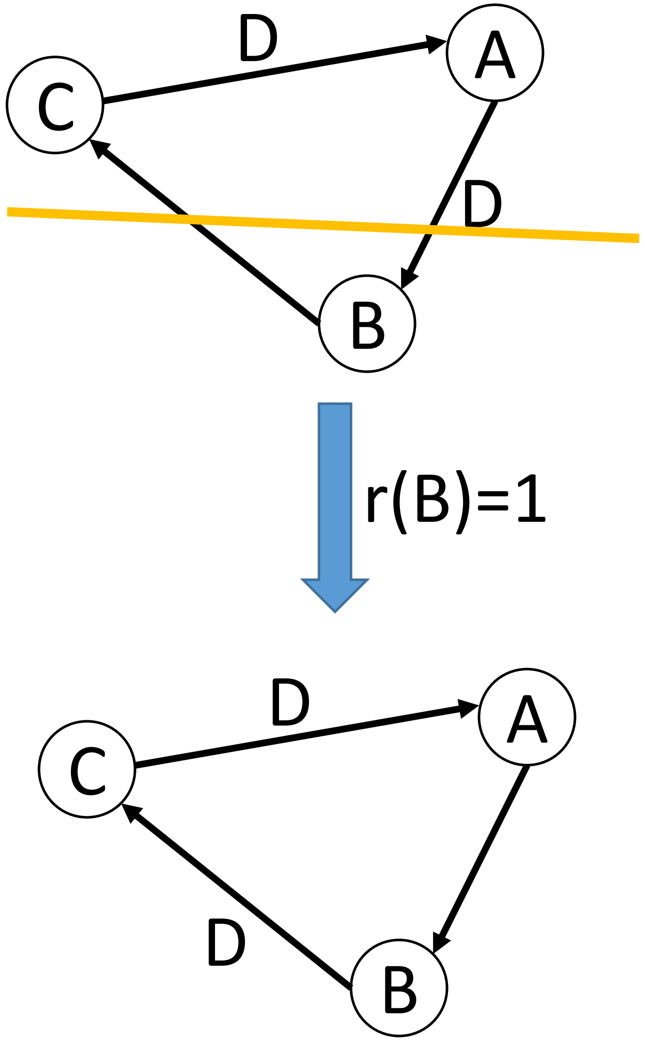
Again about the iteration bound

In this DFG, the propagation delays for nodes A, B and C are 4, 2 and 6 respectively. The critical path delay and the iteration bound are:

$$t_{cp} = \max\{t_A, t_B + t_C\} = 8$$

$$T_\infty = \frac{t_A + t_B + t_C}{2} = 6$$

Ideally, the T_∞ is the t_{cp} we get by distributing the global computational effort uniformly along the loop. In the retimed graph, we execute tasks A and B in one cycle and task C in another cycle. Both cycles take the same time.



Unfolding

Unfolding, or loop unrolling, is the name given to the conversion from a sequential to a parallel DFG in the case of applications with loops.

Method 1: writing multiple discrete-time equations

Method 2: follow the two rules given for feedforward graphs

1. For each node u in G, allocate P nodes (uj) of the same type as u (j=0, 1, … P-1)
2. Given the arc e in G, connecting nodes u to node v with w registers, allocate in Gp P arcs, going from ui to vj with wi registers according to

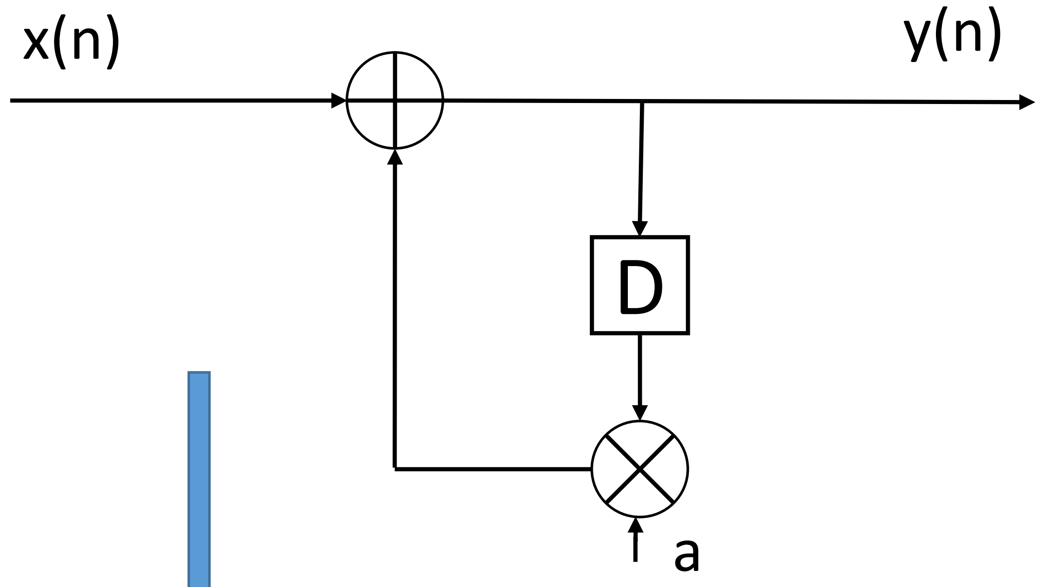
$$j = mod\left(\frac{i+w}{P}\right)$$

$$wi = \left\lfloor \frac{i+w}{P} \right\rfloor$$

i=0, 1, … P-1

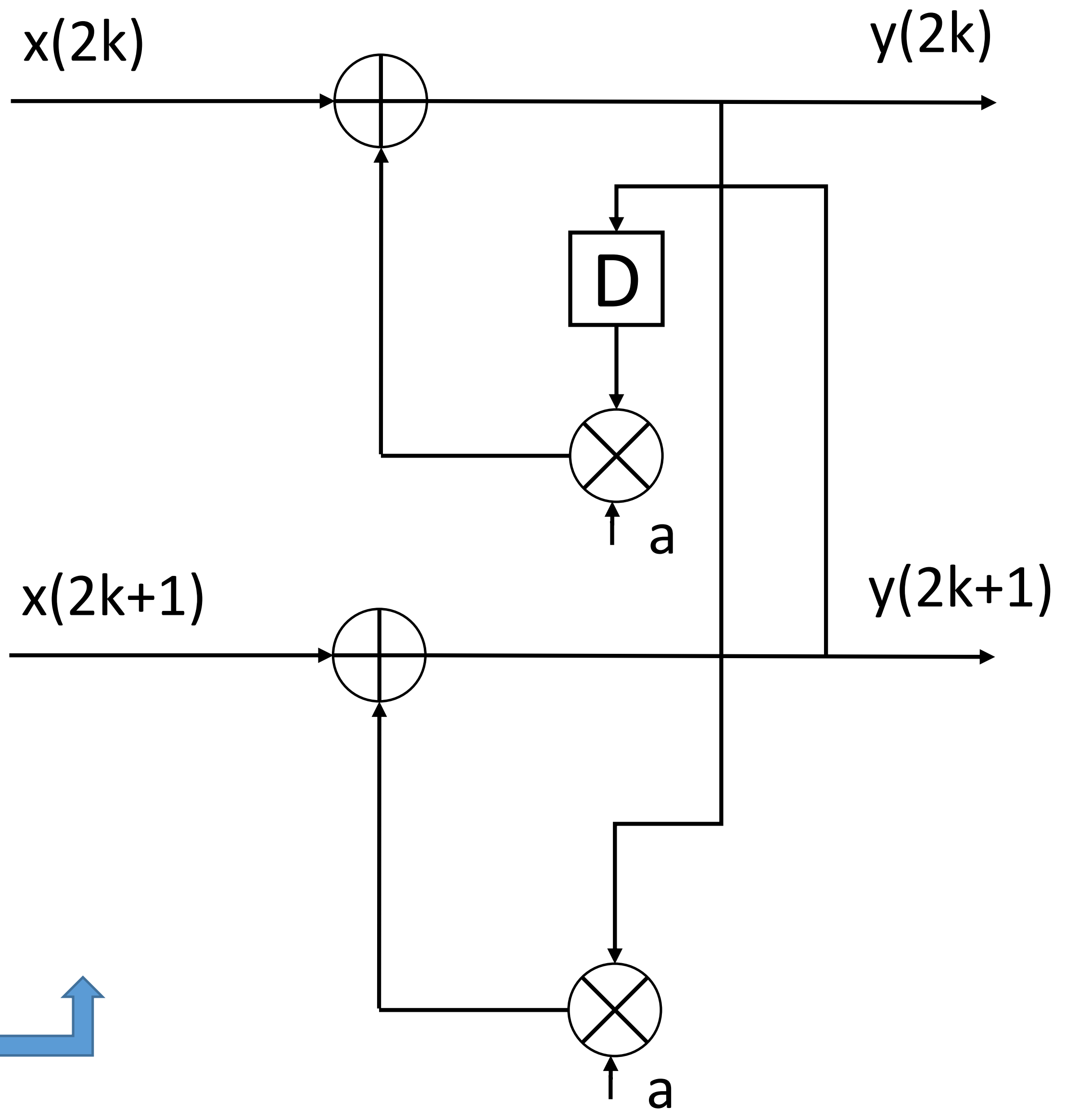
Unfolding: example1

y(n) = x(n)+a*y(n-1)



y(2k) = x(2k)+a*y(2k-1)

y(2k+1) = x(2k+1)+a*y(2k)



Unfolding: example1

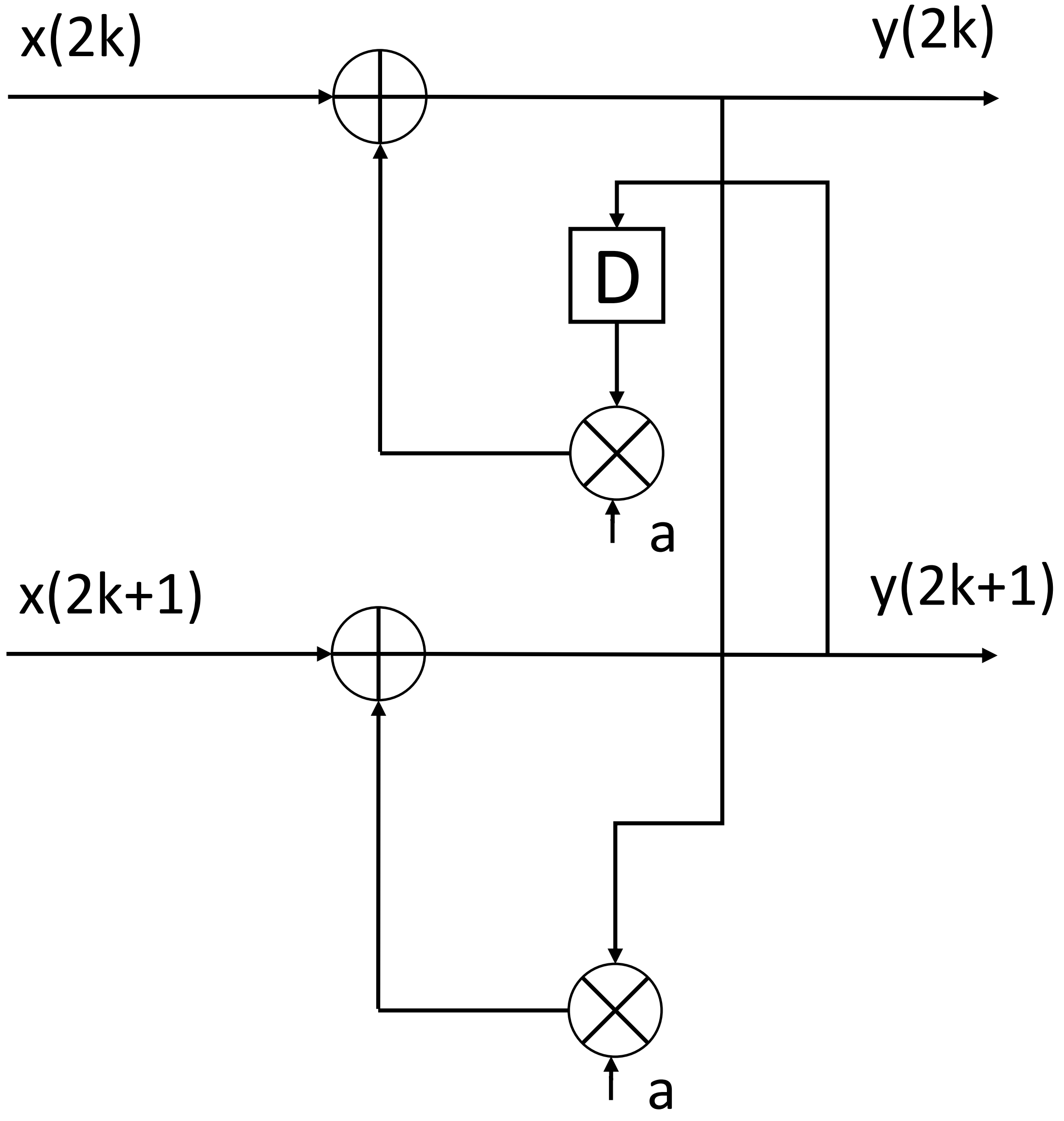
However, in this case, initial DFG and unfolded one achieve the same performance.

Initial DFG:

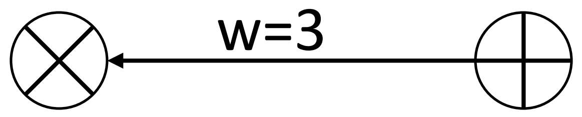
$t_{cp} = t_A + t_M$
th $= 1/(t_A + t_M)$
$T_\infty = t_A + t_M = t_{cp}$

Unfolded DFG:

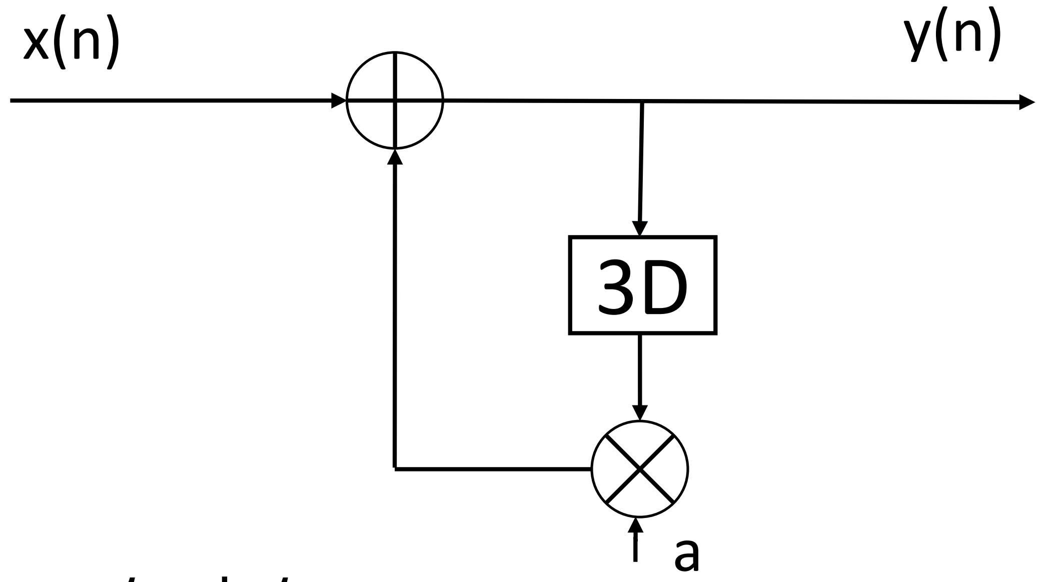
$t_{cp} = 2t_A + 2t_M$
th $= 2/(2t_A + 2t_M) = 1/(t_A + t_M)$



Unfolding: example2



y(n) = x(n)+a*y(n-3) *P=2*



$$t_{cp} = t_A + t_M$$
$$\text{th} = 1/(t_A + t_M)$$
$$T_\infty = \frac{(t_A+t_M)}{3} < t_{cp}$$

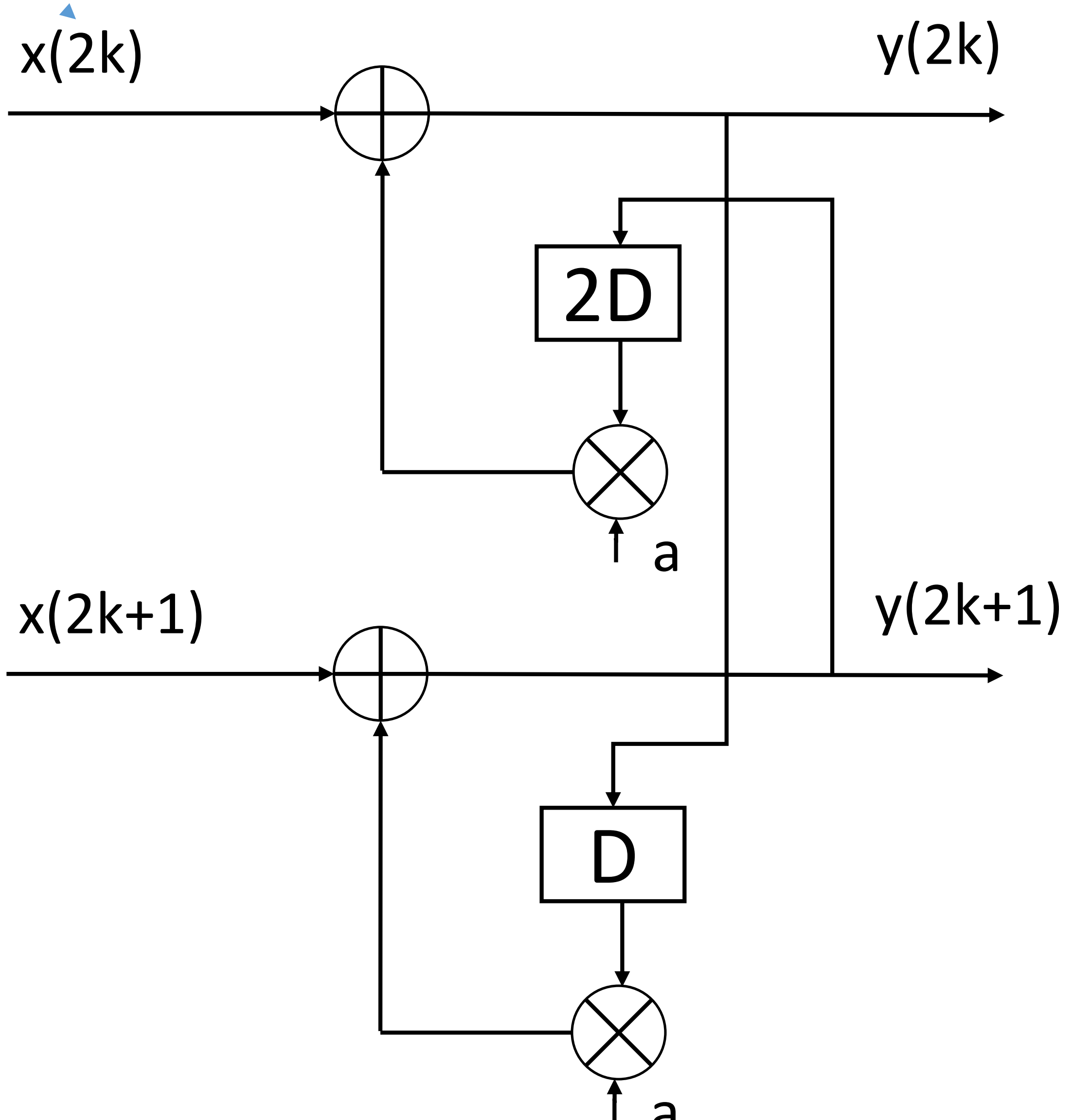
$i = 0$:

$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{3}{2}\right) = 1$$
$$w_0 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{3}{2}\right\rfloor = 1$$

$i = 1$:

$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{4}{2}\right) = 0$$
$$w_1 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{4}{2}\right\rfloor = 2$$

Unfolding: example2



$i = 0$:

$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{3}{2}\right) = 1$$

$$w_0 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{3}{2}\right\rfloor = 1$$

$i = 1$:

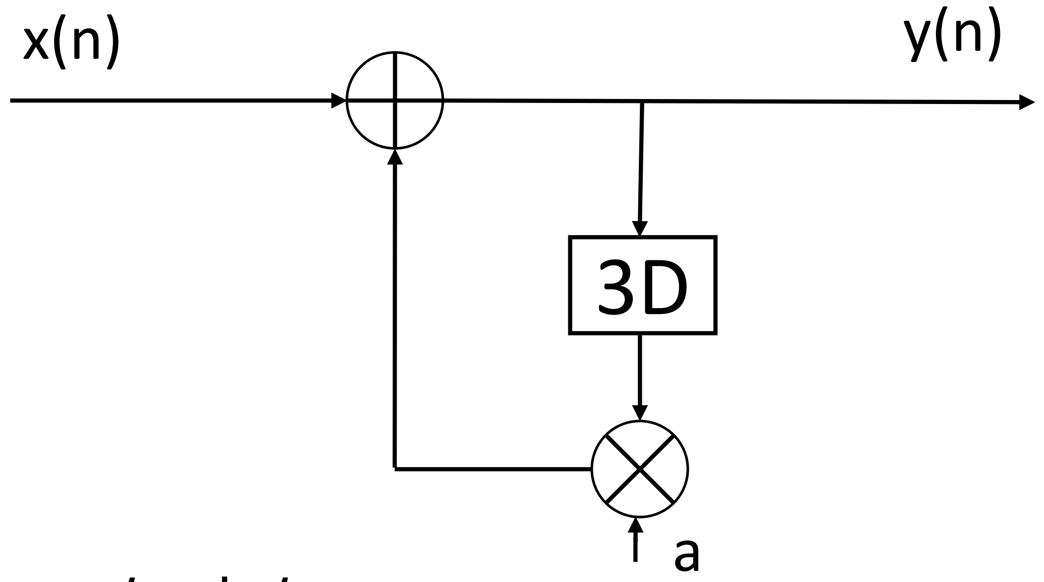
$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{4}{2}\right) = 0$$

$$w_1 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{4}{2}\right\rfloor = 2$$

$$t_{cp} = t_A + t_M$$
$$\text{th} = 2/(t_A + t_M)$$

Unfolding: example2

y(n) = x(n)+a*y(n-3) *P=3*



$t_{cp} = t_A + t_M$

$\text{th} = 1/(t_A + t_M)$

$T_\infty = \frac{(t_A+t_M)}{3} < t_{cp}$

$i = 0$:

$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{3}{3}\right) = 0$$

$$w_0 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{3}{3}\right\rfloor = 1$$

$i = 1$:

$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{4}{3}\right) = 1$$

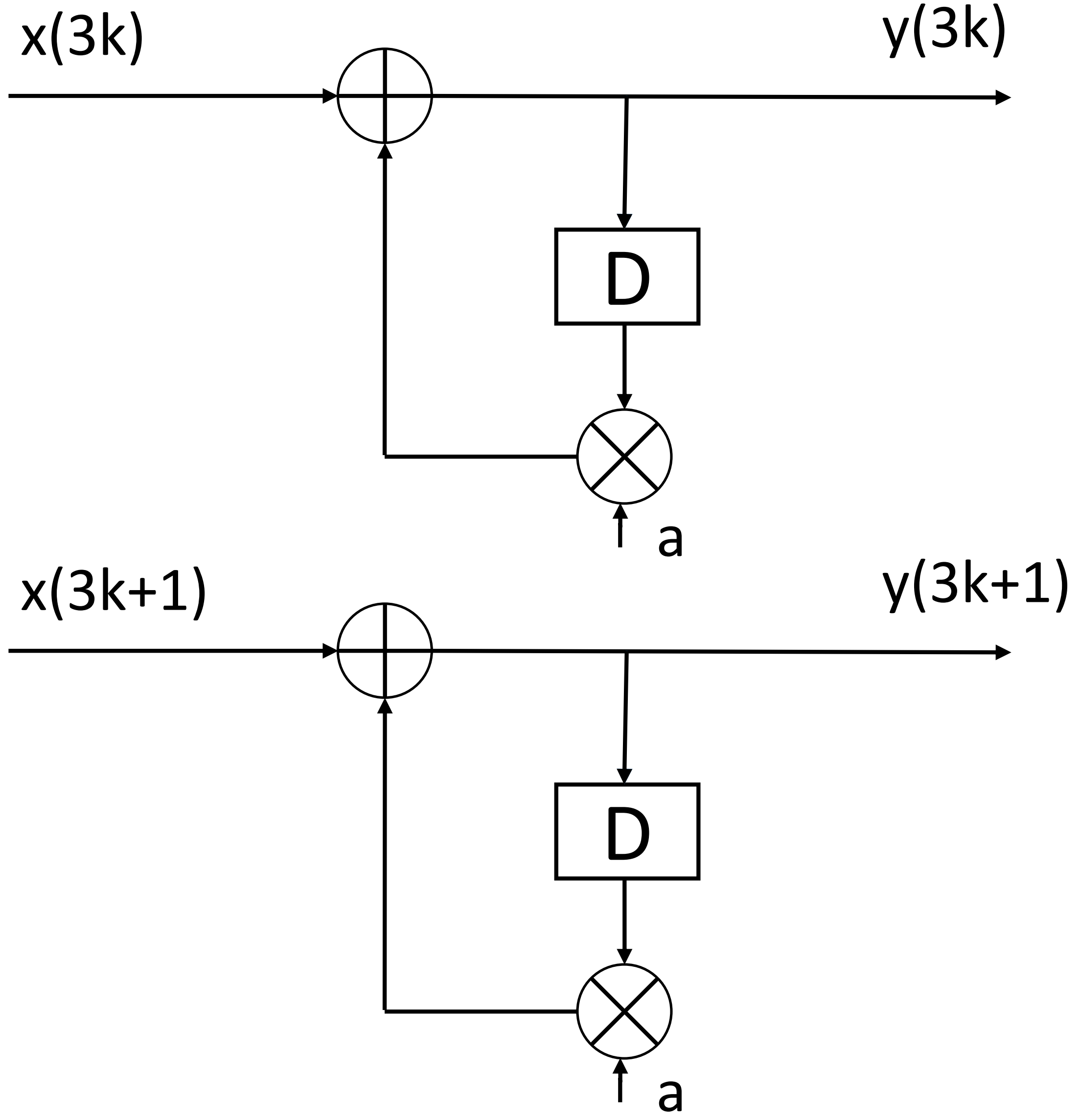
$$w_1 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{4}{3}\right\rfloor = 1$$

$i = 2$:

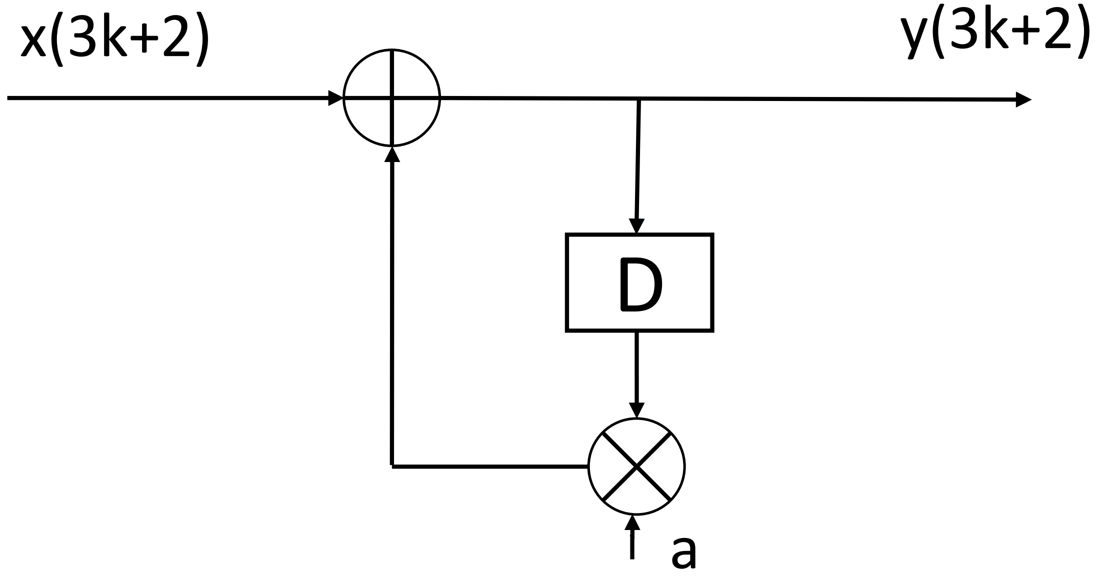
$$j = mod\left(\frac{i+w}{P}\right) = mod\left(\frac{5}{3}\right) = 2$$

$$w_2 = \left\lfloor\frac{i+w}{P}\right\rfloor = \left\lfloor\frac{5}{3}\right\rfloor = 1$$

Unfolding: example2



$j = i \qquad w_i = 1$

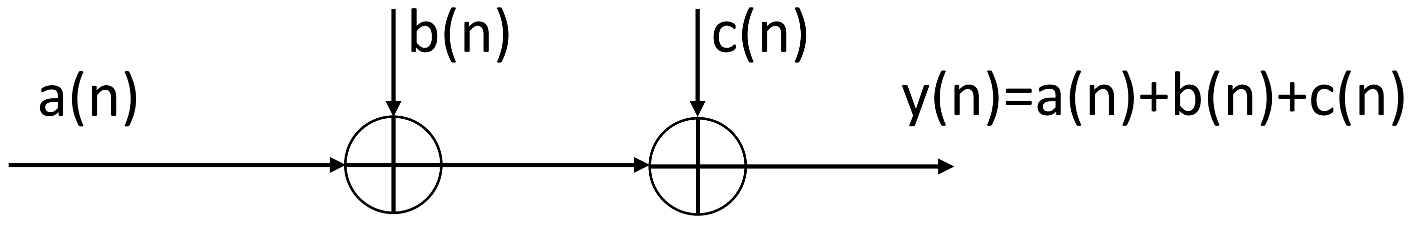


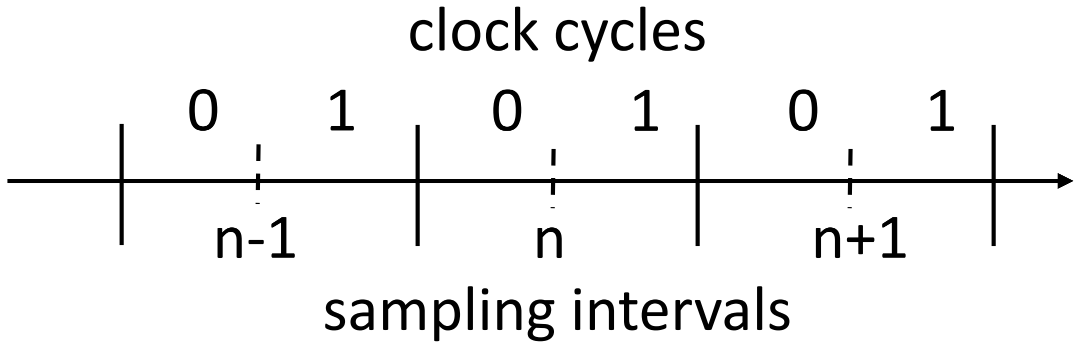
$$t_{cp} = t_A + t_M$$
$$th = 3/(t_A + t_M) = 1/T_\infty!$$

Folding

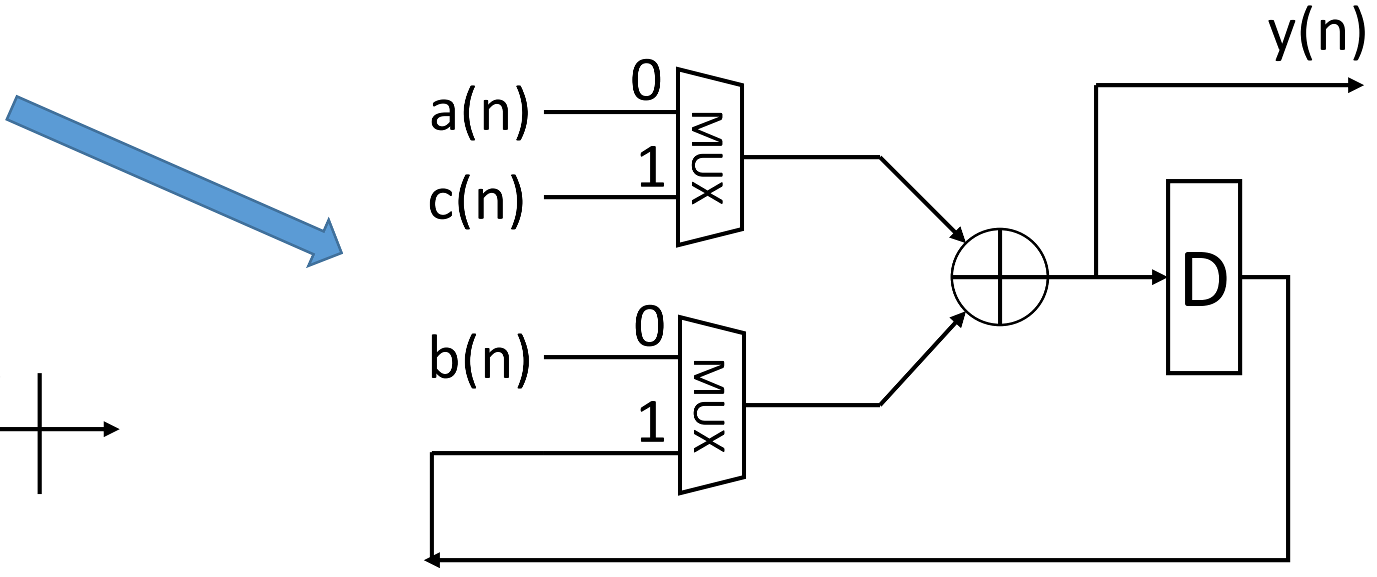
Folding, or time multiplexing: HW resources are shared among multiple operations to reduce implementation cost.

Example:





Folded DFG:



Folding (example)

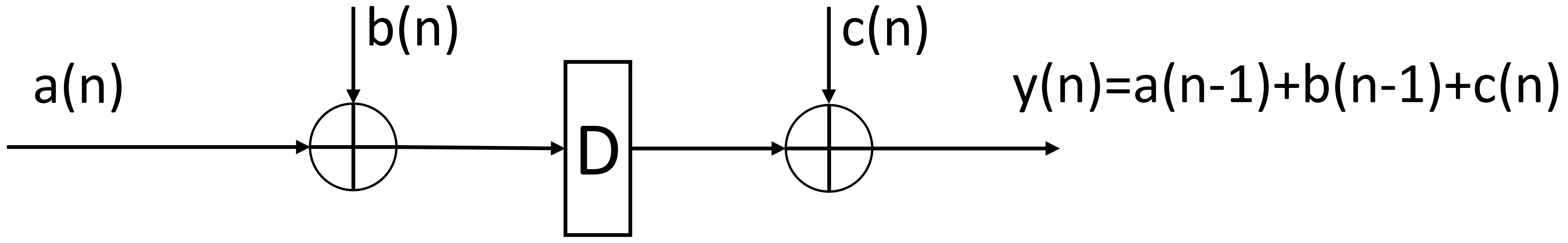
With the original DFG:

- Cost: 2 adders
- Tcp: 2Ta
- throughput: 1/(2Ta)

With the folded DFG:

- Cost: 1 adder (plus muxes)
- Tcp: Ta (plus Tmux)
- throughput: 1/2 * 1/Ta

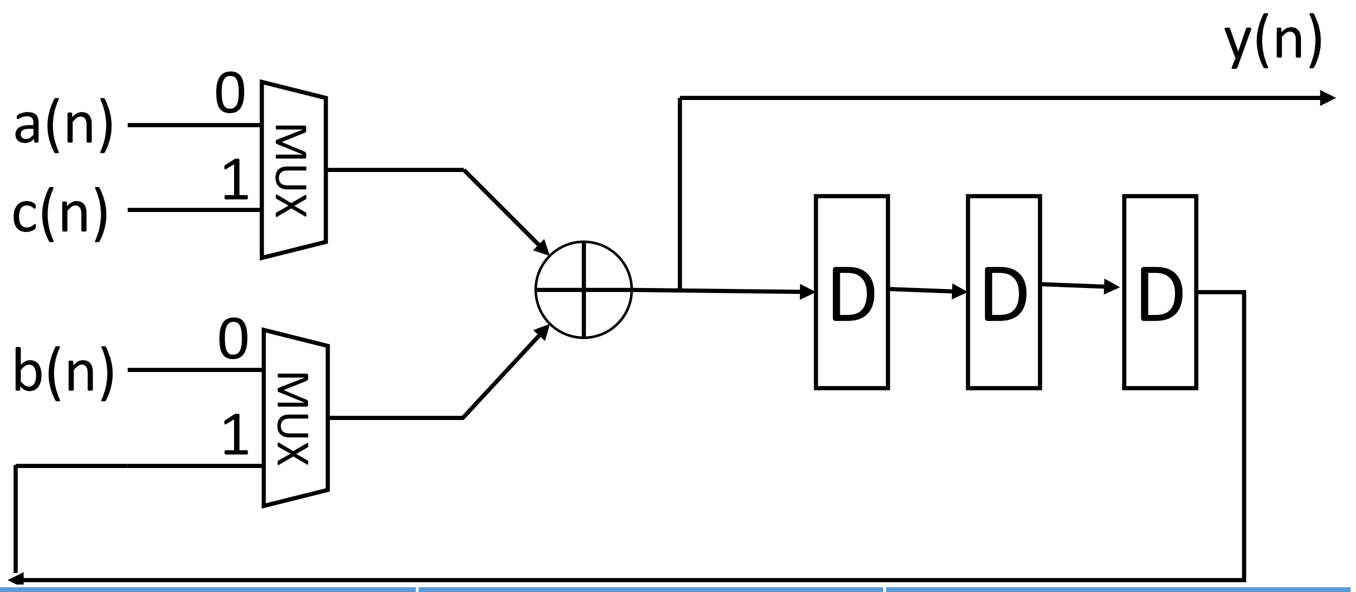
II example:



More registers are necessary in this case to keep correct synchronization of data

Folding (example)

Folded DFG:



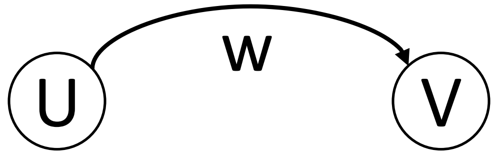
cycle	y(n)	I reg.	II reg.	III reg.
0	a(0)+b(0)	0	0	0
1	c(0)	a(0)+b(0)	0	0
2	a(1)+b(1)	c(0)	a(0)+b(0)	0
3	a(0)+b(0)+c(1)	a(1)+b(1)	c(0)	a(0)+b(0)
4	a(2)+b(2)	a(0)+b(0)+c(1)	a(1)+b(1)	c(0)
5	a(1)+b(1)+c(2)	a(2)+b(2)	a(0)+b(0)+c(1)	a(1)+b(1)

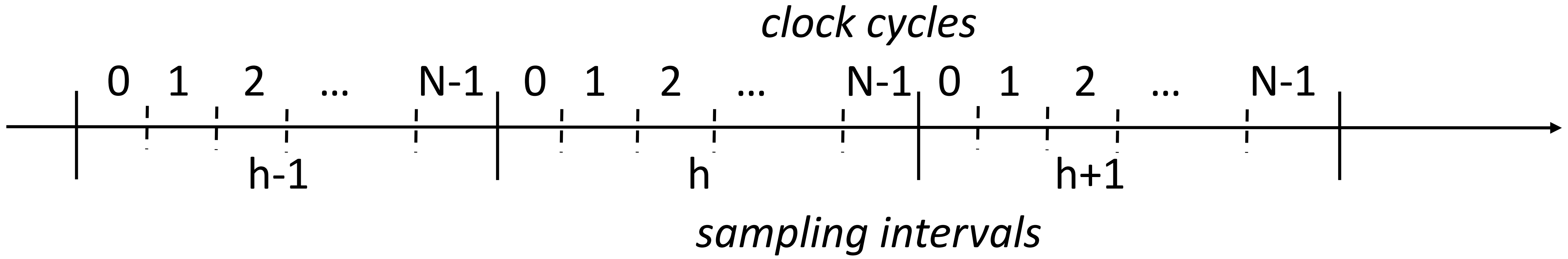
Folding: formal approach

Folding order: N

Scheduling time for node U: u

Scheduling time for node V: v





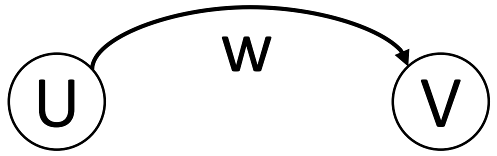
Processing task U is executed by component Hu at time:

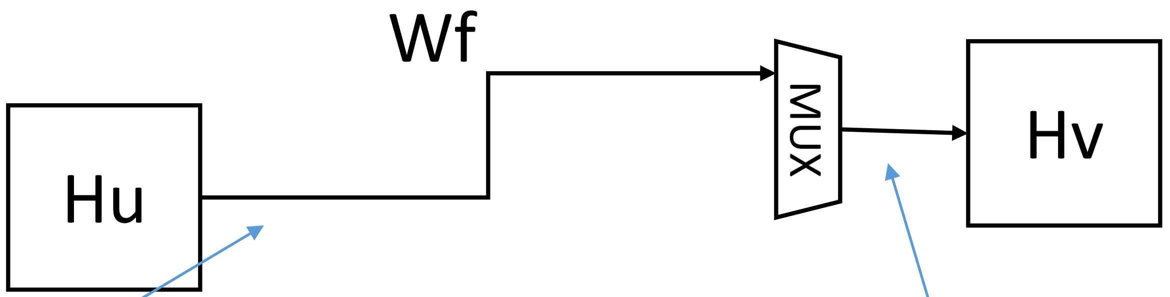
$$N*h + u \qquad 0 \le u \le N-1$$

Processing task V is scheduled to be executed on component Hv at time

$$N*(h+w) + v \qquad 0 \le v \le N-1$$

Folding: formal approach





Output from Hu at: *Nh+u*

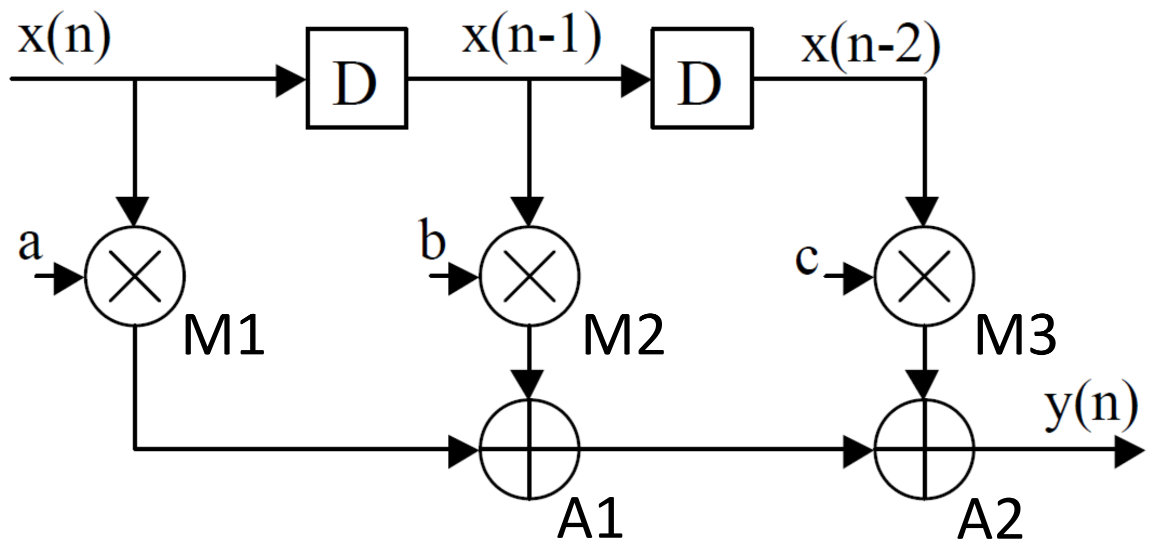
Input to Hv at: *N(h+w)+v*

Number of registers needed along the path:

$$W_f = N(h+w)+v - (Nh+u)$$

$$W_f = Nw + v - u$$

Folding: filter example



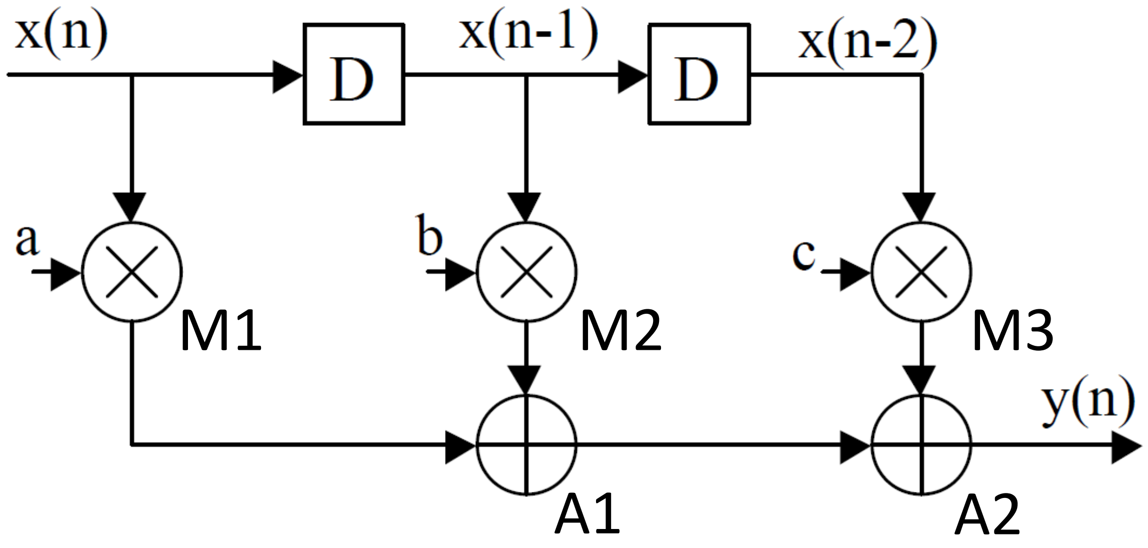
Folding order: N=3
(3 cycles per sampling interval)

Scheduling times:

- M1: 0
- M2: 1
- M3: 2
- A1: 1
- A2: 2

Warning: not all choices of scheduling times are feasible!

Folding: filter example



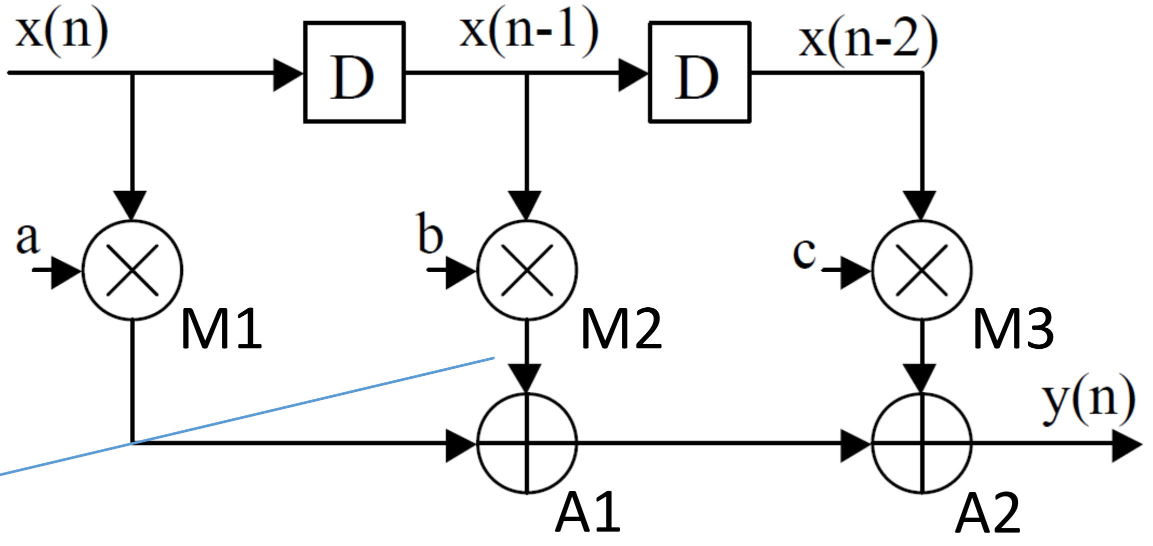
$Wf = Nw + v - u$

$$\left.\begin{array}{ll}
M1\text{->}A1: & Wf = 3*0+1-0 = 1 \\
M2\text{->}A1: & Wf = 3*0+1-1 = 0 \\
M3\text{->}A2: & Wf = 3*0+2-2 = 0 \\
A1\text{->}A2: & Wf = 3*0+2-1 = 1
\end{array}\right.$$

$$\left.\begin{array}{ll}
IN\text{->}M1: & Wf = 3*0+0-0 = 0 \\
IN\text{->}M2: & Wf = 3*1+1-2 = 2 \\
IN\text{->}M3: & Wf = 3*2+2-2 = 6
\end{array}\right\}$$

Assume that input *x* is stable for the whole sampling interval: this implies you can pick the scheduling time that minimizes the number of registers

Folding: filter example

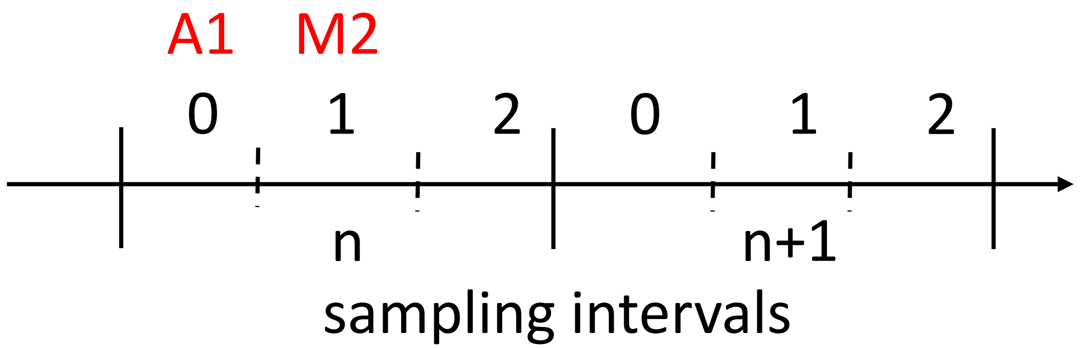


In case of scheduling times that violate data dependencies in the DFG, we can exploit pipelining or retiming to introduce a register and remove the data dependency

Example:

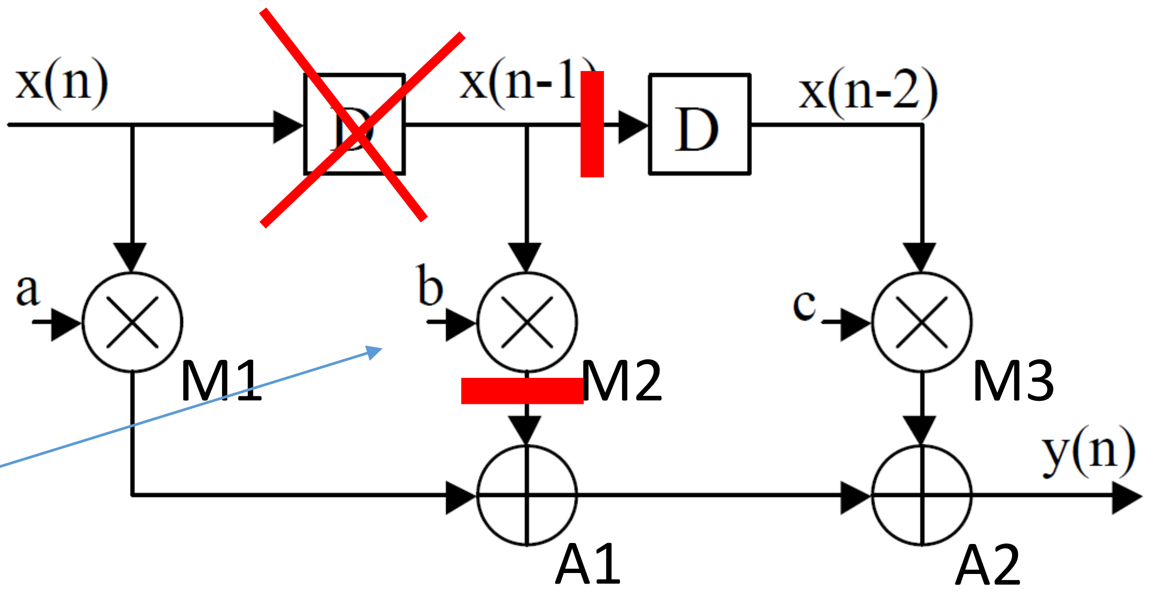
- M1: 0
- M2: 1
- M3: 2
- A1: 0
- A2: 2

Data dependency problem between M2 and A1: scheduling M2 in cycle 1 and A1 in cycle 0 (same sampling interval) is not feasible!



Folding: filter example

The scheduling becomes feasible if M2 and A1 are executed in different sampling intervals, which is obtained for example, by retiming:

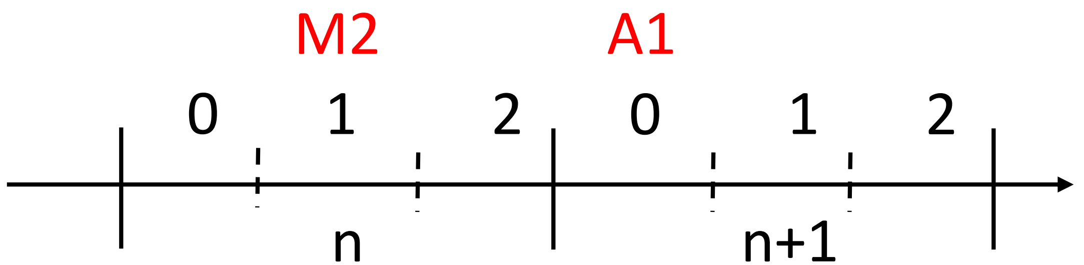


Example:

- M1: 0
- M2: 1
- M3: 2
- A1: 0
- A2: 2

Retiming of node M2: r(M2)=1

M2 and A1 are still scheduled in cycles 1 and 0, but now the two operations are executed in different sampling intervals



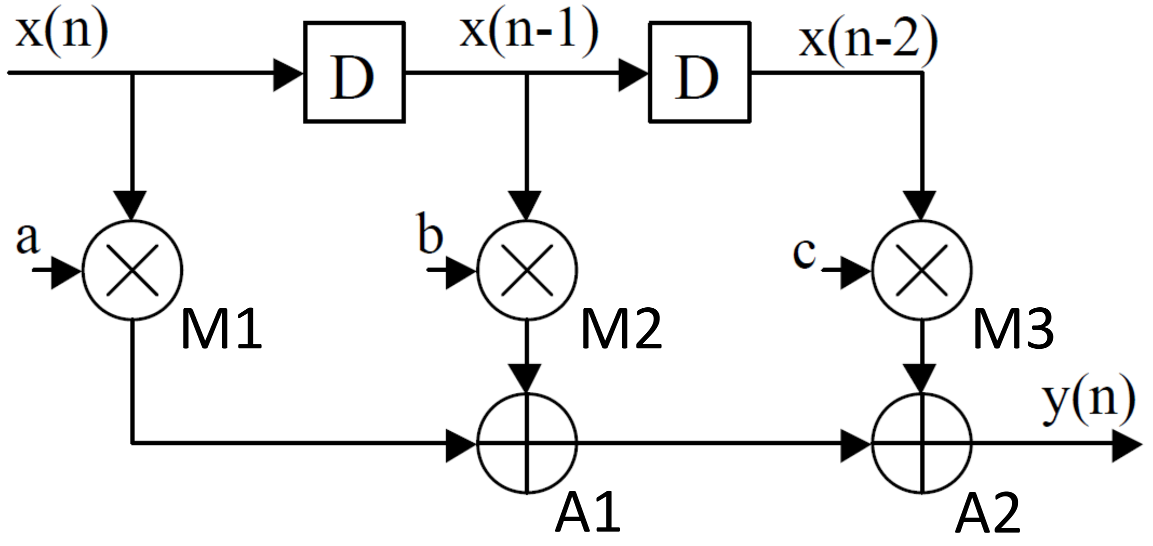
sampling intervals

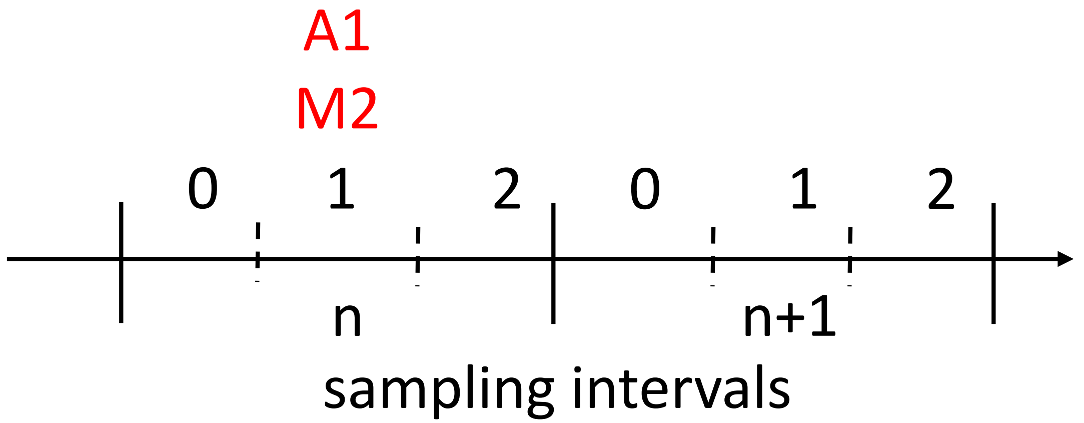
Folding: filter example

Another option to avoid the data dependency problem is the selection of a different scheduling time for A1:

- M1: 0
- M2: 1
- M3: 2
- A1: 1
- A2: 2

M2 and A1executed within the same sampling interval and the same clock cycle -> feasible





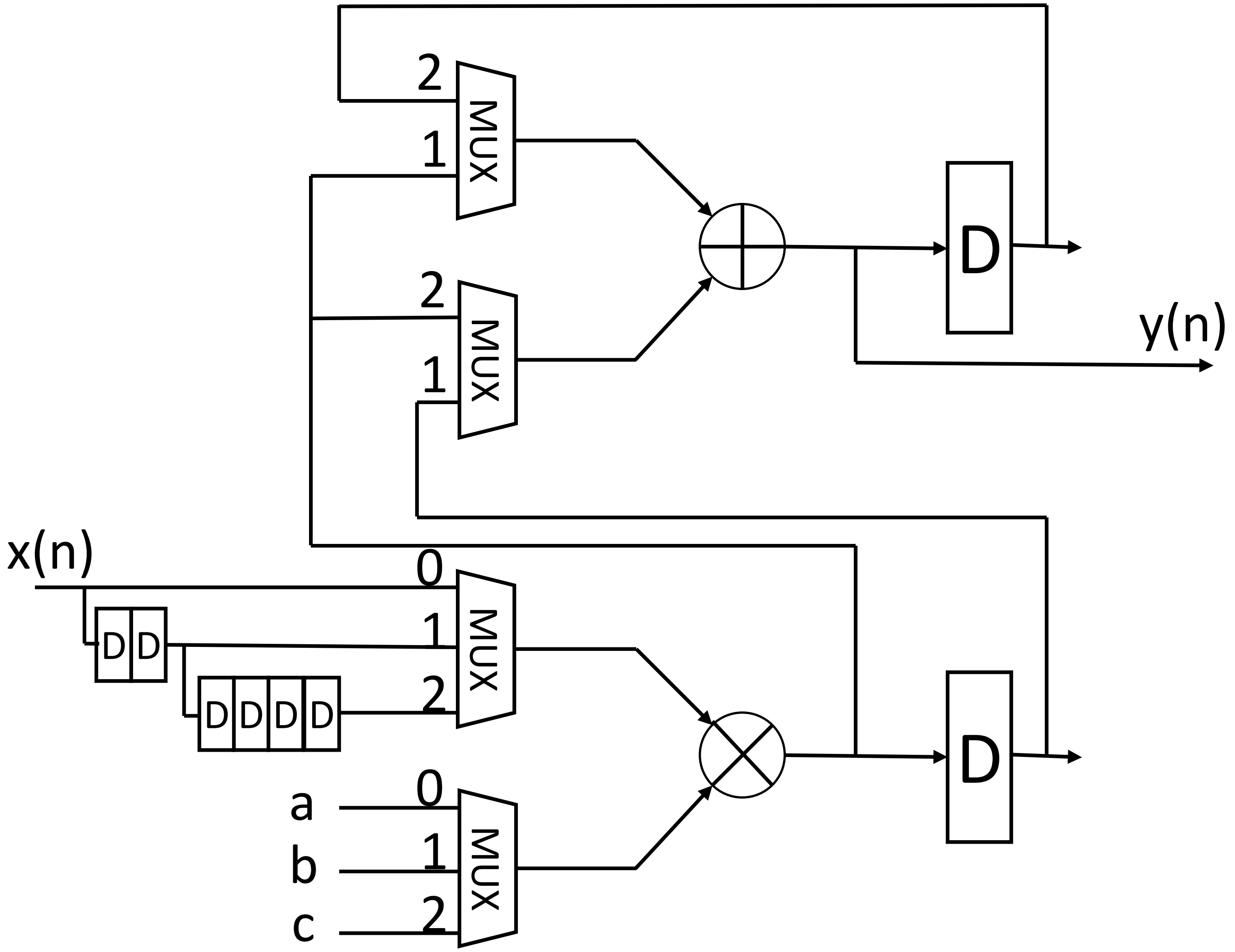
Folded architecture

Folded architecture with feasible scheduling option:

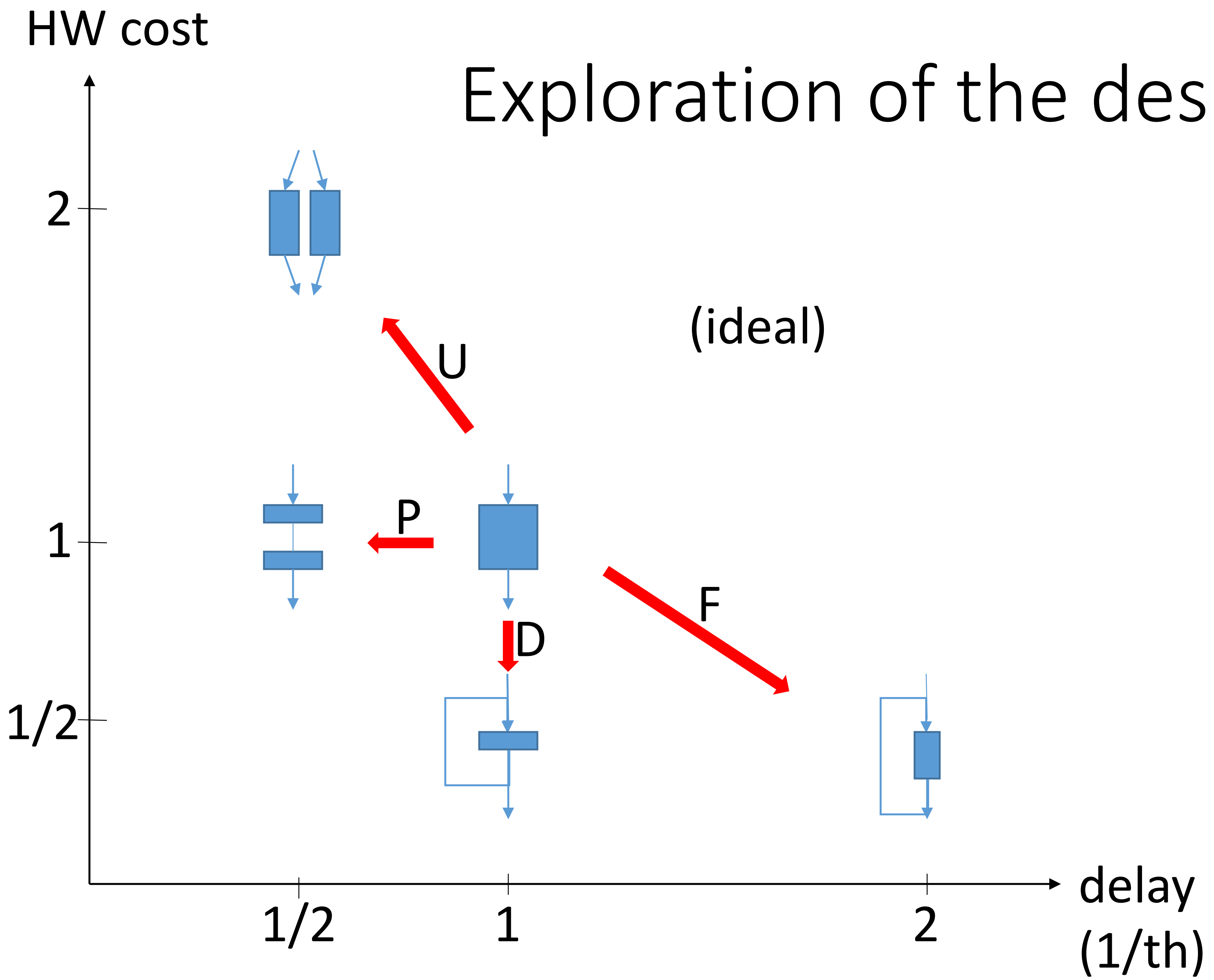
- M1: 0
- M2: 1
- M3: 2
- A1: 1
- A2: 2

Wf = Nw + v – u

M1->A1:	*Wf = 3*0+1-0 = 1*
M2->A1:	*Wf = 3*0+1-1 = 0*
M3->A2:	*Wf = 3*0+2-2 = 0*
A1->A2:	*Wf = 3*0+2-1 = 1*
IN->M1:	*Wf = 3*0+0-0 = 0*
IN->M2:	*Wf = 3*1+1-2 = 2*
IN->M3:	*Wf = 3*2+2-2 = 6*



Exploration of the design space



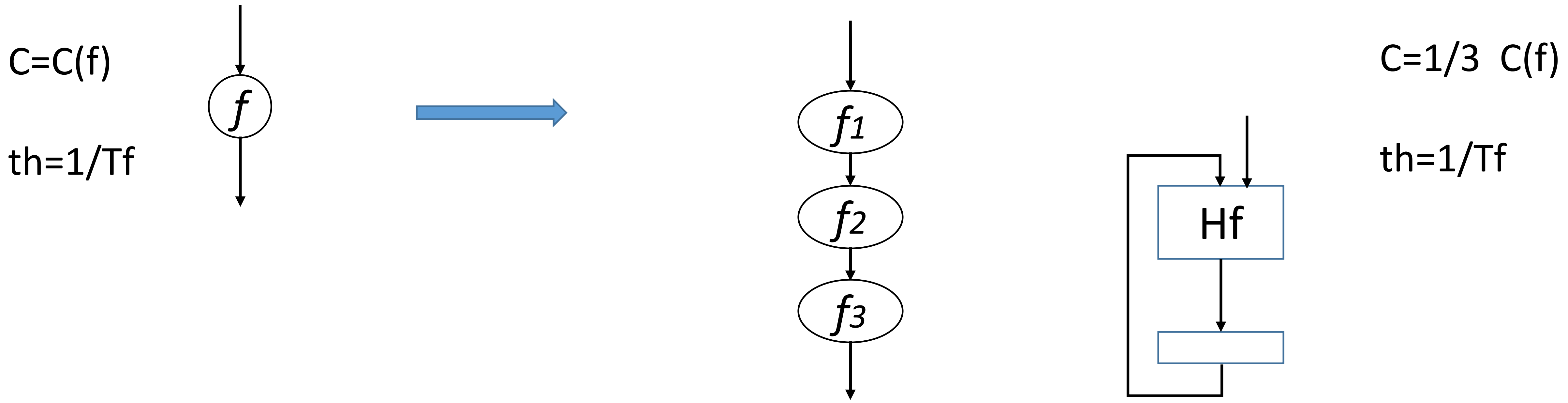
Universal transformations can be applied to any DFG, no matter what operations the nodes stand for

- U: unfolding
- P: pipelining
- F: folding
- D: decomposition

(+ retiming)

Decomposition

Decomposition: splitting a function *f* into a sequence of subfunctions that get executed one after the other on same hardware



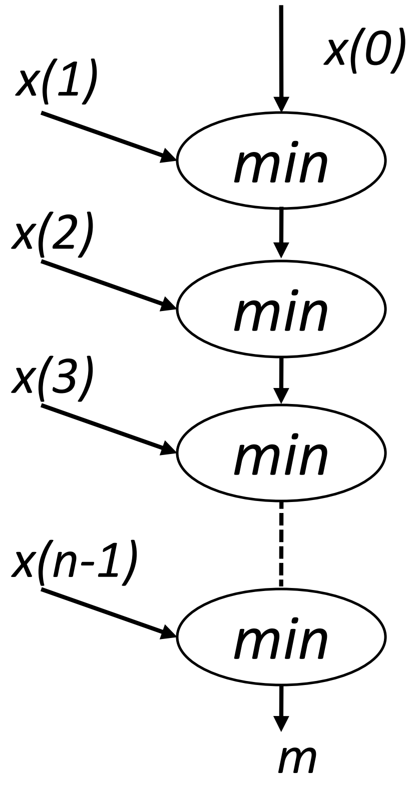
Decomposition is attractive when a computation makes repetitive use of a single subfunction.

Example: multiplication as repeated shift & add operations

Non-universal techniques

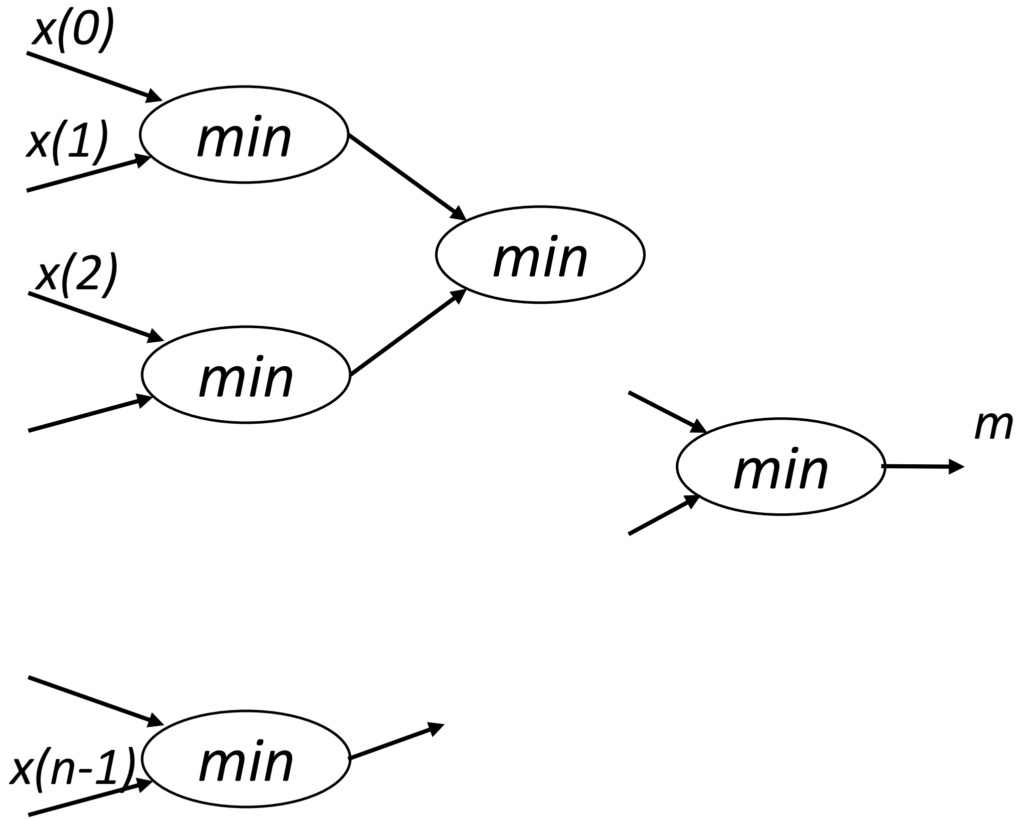
Take advantage of specific properties of the operations involved (e.g. associativity transform, commutativity transform).

Example: *m = min{x(0),x(1),…x(n-1)}*



From linear to tree-like architecture

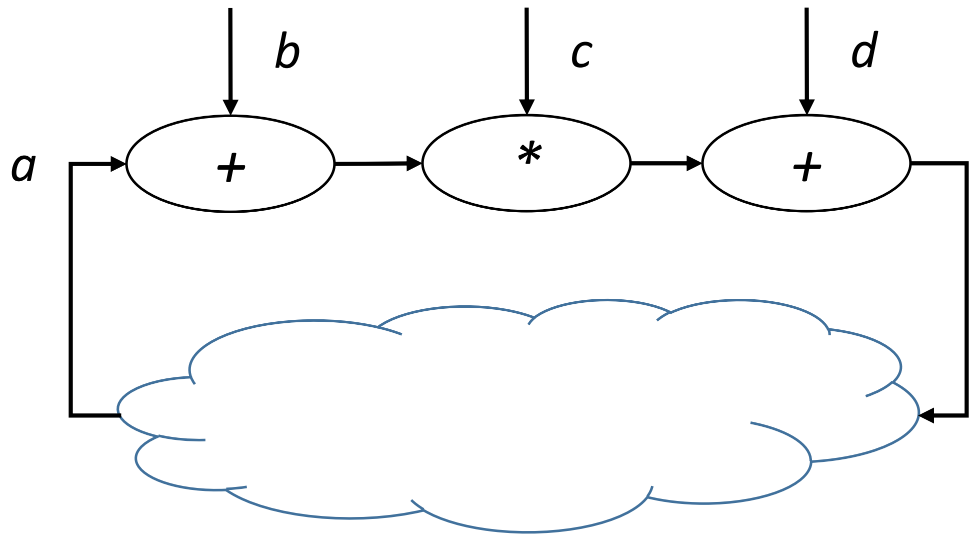
(associative property)



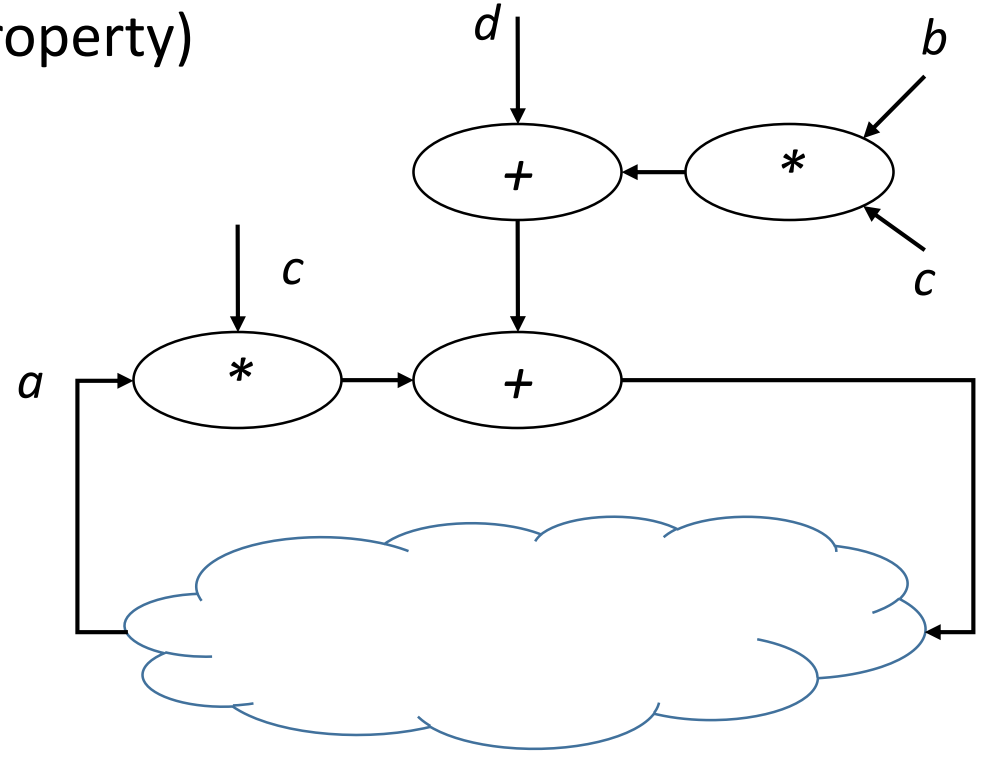
Non-universal techniques

Example: *y = (a+b)*c+d*

*y = a*c + (b*c + d)*

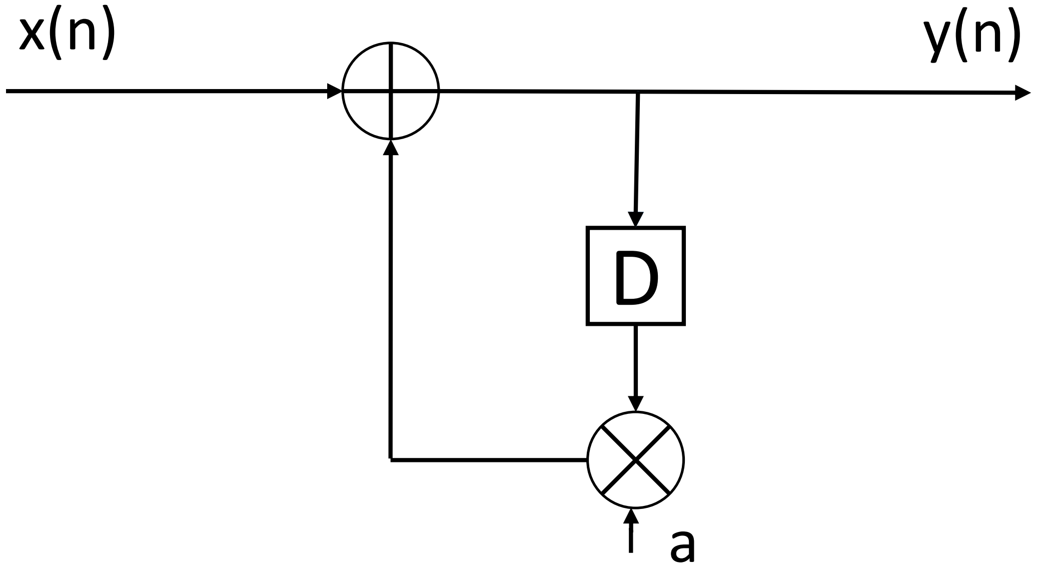


(distributive property)



Look-ahead transform

Example:



y(n) = x(n) + a y(n-1)

Tcp = Ta + Tm

T∞ = Ta + Tm

$\Rightarrow$ No room for improvement by means of universal methods!

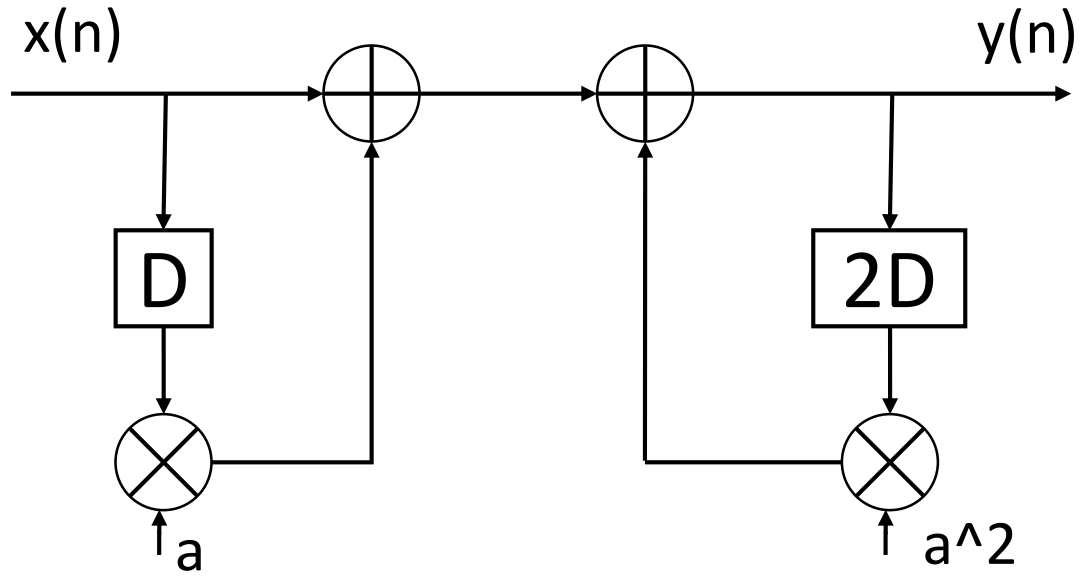
However:

y(n) = x(n) + a y(n-1)

y(n+1) = x(n+1) + a y(n) =

= x(n+1) + a x(n) + a^2 y(n-1)

Look-ahead transform



y(n) = x(n) + a x(n-1) + a^2 y(n-2)

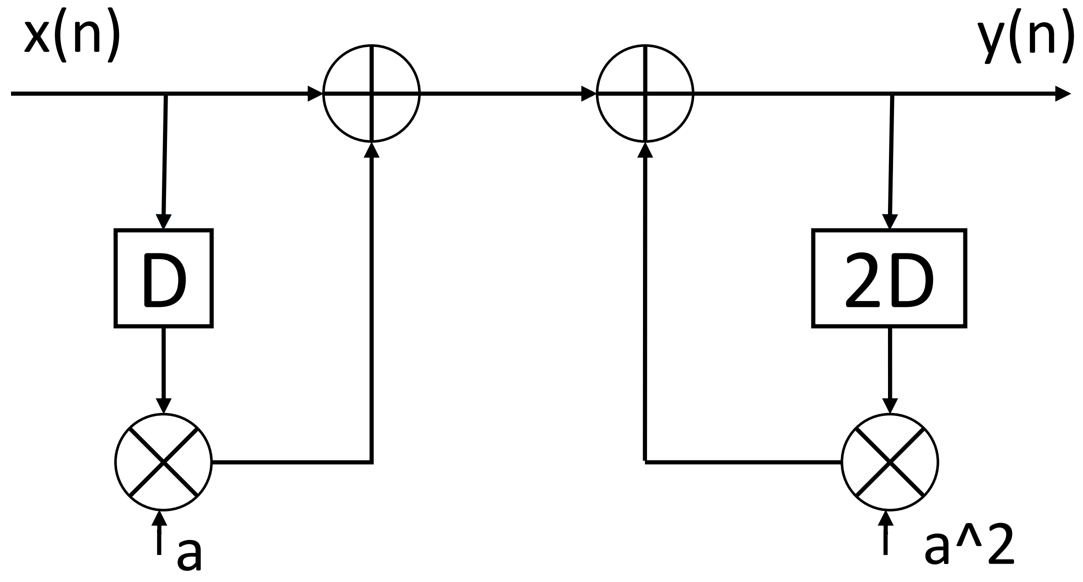
Tcp = 2Ta + Tm

T∞ = (Ta + Tm)/2

⬇

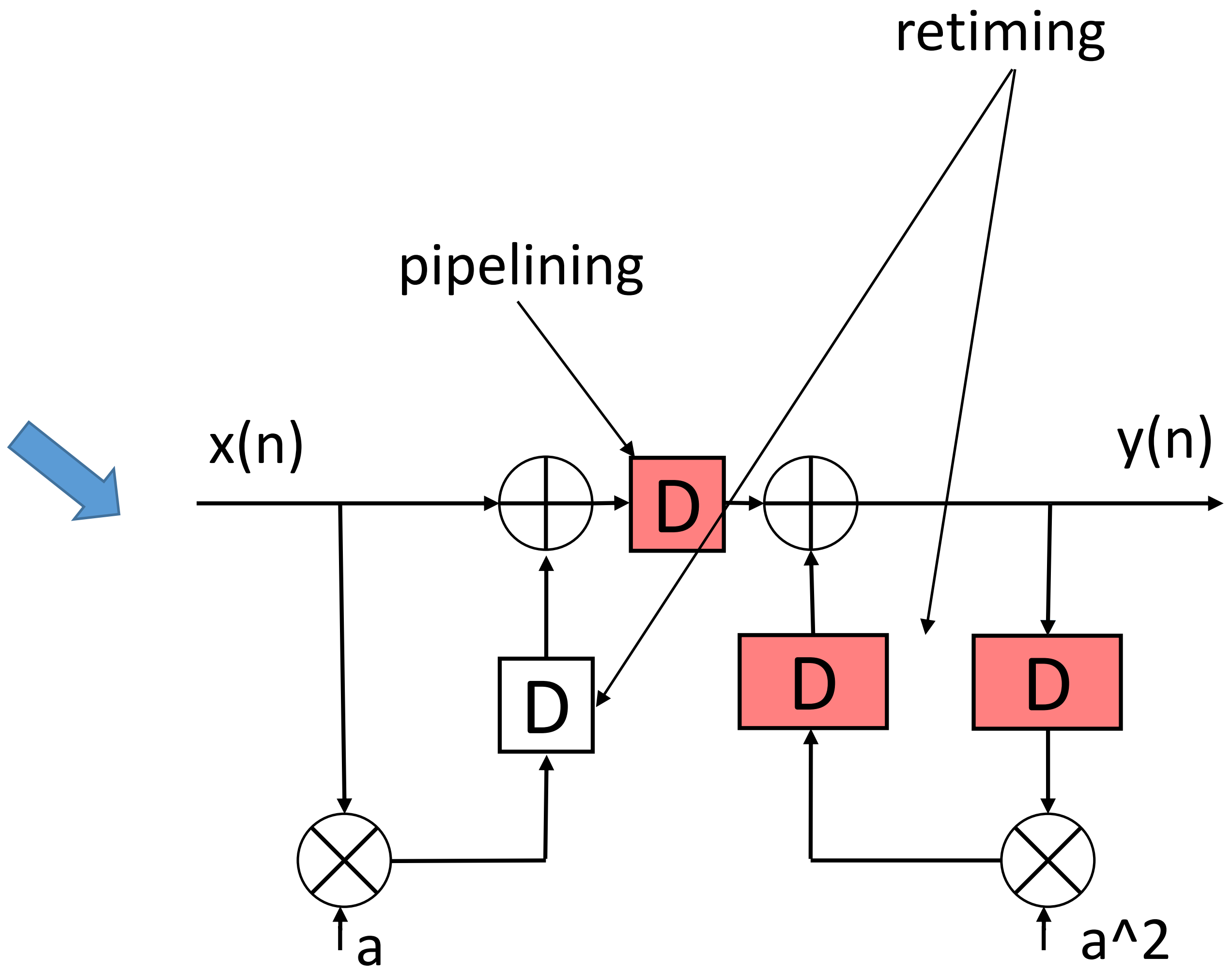
Now, we can increase *th* by means of universal techniques (e.g. retiming and pipelining)

Look-ahead transform



Tcp = Tm

th = 1/Tm



Look-ahead transform

Repeated look-ahead transforms:

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

$y(n+1) = x(n+1) + a\ x(n) + a^2\ y(n-1)$

$y(n+2) = x(n+2) + a\ y(n+1)$

$y(n+2) = x(n+2) + a\ x(n+1) + a^2\ x(n) + a^3 y(n-1)$

$y(n) = x(n) + a\ x(n-1) + a^2\ x(n-2) + a^3 y(n-3)$

3 registers along the feedback loop

$T\infty = (Ta+Tm)/3$

Look-ahead transform

In the z domain:

$$y(n) = x(n) + a\,y(n-1) \qquad \rightarrow \qquad Y(z) = X(z) + a\,Y(z)z^{-1}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - a\,z^{-1}}$$

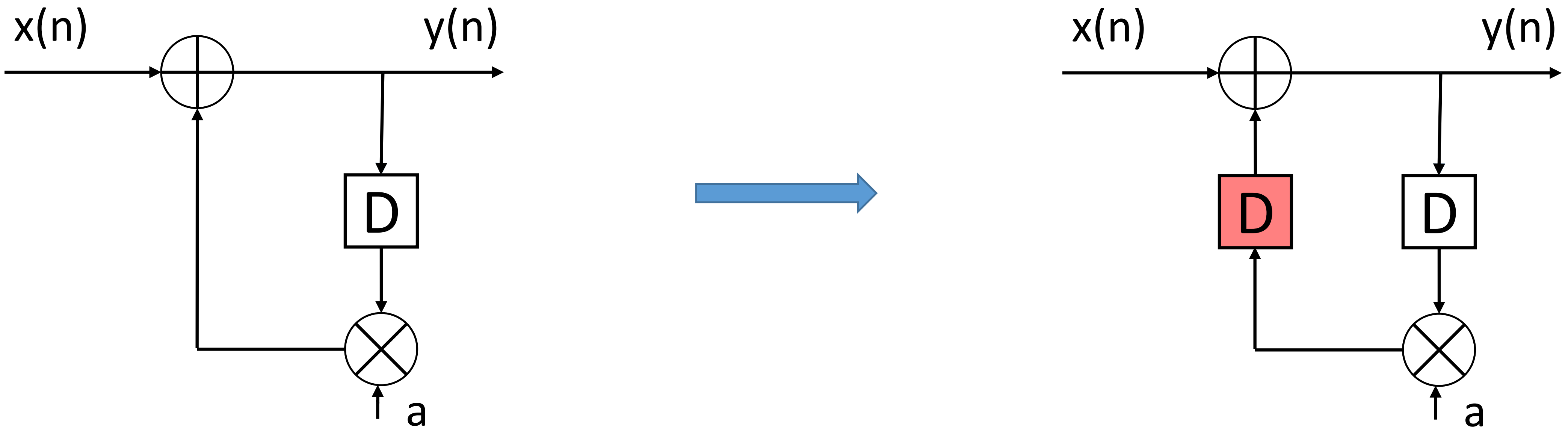
$$H(z) = \frac{1}{1 - a\,z^{-1}} \cdot \frac{1 + a\,z^{-1}}{1 + a\,z^{-1}} = \frac{1 + a\,z^{-1}}{1 - a^2\,z^{-2}}$$

Non recursive part

two registers along the feedback loop

Pipeline interleaving

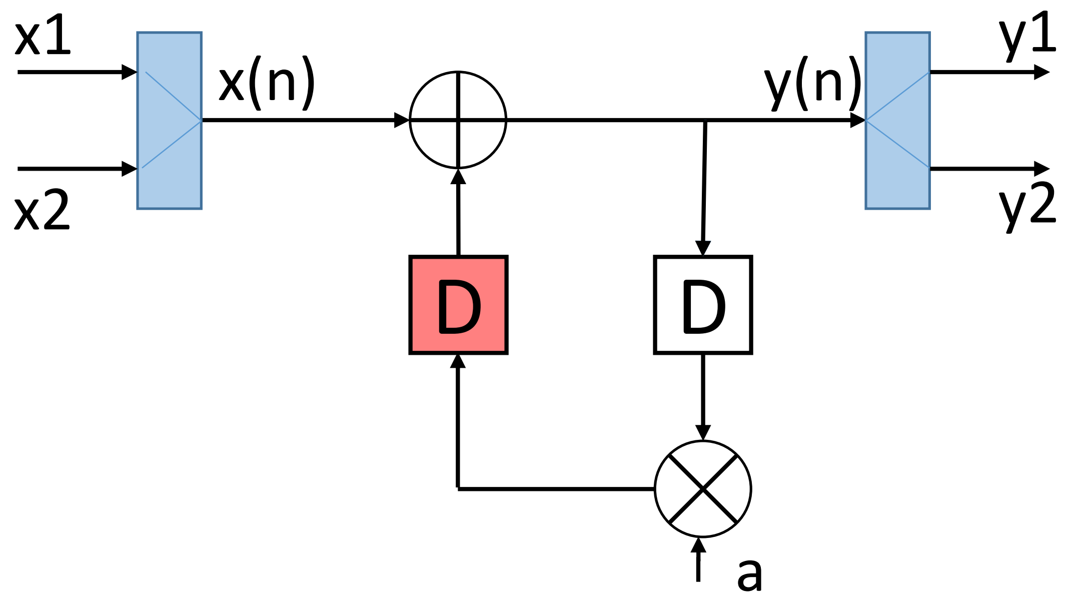
Example:



Additional register along the loop:

- the critical path is shorter, but…
- the behavior is changed!

Pipeline interleaving



Two streams of samples are processed in an interleaved way

- x1(k)=x(2k) y1(k)=y(2k)
- x2(k)=x(2k+1) y2(k)=y(2k+1)

y(n)=x(n)+a y(n-2) -> y(2k)=x(2k)+a y(2k-2) -> y1(k)=x1(k)+ay1(k-1)

y(n)=x(n)+a y(n-2) -> y(2k+1)=x(2k+1)+a y(2k-1) -> y2(k)=x2(k)+ay2(k-1)

References and readings

- K. K. Parhi, “VLSI Digital Signal Processing”, John Wiley & Sons Inc, 2015, chapters 1 to 6
- Hubert Kaeslin, “Top-Down Digital VLSI Design”, Morgan Kaufmann, 2014, chapter 3
- G. Fettweis , and H. Meyr, “Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck”, IEEE Transactions on Communications, 1989, Vol. 37, issue 8
- A.V. Oppenheim, and R.W Schafer, “Discrete-time Signal Processing”, Prentice-Hall

Acknowledgements

These slides contain material developed and copyright by Prof. K.K. Parhi (UMN) and Prof. H. Kaeslin (ETH)