Part 4
Algorithms and Architectures for VLSI

- Algorithm strength reduction
- Pipelining
- Retiming
- Pipelined recursive filters
- Parallel processing
- Unfolding
- Parallel recursive filters
- Folding

Algorithm Strength Reduction

Strength Reduction

Strong operations = computationally expensive operations.
Weak operations = computationally inexpensive operations.

Strength reduction = replacing strong operations by weak operations.

In VLSI, strength reduction increases the throughput, or reduces the area, or reduces the power consumption.

Fast Fourier Transform

Discrete Fourier transform (DFT):
$x(n)$ is defined for $n = 0,1,\ldots, N-1$. $X(z) = \sum_{n=0}^{N-1} x(n)z^{-n}$

DFT is defined as
$$X(k) = X(z)|_{z=e^{j2\pi k/N}} = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}$$
for $k = 0,1,\ldots, N-1$.

Direct computation of DFT requires N^2 complex multiply and $N(N-1)$ complex add.

Fast Fourier transform (FFT):

Let *N* be a power of 2. Using 2-phase polyphase components $X(z) = X_0(z^2) + z^{-1}X_1(z^2)$,

$$X(k) = \sum_{n=0}^{N/2-1} x_0(n)e^{-j4\pi kn/N} + e^{-j2\pi k/N} \sum_{n=0}^{N/2-1} x_1(n)e^{-j4\pi kn/N}$$

where $x_0(n) = x(2n), x_1(n) = x(2n+1)$.

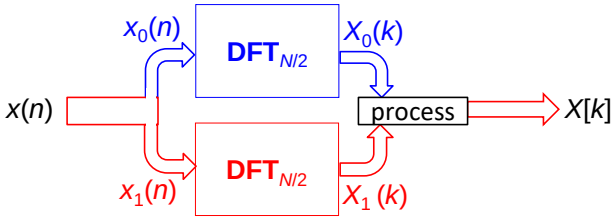
But $\sum_{n=0}^{N/2-1} x_0(n)e^{-j2\pi kn/\left(\frac{N}{2}\right)} = X_0(k)$, the *N*/2 point DFT of $x_0(n)$.

(Note that DFT coefficients *X*(*k*) are periodic.)

Similarly, $\sum_{n=0}^{N/2-1} x_1(n)e^{-j2\pi kn/\left(\frac{N}{2}\right)} = X_1(k)$, the *N*/2 point DFT of $x_1(n)$.

Therefore: $X(k) = X_0(k) + e^{-\frac{j2\pi k}{N}} X_1(k)$

This leads to a divide and conquer approach known as the decimation-in-time FFT:



Each *N*/2 point DFT requires $N^2/4$ complex multiply and $N(N-2)/4$ complex add. The "process" requires *N* complex multiply and *N* complex add. Overall, the structure needs $N(N+2)/2$ complex multiply and $N^2/2$ complex add.

Recursion: Compute each *N*/2 point DFT using two *N*/4 point DFTs, and so forth. FFT requires about $(N/2)\log N$ complex multiply and $N\log N$ complex add.

Complex Multiplication

Multiply a given number *a+jb* with an input *A+jB* :

$$(a+jb)(A+jB) = (aA - bB) + j(aB + bA)$$

4 real multiply

2 real add

Rewrite it as:

$$(aA - bB + bA - bA) + j(aB + bA + bB - bB)$$
$$= \{(a+b)A - b(A+B)\} + j\{b(A+B) + (a-b)B\}$$

Let $c = a + b, d = a - b$ be precomputed, then:

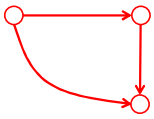
$$= \{cA - b(A+B)\} + j\{b(A+B) + dB\}$$

3 real multiply

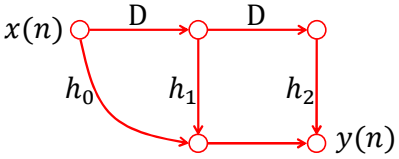
3 real add

Data Flow Graph

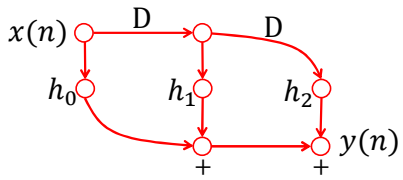
(Directed) graph: nodes and directed edges



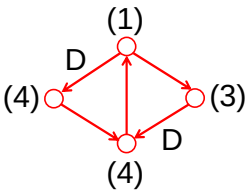
Signal flow graph:
Nodes = interconnections such as many to one (add), one to many
Directed edges = transformations such as multiply, delay



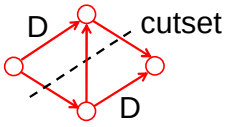
Data flow graph:
Nodes = computations such as add, multiply
Directed edges = data path between nodes, including any delay



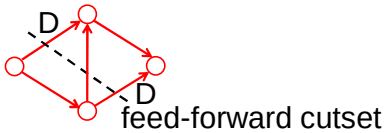
In a data flow graph, nodes are labelled by computation time, and edges are labelled by no of delays.

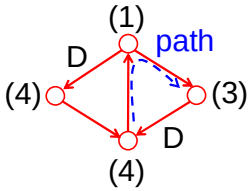


Cutset: set of edges which, if removed, makes the graph disjoint



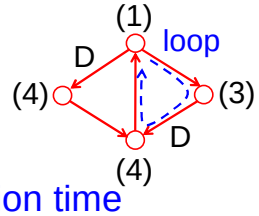
Feed-forward cutset: cutset with all forward direction edges





A path of a graph begins at some node and ends at any node.

A loop of a graph begins and ends at the same node.

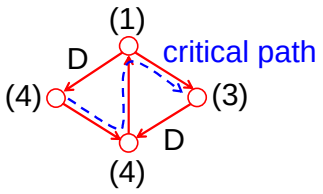


Loop bound of a loop = $\frac{\text{loop computation time}}{\text{no of delays in the loop}}$

Loop bound of the above loop = (1+3+4)/(0+1+0) = 8

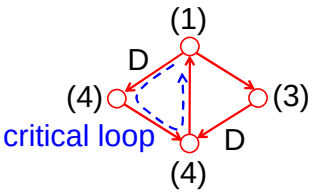
Critical path of a graph is the longest computation time among all zero delay paths.
Clock cycle is lower bounded by the critical path.

Critical path of the above graph = 4+4+1+3 = 12



Iteration bound is the maximum loop bound of a graph.
Critical loop is the loop with the maximum loop bound.

In this graph:
Iteration bound = 9
Critical loop = the left loop

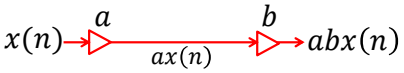


Pipelining

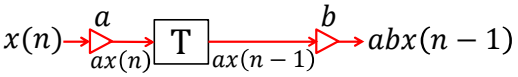
Introduction

Pipelining = idea of a water pipe, continue pumping in the water without waiting for the water in the pipe to come out

Critical path = maximum processing time taken by a path without any latch



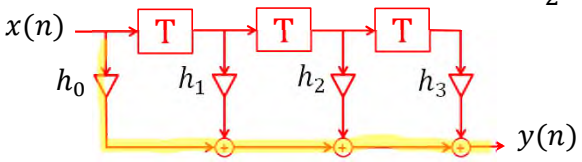
T_M = multiplication time
Critical path = $2T_M$
Sampling period $T_s \geq 2T_M$



Insert a latch in between.
Critical path = T_M
Sampling frequency is doubled.

Pipelining reduces the critical path and therefore increases the clock/sampling speed

Consider a length 4 FIR filter: $y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + h_3x(n-3)$



Critical path = $T_M + 3T_A$ (where T_A = addition time)

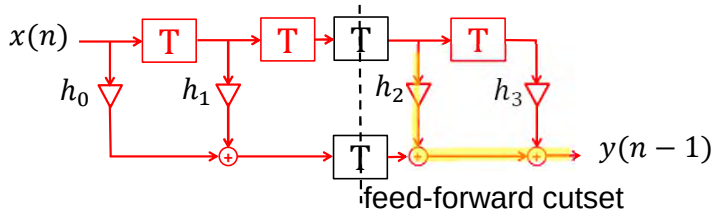
Latency and Throughput

Latency = number of clock periods between the output sample and the corresponding input sample

Throughput = number of input samples/clock period

The sequential filter has latency = 1 clock (assuming critical path ≈ T_S), throughput = 1 input/clock

Pipelining option 1: insert latches at a feed-forward cutset



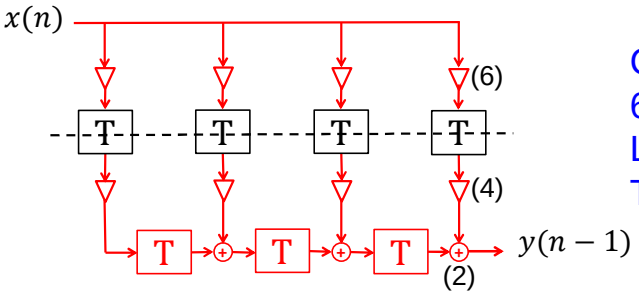
Critical path is reduced to $T_M + 2T_A$.

Pipelining increases the latency.
Each output is computed using 2 clock periods.
So, this pipelined filter has latency = 2 clocks.

Pipelining may reduce the throughput.
This pipelined filter has throughput = 1 input/clock

Pipelining option 2: fine-grain pipelining
Let T_M = 10 gate delays, T_A = 2 gate delays

To achieve a critical path less than T_M , break the multiplier into two parts with processing times of 6 gate delays and 4 gate delays.



Critical path = 6 gate delays
Latency = 2
Throughput = 1

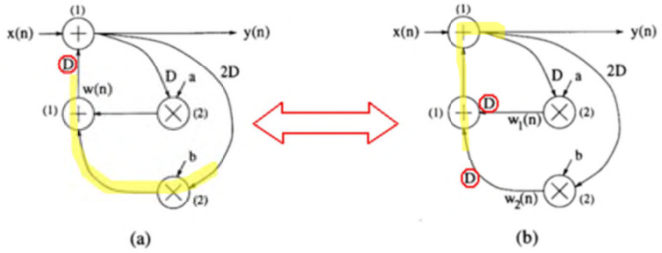
Pipelining reduces critical path by increasing latency.
We now study techniques such as retiming that reduces critical path without increasing latency.

Retiming

Retiming

Retiming = moving around delays without changing the functionality of a given system.

Retiming is done to increase the throughput, or reduce the area/power consumption/no of registers.



Example: Critical path of (a) = 2 + 1 = 3

Move the marked delay before the adder.

Critical path of (b) = 2 or 1 + 1 = 2

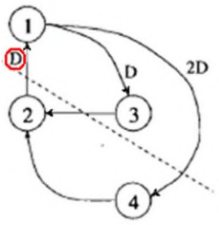
(a) and (b) have same input/output relationships.

(b) has a smaller clock cycle.

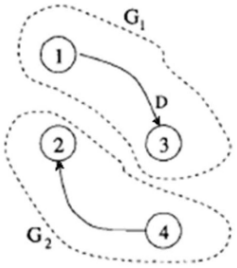
Cutset Retiming

Cutset retiming is a systematic way to achieve retiming.

Recall that a cutset is a set of edges which, when removed, makes a graph *G* disjoint. In this example, the 3 edges in the cutset are 2→1, 3→2, and 1→4.



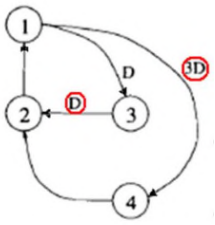
Removing the cutset makes two disjointed subgraphs G_1 and G_2.



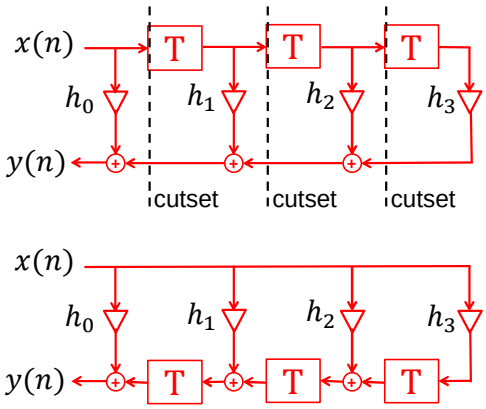
Cutset retiming consists of adding *k* delays to each edge from G_1 to G_2, and removing *k* delays from each edge from G_2 to G_1.

Choose *k* such that no retimed edge has a negative delay.

k=1 delay is added to the edges 3→2 and 1→4, and 1 delay is removed from the edge 2→1.



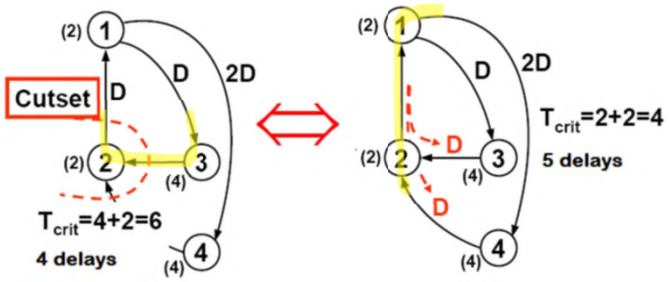
Pipelining option 3: retiming
reduce the critical path using transposed form FIR structure



Critical path = T_M + T_A , independent of the length of the filter
This is better than other options, because latency is not increased, and it does not require additional latches.

Node Retiming

Consider a cutset around a node.

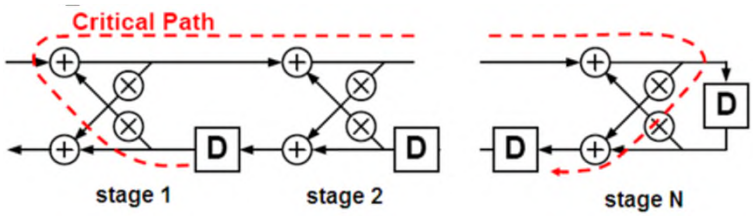


k=1 delay is added to the edges 3→2 and 4→2, and 1 delay is removed from the edge 2→1, to obtain the retimed graph.

Critical path before retiming = 4 + 2 = 6
Critical path after retiming = 4 or 2 + 2 = 4
However, no of delays increases from 4 to 5.

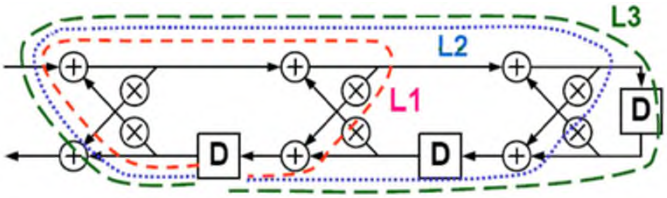
Cutset Retiming in Lattice Filter

Consider an *N*-stage lattice filter.



Critical path = $(N+1)T_A + 2T_M$

Loop bound of loops L1, L2, and L3: $T_{L1} = \dfrac{3T_A + 2T_M}{1}$



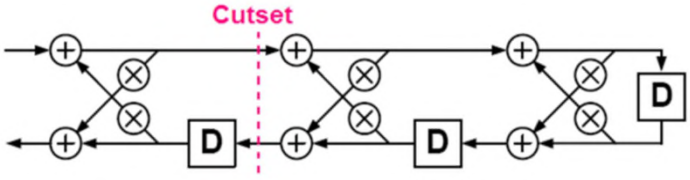
$$T_{L2} = \frac{5T_A + 2T_M}{2}$$

$$T_{L3} = \frac{5T_A + T_M}{3}$$

Find the iteration bound and the critical loop:

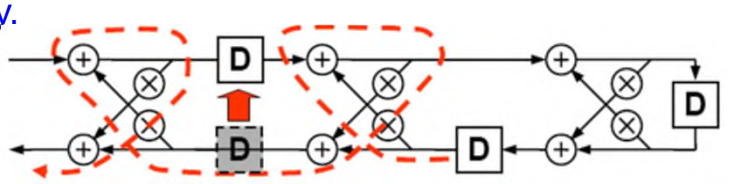
$$\max\{T_{L1}, T_{L2}, T_{L3}, \ldots\} = \frac{3T_A + 2T_M}{1}$$ for loop L1

Cut the critical loop (loop L1):



Repeat the above for remaining loops.
(Cut every alternate loop L3, L5, etc.)

Cutset retiming for the loop L1:
Add a delay on the top edge, remove the bottom edge delay.



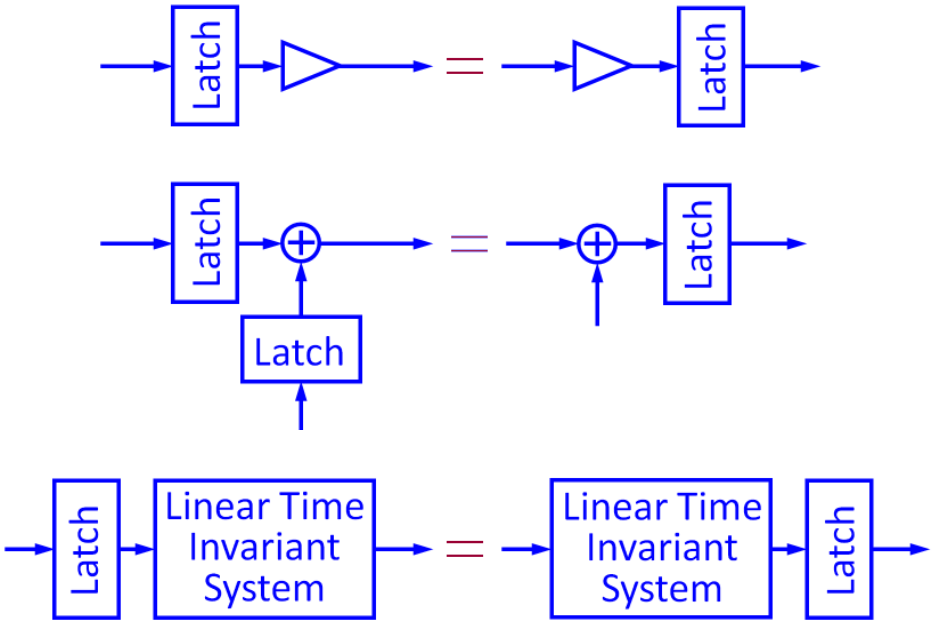
Critical path = $4T_A + 4T_M$

After repeated cutset retiming is applied, this is the critical path for any lattice filter independent of its number of stages.

For large T_M, retiming is better only for large N.
For example, let $T_M = 2T_A$.
Critical path before retiming = $(N+1)T_A+2T_M = (N+5)T_A$
Critical path after retiming = $4T_A+4T_M = 12T_A$
So, retiming is better only if $N > 7$.

Pipelined Recursive Filters

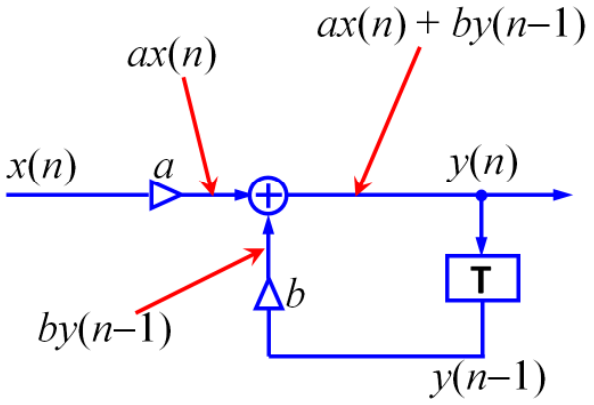
A latch may be interchanged with a processing stage:



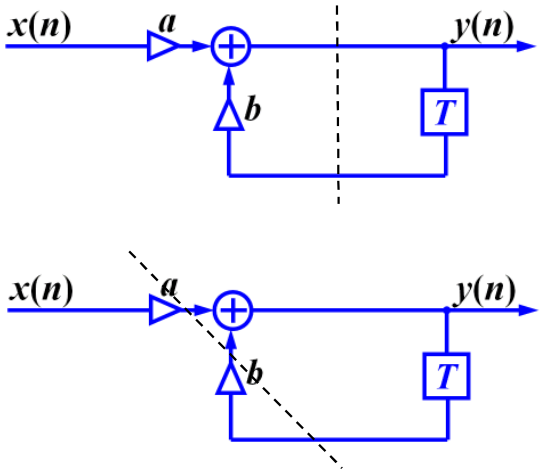
First Order Filters

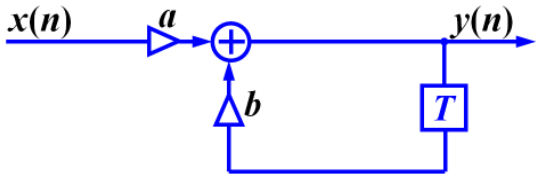
Consider the implementation of the recursive filter

$$y(n) = ax(n) + by(n-1)$$

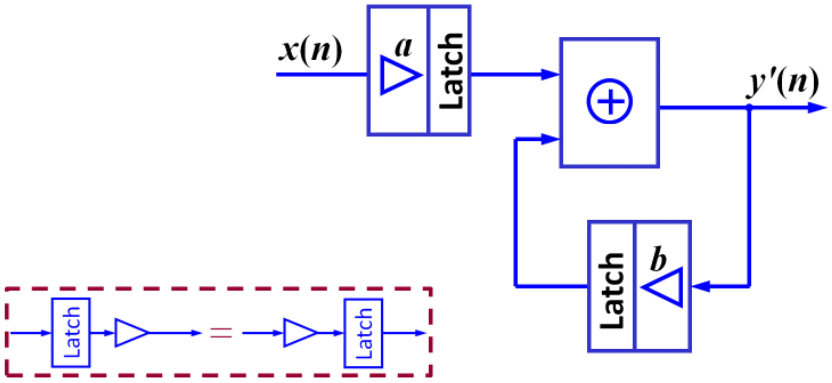


Due to the feedback loop, the following cutsets are not feed-forward:

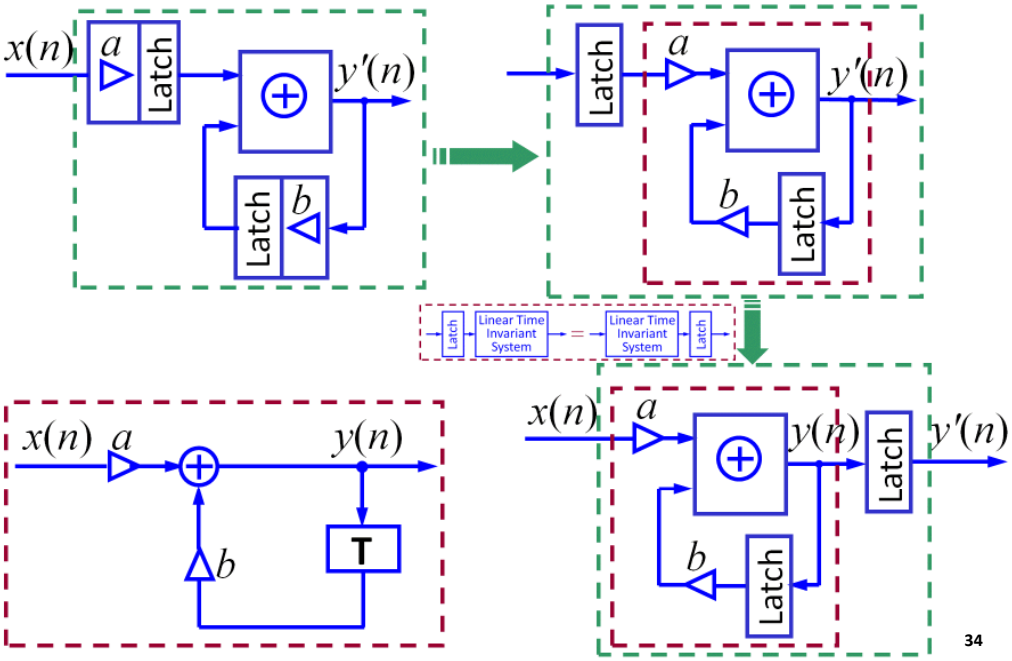




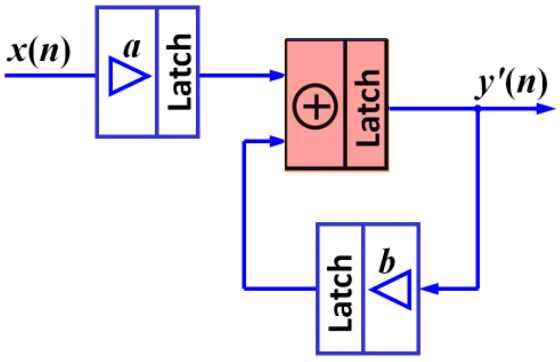
If the multiplier has a latch, the filter can be implemented as follows.



Note that $y'(n) = y(n-1)$.



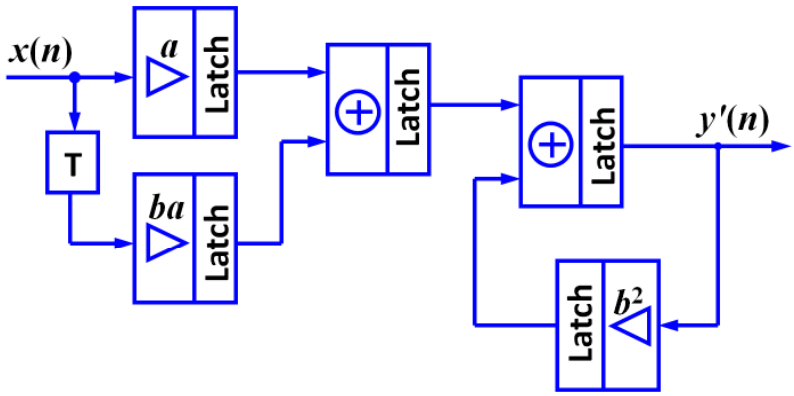
However, pipelining is not yet achieved since the critical path is still T_M+T_A .

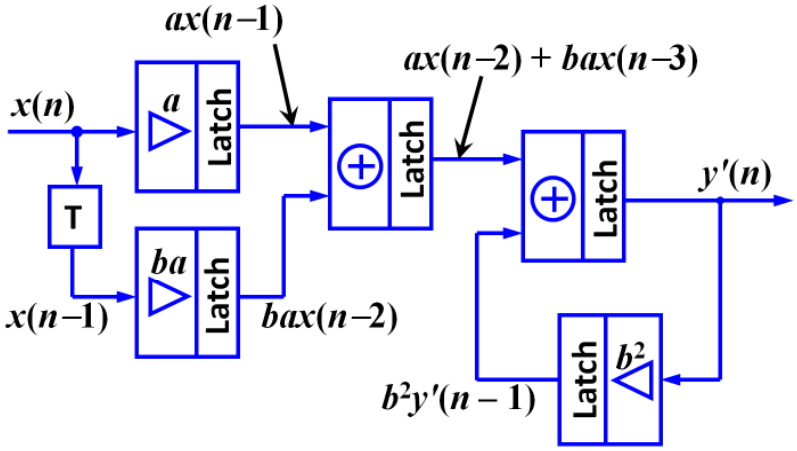


If a latch is inserted at the adder, the feedback loop will have two latches.
Feedback term is not $y'(n-1)$ but $y'(n-2)$.

Rewrite the filter output $y(n) = ax(n) + by(n-1)$ without using $y(n-1)$ but using $y(n-2)$ instead.

$$y(n) = ax(n) + by(n-1)$$
$$y(n-1) = ax(n-1) + by(n-2)$$
$$y(n) = ax(n) + bax(n-1) + b^2y(n-2)$$





The input/output relationship for the above implementation is $y'(n) = ax(n-3) + bax(n-4) + b^2y'(n-2)$.

We have, $y(n) = ax(n) + by(n-1) = ax(n) + bax(n-1) + b^2y(n-2)$. Replacing n by $n-3$, we have $y(n-3) = ax(n-3) + bax(n-4) + b^2y(n-5)$.

It can be seen that $y'(n) = y(n-3)$.

Latency = 3 (because of the latch), throughput = 1

If the adder has a latch and the multiplier is a two-latch pipelined multiplier, the feed back loop would be three-latch pipelined. Thus, $y(n-1)$ and $y(n-2)$ should be removed for the computation to be possible.

$n \rightarrow n-1$

$$y(n) = ax(n) + by(n-1)$$

$$y(n-1) = ax(n-1) + by(n-2)$$

$n \rightarrow n-2$

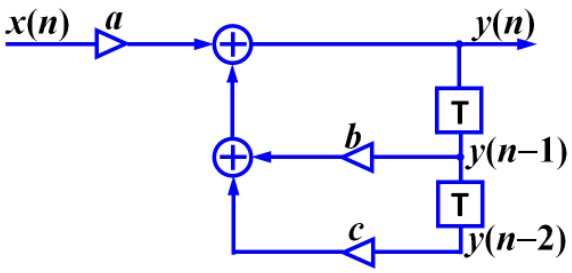
$$y(n) = ax(n) + bax(n-1) + b^2y(n-2)$$

$$y(n-2) = ax(n-2) + by(n-3)$$

$$y(n) = ax(n) + bax(n-1) + b^2ax(n-2) + b^3y(n-3)$$

Higher Order Filters

Example: $y(n) = ax(n) + by(n-1) + cy(n-2)$



$$y(n) = ax(n) + by(n-1) + cy(n-2)$$

$$y(n-1) = ax(n-1) + by(n-2) + cy(n-3)$$

$$y(n) = ax(n) + bax(n-1) + b^2y(n-2) + bcy(n-3) + cy(n-2)$$

$$= ax(n) + abx(n-1) + (b^2+c)y(n-2) + bcy(n-3)$$

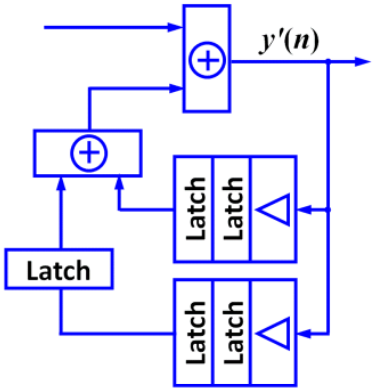
Example: Implement the filter

$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + b_1y(n-1) + b_2y(n-2)$

using 2-latch multipliers and non-pipelined adders.

Solution:

The structure of the feedback loop is as shown. There are two delays in the feedback path. Thus, it is a 2-latch pipelined system. Suppose the output of the 2-latch pipelined implementation is $y'(n)$. Since it is a 2-latch pipelined system, $y'(n-1)$ should be removed from the computation of $y'(n)$.



$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + b_1y(n-1) + b_2y(n-2)$

Substitute n by $n-1$, we have

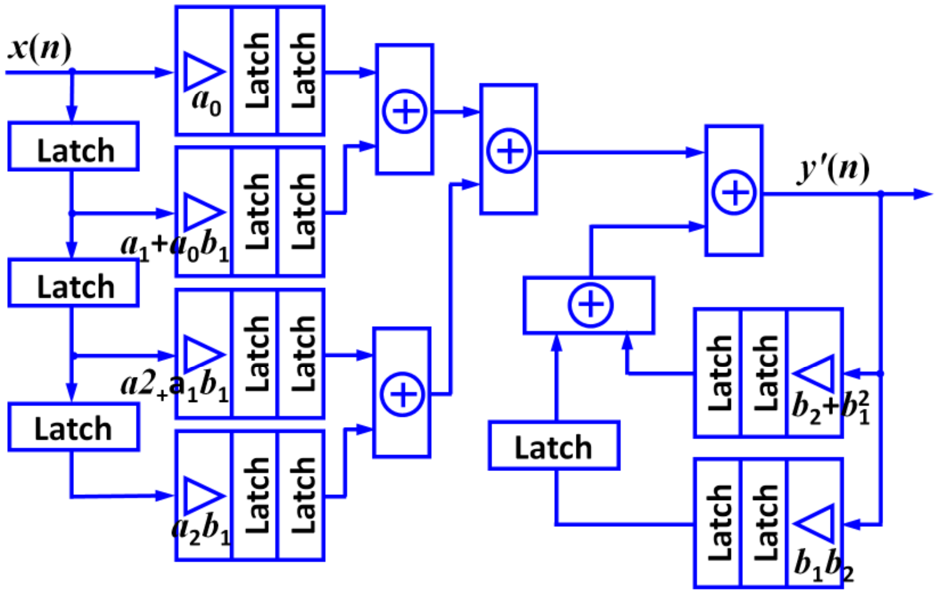
$y(n-1) = a_0x(n-1) + a_1x(n-2) + a_2x(n-3) + b_1y(n-2) + b_2y(n-3)$

Eliminating $y(n-1)$ in $y(n)$, we have

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + b_1\{a_0x(n-1) + a_1x(n-2) + a_2x(n-3) + b_1y(n-2) + b_2y(n-3)\} + b_2y(n-2)$$

$$= a_0x(n) + (a_1 + a_0b_1)x(n-1) + (a_2+a_1b_1)x(n-2) + a_2b_1x(n-3) + (b_2+b_1b_1)y(n-2) + b_1b_2y(n-3)$$

$y(n) = a_0x(n) + (a_1 + a_0b_1)x(n-1) + (a_2+a_1b_1)x(n-2) + a_2b_1x(n-3)$
$+ (b_2+b_1b_1)y(n-2) + b_1b_2y(n-3)$

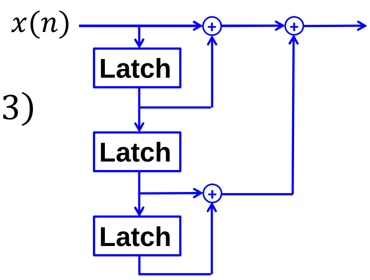


Example: Implement the filter $y(n) = x(n) + y(n-1)$ using 4-latch pipelined adders (no multiplier).

Solution: Feedback path has 4 latches, so *y*(*n*–1), *y*(*n*–2), and *y*(*n*–3) should be removed from the computation.

$$y(n) = x(n) + y(n-1)$$
$$= x(n) + x(n-1) + x(n-2) + x(n-3) + y(n-4)$$

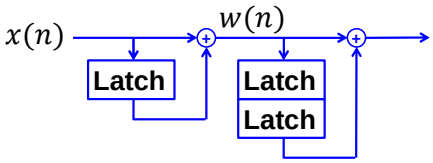
$x(n) + x(n-1) + x(n-2) + x(n-3)$
or
$(1 + z^{-1} + z^{-2} + z^{-3})X(z)$
needs 3 adders.



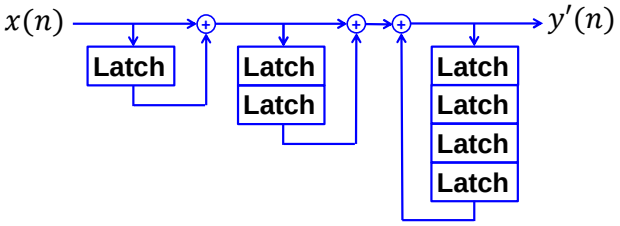
However, writing it as

$$\left(1+z^{-2}\right)\{(1+z^{-1})X(z)\} = \left(1+z^{-2}\right)W(z)$$

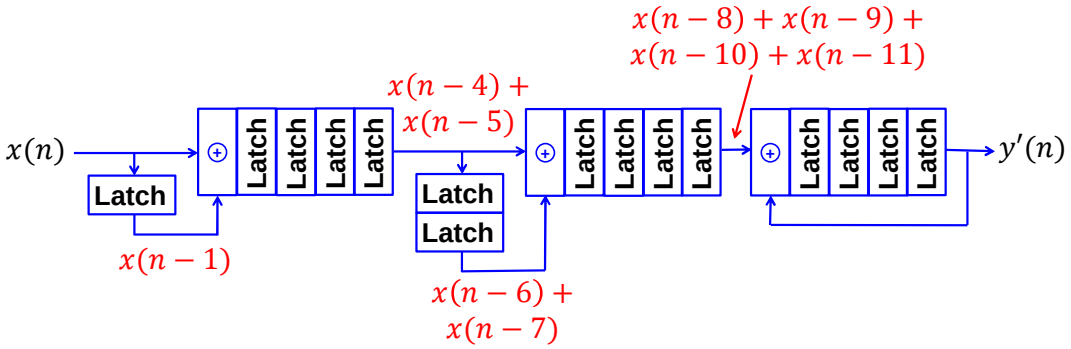
where $W(z) = (1+z^{-1})X(z)$, needs only 2 adders.



The complete implementation before pipelining is:



The pipelined implementation is:



The input/output relationship is:

$$y'(n) = x(n-12) + x(n-13) + x(n-14) + x(n-15) + y'(n-4)$$

Comparing with

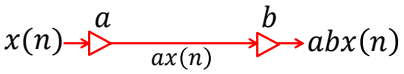
$y(n) = x(n) + x(n-1) + x(n-2) + x(n-3) + y(n-4)$

it can be seen that $y'(n) = y(n-12)$.

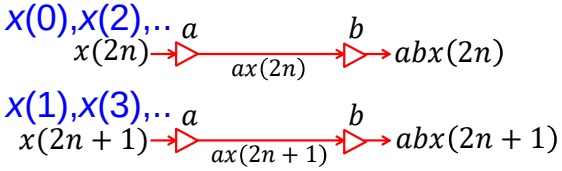
Parallel Processing

Introduction

- Increases the sampling rate/throughput
- Replicates the hardware
- Multiple outputs are computed in parallel in a clock period



sampling period T_s = clock cycle $T_c \geq 2T_M$
(T_M = multiplication time)



Duplicate the multipliers.
$T_c \geq 2T_M$ but $T_s = ½T_c \geq T_M$
Sampling frequency is doubled.

In parallel processing the critical path is unchanged, but the sampling becomes a fraction of the clock cycle.

Polyphase Parallel FIR Filters

A length *N* FIR filter requires *N* multiply and *N*–1 add of delay less than the sampling period *T*:

$$Y(z) = X(z)H(z), \text{where } X(z) = \sum x(n)z^{-n} \text{ etc.}$$

The input 2-phase polyphase components are

$$X_0(z) = \sum x(2n)z^{-n}, X_1(z) = \sum x(2n+1)z^{-n}$$

such that $X(z) = X_0(z^2) + z^{-1}X_1(z^2)$.

$X_0(z)$ is the z-transform of even-numbered inputs $x(2n)$.
$X_1(z)$ is the z-transform of odd-numbered inputs $x(2n+1)$.

Similarly, $Y(z) = Y_0(z^2) + z^{-1}Y_1(z^2)$
and $H(z) = H_0(z^2) + z^{-1}H_1(z^2)$.

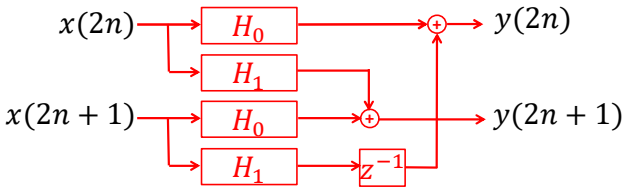
Then
$$\begin{aligned} Y(z) &= \{X_0(z^2) + z^{-1}X_1(z^2)\}\{H_0(z^2) + z^{-1}H_1(z^2)\} \\ &= \{X_0(z^2)H_0(z^2) + z^{-2}X_1(z^2)H_1(z^2)\} \\ &+ z^{-1}\{X_0(z^2)H_1(z^2) + X_1(z^2)H_0(z^2)\} \\ &= Y_0(z^2) + z^{-1}Y_1(z^2) \end{aligned}$$

It follows that: $Y_0(z) = X_0(z)H_0(z) + z^{-1}X_1(z)H_1(z)$
$Y_1(z) = X_0(z)H_1(z) + X_1(z)H_0(z)$

In matrix form (leaving z for brevity):

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-1}H_1 \\ H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$$

Implementation:



(Note that z^{-1} implies a delay of $2T$.)

The polyphase parallel FIR filter requires $2N$ multiply and $2(N–1)$ add with double the sampling frequency.
⇒ Strength reduction due to increased throughput.

Alternately, keeping the same sampling frequency, a slower multiplier/adder takes less area.
⇒ Strength reduction due to reduced area.

3-phase polyphase:

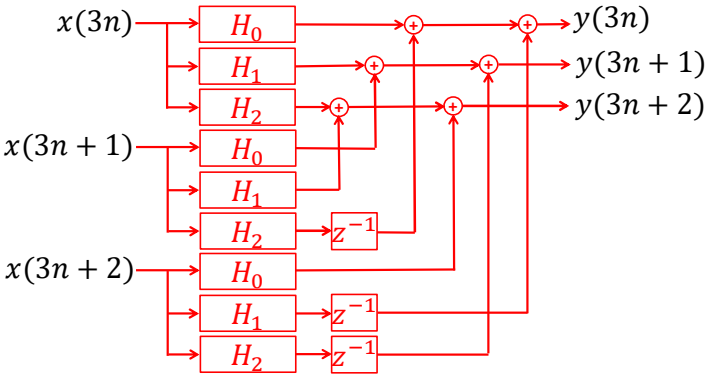
$$X(z) = X_0(z^3) + z^{-1}X_1(z^3) + z^{-2}X_2(z^3)$$

where $X_0(z)$ is the z-transform of inputs $x(3n)$, etc.

Then

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-1}H_2 & z^{-1}H_1 \\ H_1 & H_0 & z^{-1}H_2 \\ H_2 & H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix}$$

Implementation:



(z^{-1} implies a delay of 3T.)

The 3-phase polyphase parallel FIR filter requires 3N multiply and 3(N–1) add with triple the sampling frequency.

Generalization to L-phase polyphase:

Let: $X(z) = \sum_{l=0}^{L-1} z^{-l} X_l(z^L), Y(z) = \sum_{l=0}^{L-1} z^{-l} Y_l(z^L)$

$$H(z) = \sum_{l=0}^{L-1} z^{-l} H_l(z^L)$$

Then it may be shown that

$$Y_l(z) = \sum_{i=0}^{l} H_i(z) X_{l-i}(z) + z^{-1} \sum_{i=l+1}^{L-1} H_i(z) X_{L+l-i}(z)$$

for $0 \le l \le L-1$.

In matrix form:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{L-1} \end{bmatrix} = \begin{bmatrix} H_0 & z^{-1}H_{L-1} & \cdots & z^{-1}H_1 \\ H_1 & H_0 & \cdots & z^{-1}H_2 \\ \vdots & \vdots & \ddots & \vdots \\ H_{L-1} & H_{L-2} & \cdots & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{L-1} \end{bmatrix}$$

The *L*-phase polyphase parallel FIR filter requires *L* times the required number of multiply and add, with *L* times the sampling frequency.

Low-Complexity Parallel FIR Filters

Recall that in the 2-phase polyphase parallel FIR filters, the second output is given by:

$$Y_1(z) = X_0(z)H_1(z) + X_1(z)H_0(z)$$

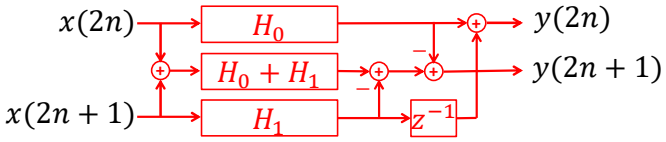
It may be simplified as:

$$Y_1(z) = \{X_0(z) + X_1(z)\}\{H_0(z) + H_1(z)\} - X_0(z)H_0(z) - X_1(z)H_1(z)$$

new computation

already computed for $Y_0(z)$

Implementation:



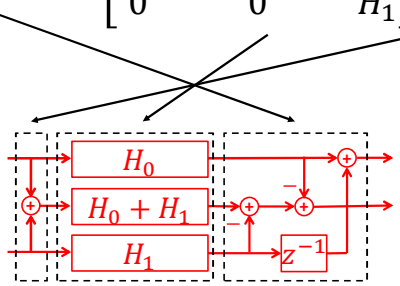
This low-complexity implementation needs only 3 subfilters, each of length *N*/2.
$\Rightarrow$ It requires 1.5*N* multiply and 1.5*N*+1 add.

Other equivalent low-complexity parallel FIR filter structures may be obtained. For example, a structure with 3 subfilters H_0, H_0–H_1, and H_1 may be realized.

Systematic procedure to find a low-complexity structure:

Given $\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-1}H_1 \\ H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}$ or **Y**=**H**.**X**, diagonalize **H**.

$$\mathbf{H} = \begin{bmatrix} H_0 & z^{-1}H_1 \\ H_1 & H_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & z^{-1} \\ -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} H_0 & 0 & 0 \\ 0 & H_0+H_1 & 0 \\ 0 & 0 & H_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$



3-phase polyphase:

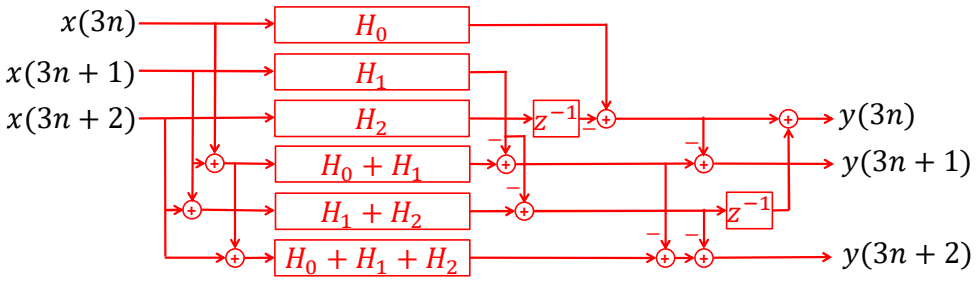
$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} H_0 & z^{-1}H_2 & z^{-1}H_1 \\ H_1 & H_0 & z^{-1}H_2 \\ H_2 & H_1 & H_0 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix}$$

Diagonalize **H**:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & z^{-1} & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -z^{-1} & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times$$

$$\begin{bmatrix} H_0 & & & & & \\ & H_1 & & & & \\ & & H_2 & & & \\ & & & H_0 + H_1 & & \\ & & & & H_1 + H_2 & \\ & & & & & H_0 + H_1 + H_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Implementation:

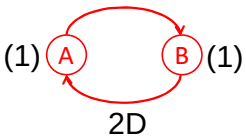


This structure needs 6 subfilters, each of length *N*/3.
⇒ It requires 2*N* multiply and 2*N*+4 add.

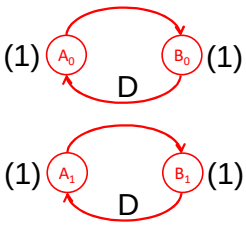
Unfolding

Unfolding

Unfolding replicates a system *J* times, where *J* is the unfolding factor.
Unfolding reduces the sampling period, reaching the iteration bound. It also achieves parallel processing.



Iteration bound = (1+1)/2 = 1
Operations: $A_0 \rightarrow B_0$, $A_1 \rightarrow B_1$, $A_2 \rightarrow B_2$,…

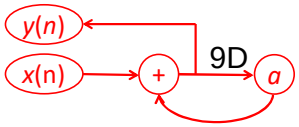


J unfolding: *J* copies, each delay equals *J* earlier delays.

Iteration bound = (1+1)/1 = 2
Top: $A_0 \rightarrow B_0$, $A_2 \rightarrow B_2$,…
Bottom: $A_1 \rightarrow B_1$, $A_3 \rightarrow B_3$,…

Example 1:

$$y(n) = ay(n-9) + x(n)$$

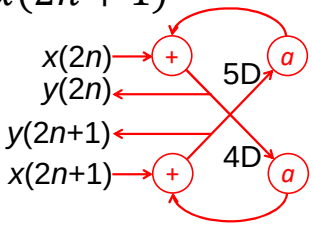


J=2 unfolding:

$$\begin{cases} y(2n) = ay(2n-9) + x(2n) \\ y(2n+1) = ay(2n-8) + x(2n+1) \end{cases}$$

But $y(2n-9)$ is $y(2n+1)$ delayed by 5, since each delay after unfolding equals 2 delays.
Similarly, $y(2n-8)$ is $y(2n)$ delayed by 4.

$$\begin{cases} y(2n) = ay[2(n-5)+1] + x(2n) \\ y(2n+1) = ay[2(n-4)] + x(2n+1) \end{cases}$$

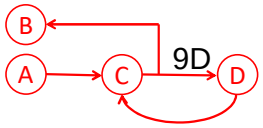


Algorithm for Unfolding

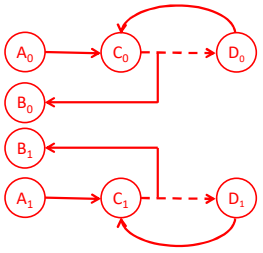
Systematic procedure for unfolding:

Step 1: For each node *U* in the original data flow graph, draw *J* nodes $U_0, U_1, \ldots, U_{J-1}$.

Original data flow graph:



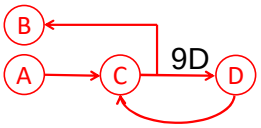
Replicated data flow graph:



Step 2: For each edge $U \rightarrow V$ with m delays in the original data flow graph, draw J edges $U_i \rightarrow V_{(i+m)\%J}$ with $\left\lfloor \frac{i+m}{J} \right\rfloor$ delays for $i = 0, 1, \ldots, J-1$.

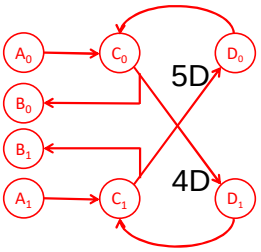
(% = mod or modulo, find the remainder.)

($\lfloor \bullet \rfloor$ = floor, same as truncation.)

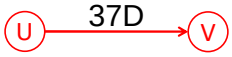


Edges without delay such as $A \rightarrow C$, $D \rightarrow C$ are unchanged.

Edge $C \rightarrow D$ has $m = 9$ delays.
So, $C_0 \rightarrow D_{(0+9)\%2} = D_1$ has $\lfloor (0+9)/2 \rfloor = 4$ delays.
$C_1 \rightarrow D_{(1+9)\%2} = D_0$ has $\lfloor (1+9)/2 \rfloor = 5$ delays.

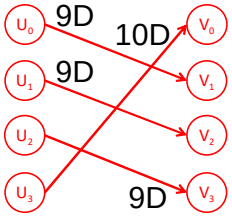


Example 2:

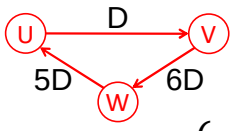


Let $J = 4$.
Recall that $U_i \rightarrow V_{(i+m)\%J}$ with $\lfloor (i+m)/J \rfloor$ delays.

$$\left\lfloor \frac{i+m}{J} \right\rfloor = \left\lfloor \frac{i+37}{4} \right\rfloor = \begin{cases} 9 & i = 0,1,2 \\ 10 & i = 3 \end{cases}$$



Example 3:

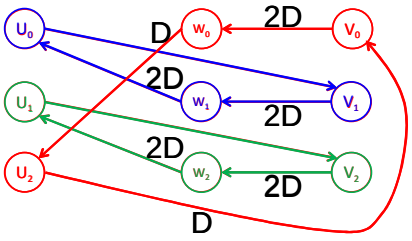


Let $J = 3$.

For $U_i \rightarrow V_{(i+1)\%3}$, $\left\lfloor \frac{i+m}{J} \right\rfloor = \left\lfloor \frac{i+1}{3} \right\rfloor = \begin{cases} 0 & i = 0,1 \\ 1 & i = 2 \end{cases}$

For $V_i \rightarrow W_{(i+6)\%3}$, $\left\lfloor \frac{i+m}{J} \right\rfloor = \left\lfloor \frac{i+6}{3} \right\rfloor = 2$

For $W_i \rightarrow U_{(i+5)\%3}$, $\left\lfloor \frac{i+m}{J} \right\rfloor = \left\lfloor \frac{i+5}{3} \right\rfloor = \begin{cases} 1 & i = 0 \\ 2 & i = 1,2 \end{cases}$



Consists of 3 disjoint loops since gcd(12 delays, J=3) = 3.

Properties of Unfolding

Unfolding preserves the number of delays in a graph, since:

$$\left\lfloor \frac{0+m}{J} \right\rfloor + \left\lfloor \frac{1+m}{J} \right\rfloor + \cdots + \left\lfloor \frac{J-1+m}{J} \right\rfloor = m$$

Unfolding preserves precedence constraints of a system.

Unfolding of a loop with m delays leads to gcd(m,J) loops.
Each of these loops contains m/gcd(m,J) delays.
Each of these loops contains J/gcd(m,J) copies of each node that appears in the original loop.

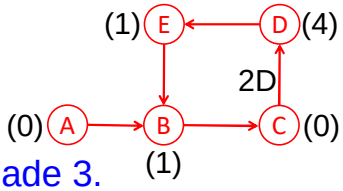
Unfolding a graph with iteration bound T_∞ results in the new iteration bound JT_∞.

Applications of Unfolding

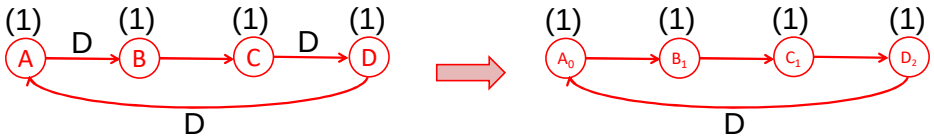
- Sampling period reduction
  - Case 1: Longest node computation time is greater than the iteration bound
  - Case 2: Iteration bound is not an integer
  - Case 3: Both case 1 and case 2
- Parallel processing

Sampling Period Reduction

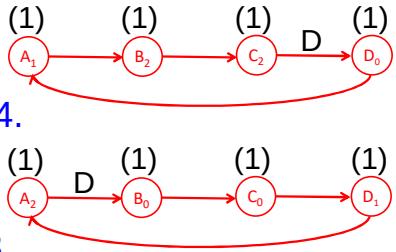
Example of case 1:
Longest node computation time (for D) 4 > iteration bound $T_\infty = 3$.
Sampling period cannot be made 3.
Use J unfolding to change the iteration bound to $JT_\infty > 4$.



Example of case 2:
Iteration bound $T_\infty = 4/3$, not an integer.



Use J unfolding to change the iteration bound to JT_∞ = integer.
With $J = 3$, iteration bound $JT_\infty = 4$.
New sampling period = 1 (=4), but 3 inputs/outputs, so effective sampling period = 4/3.



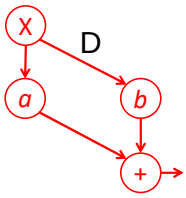
For case 3, use J unfolding to change the iteration bound to JT_∞ = integer > longest node computation time.
If $T_\infty = 4/3$ and longest node time is 6, use J=6 so that $JT_\infty = 8$, an integer greater than 6.

Parallel FIR Filters

Unfolding converts a sequential system to a parallel system.

Consider the following sequential system (FIR filter):

$$y(n) = ax(n) + bx(n-1)$$



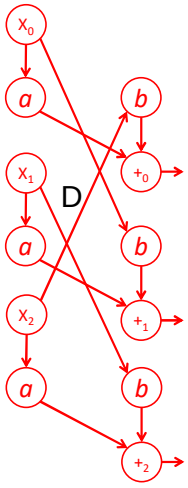
Critical path = $T_M + T_A$

So, clock cycle $T_c \geq T_M + T_A$

Since sampling period $T_s = T_c$, $T_s \geq T_M + T_A$

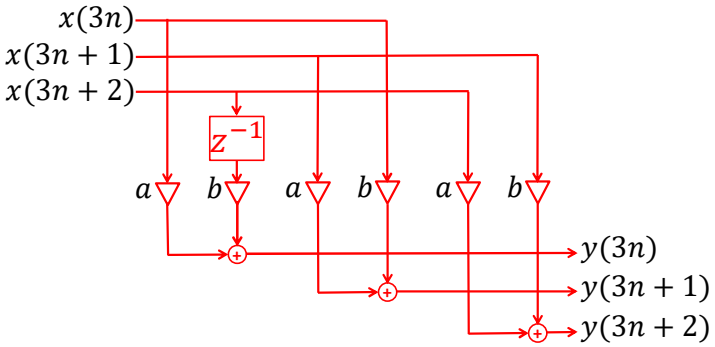
Convert it to a parallel system with unfolding factor (also known as block size) J=3

At the n–th clock cycle, 3 inputs $x(3n)$, $x(3n+1)$, and $x(3n+2)$ are processed, and 3 outputs $y(3n)$, $y(3n+1)$, and $y(3n+2)$ are generated.



Critical path = $T_M + T_A$
Clock cycle $T_c \geq T_M + T_A$
But $3T_s = T_c$, or $T_s \geq (T_M+T_A)/3$

Parallel architecture:

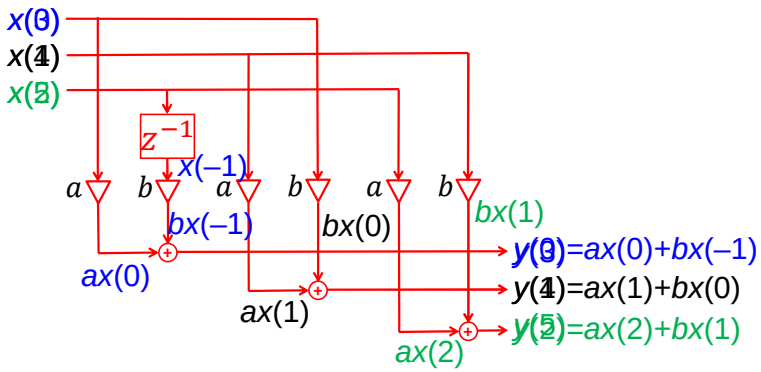


In unfolding factor J parallel structures, since $T_c = JT_s$, a latch z^{-1} of 1 clock cycle produces an effective delay of J sampling periods, JT_s .

In the above structure, $x(3n+2)$ passing through z^{-1} is delayed by 3 sampling periods, or becomes $x(3n–1)$.

Since $T_s = 1/3\ T_c$, the sampling frequency is tripled.
Latency = 1 clock, throughput = 3 inputs/clock

Functioning of the parallel architecture:
Let n=0.



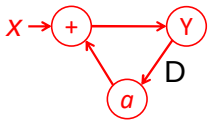
Parallel Recursive Filters

First Order Filters

Even though unfolding works for the IIR filter in unfolding example 1, it doesn't work for most IIR filters

Consider a first order IIR filter:

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

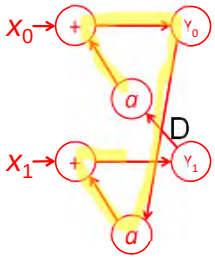


Critical path = T_M+T_A, so $T_s \geq T_M$+T_A

Hardware complexity: sequential architecture
1 multiply, 1 add

Option 1: Convert it to a parallel system with unfolding factor 2

Critical path = $2T_M + 2T_A$
so $T_s \geq (2T_M+2T_A)/2$
no improvement!

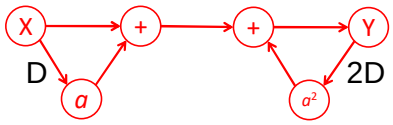


Hardware complexity: parallel architecture
- 1 multiply, 1 add
- 2 sets

total 2 multiply, 2 add
latency = 1 clock, throughput = 2 inputs/clock

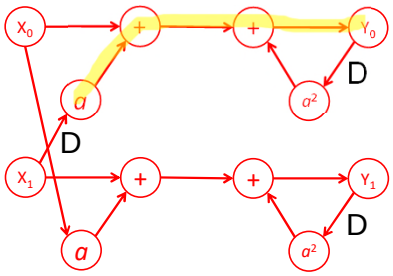
Option 2: To obtain disjoint loops, change D to 2D (J=2). Rewrite the filter output without using $y(2n–1)$ but using $y(2n–2)$ instead.

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

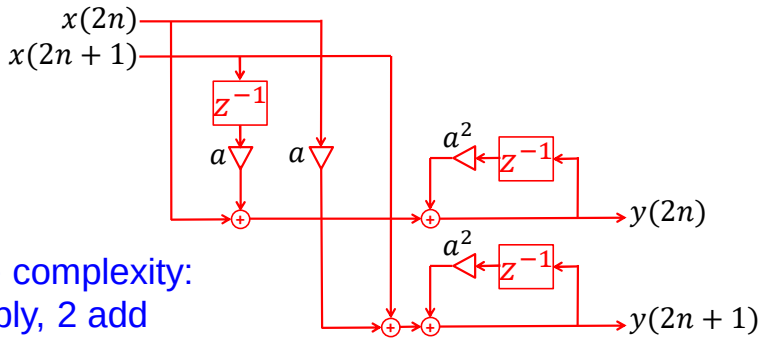


Convert to parallel

Critical path = $T_M + 2T_A$
so $T_s \geq (T_M+2T_A)/2$
improvement



Parallel architecture:



Hardware complexity:
- 2 multiply, 2 add
- 2 sets

total 2^2 multiply, 2^2 add

For general unfolding factor J,
- J multiply, J add
- J sets

total J^2 multiply, J^2 add

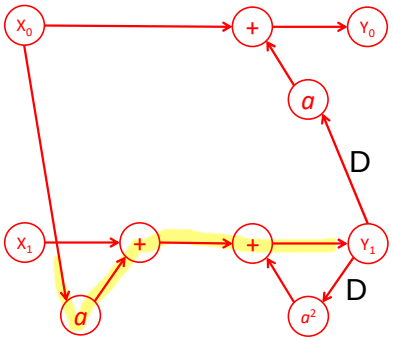
Option 3: Incremental block processing:
The J^2 increase in the hardware complexity can be reduced at the expense of $1/J$ improvement in T_s, or latency.

No improvement in option 1, since y_1 is computed using y_0
To break the critical path, compute y_1 without using y_0

It is called incremental block processing, because
- Some outputs are computed using block processing (option 1), such as y_0
- Other outputs are computed without using y_i, such as y_1 computed without using y_0 (uses delayed y_1 instead)

y_0 computed as block processing (option 1)

$$y_0(n) = x(n) + ay(n-1)$$



y_1 computed from past y_1 [which is $y(n-2)$]

$$y_1(n) = x(n) + ax(n-1) + a^2y(n-2)$$

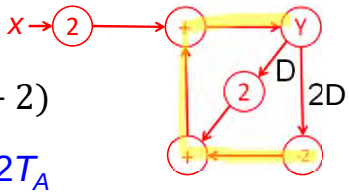
Critical path = $T_M + 2T_A$

Hardware complexity:
- set 1: 1 multiply, 1 add
- set 2: 2 multiply, 2 add

total 3 multiply, 3 add

Higher Order Filters

Consider a second order IIR filter:



$$y(n) = 2x(n) + 2y(n-1) - 2y(n-2)$$

Critical path = T_M+2T_A, so $T_s \geq T_M+2T_A$

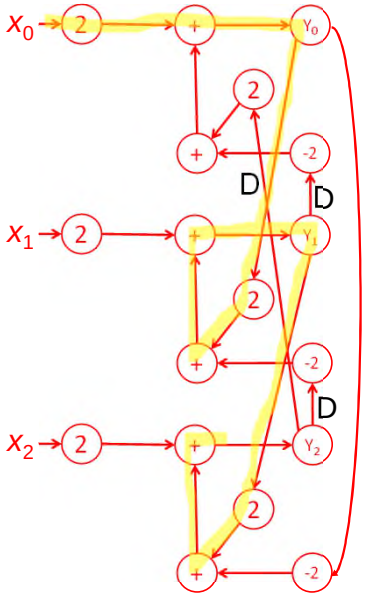
Hardware complexity: sequential architecture
- inputs 1 multiply, 0 add
- past outputs 2 multiply, 2 add

total 3 multiply, 2 add

In general, for any IIR filter with #inputs/numerator order N, and #past outputs/ denominator order M,
- inputs N multiply, N–1 add
- past outputs M multiply, M add

total $N+M$ multiply, $N+M$–1 add

Option 1: Convert it to a parallel system with J=3



Critical path = $3T_M + 5T_A$
so $T_s \geq (3T_M+5T_A)/3$
marginal improvement

Hardware complexity:
- 3 multiply, 2 add
- 3 sets

total 9 multiply, 6 add

Option 2: To obtain disjoint loops, change D to 3D (J=3).
Rewrite the filter output using $y(n–3)$ and $y(n–6)$.

$$y(n) = 2x(n) + 2y(n-1) - 2y(n-2)$$
$$= 2x(n) + 4x(n-1) + 2y(n-2) - 4y(n-3)$$
$$\vdots$$
$$= 2x(n) + 4x(n-1) + 4x(n-2) + 0x(n-3)$$
$$-8x(n-4) - 8y(n-5) + 8y(n-6) \quad (1)$$

But $y(n–3)$ is equal to:

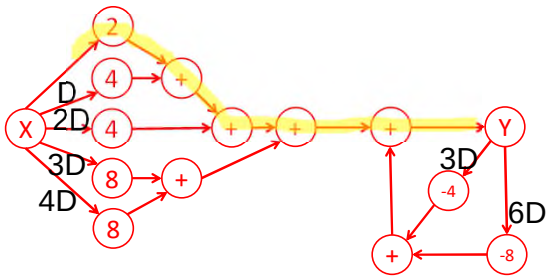
$$y(n-3) = 2x(n-3) + 4x(n-4) + 2y(n-5) - 4y(n-6) \quad (2)$$

To cancel $y(n–5)$, scale equation (2) by 4 and add to (1):

$$y(n) + 4y(n-3) = 2x(n) + 4x(n-1) + 4x(n-2)$$
$$+8x(n-3) + 8x(n-4) - 8y(n-6)$$

or

$$y(n) = 2x(n) + 4x(n-1) + 4x(n-2) + 8x(n-3) + 8x(n-4) - 4y(n-3) - 8y(n-6)$$

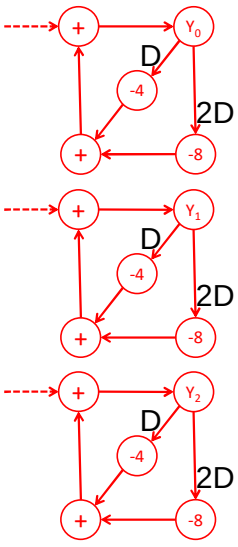


Critical path = T_M+4T_A

Convert to parallel system.
Loops will be disjoint.

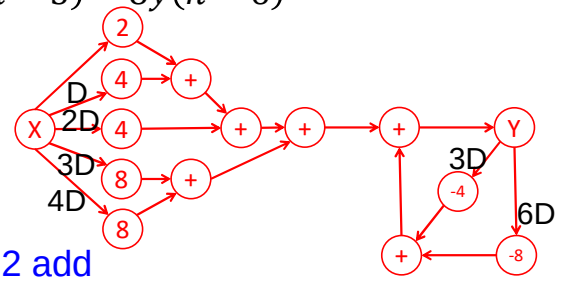
The input (left) part has the same critical path,
so critical path = T_M + 4T_A
so $T_s \geq (T_M+4T_A)/3$

improvement



Hardware complexity:

$$y(n) = 2x(n) + 4x(n-1) + 4x(n-2) + 8x(n-3) + 8x(n-4) - 4y(n-3) - 8y(n-6)$$



- inputs 5 multiply, 4 add
- past outputs 2 multiply, 2 add
- 3 sets

total 21 multiply, 18 add

In general,

- inputs *JM–M+N* multiply, *JM–M+N*–1 add
- past outputs *M* multiply, *M* add
- *J* sets

total J^2M+JN multiply, $J^2M+J(N-1)$ add

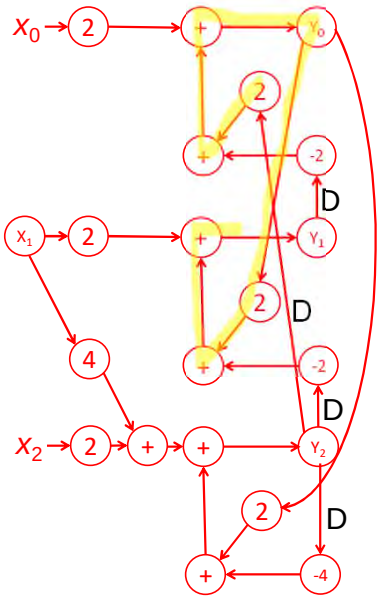
Option 3: Incremental block processing

To break the critical path, compute y_2 without using y_1

- y_0, y_1 computed using block processing (option 1)
- y_2 computed without using y_1 (uses delayed y_2 instead)

$$y_2(n) = 2x(n) + 4x(n-1) + 2y(n-2) - 4y(n-3)$$

Critical path = $2T_M+4T_A$
so $T_s \geq (2T_M+4T_A)/3$



Hardware complexity:

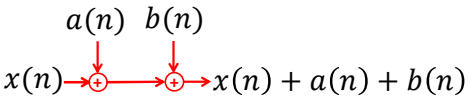
- inputs 4 multiply, 1 add
- past outputs 6 multiply, 6 add

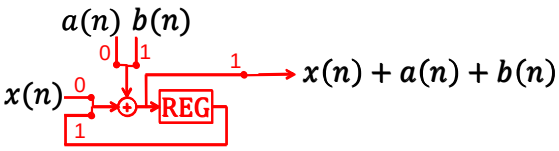
total 10 multiply, 7 add

Folding

Folding

Folding reduces the silicon area (hardware complexity) by time multiplexing multiple operations into single functional units such as adders and multipliers.



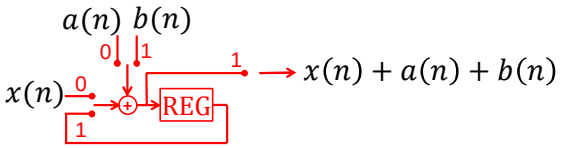


2 times folded architecture: 2 adders are folded to 1 adder

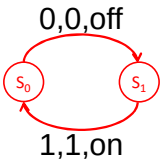
Clock cycle 0: Add *x*(*n*) with *a*(*n*), store in register.

Clock cycle 1: Add register with *b*(*n*), this is the output.

Folding introduces registers/storage and control signals.



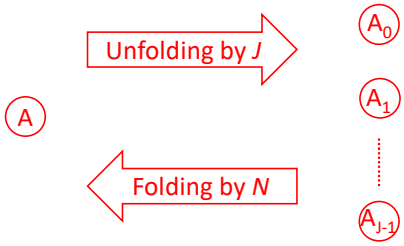
Control signals are generated by a finite state machine.
Outputs = control for 2 multiplexers and 1 switch



Input data to a *N* folded architecture is assumed to be valid for *N* cycles, and one output sample is produced every *N* cycles, where *N* is the folding factor.

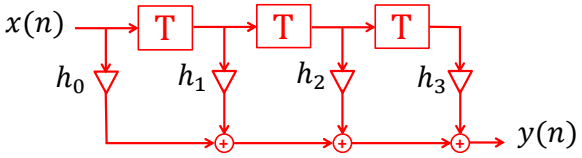
Hardware is reduced by a factor of *N*, but computation time is increased to *N* clock cycles. Latency is increased accordingly.
⇒ Folding trades hardware complexity with computation time.

Folding = reverse transformation of unfolding.

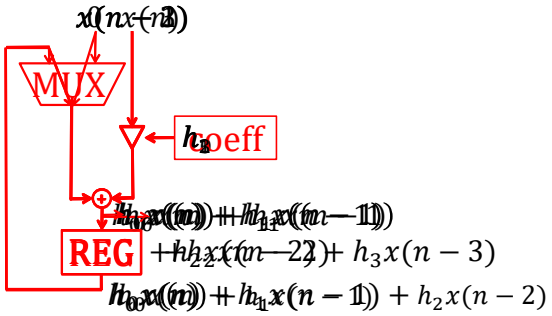


Folding an FIR Filter

FIR filter:
1 sampling period =
1 computation cycle.
N fixed multipliers,
N – 1 adders.



Folding:



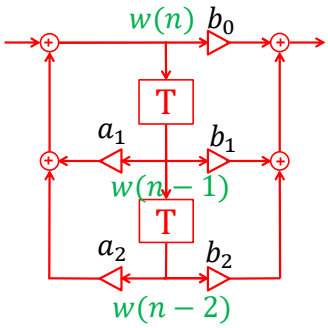
Cycle 0:

1 sampling period = *N* computation cycles
1 generalized multiplier, 1 adder, 1 coefficient memory, control

This strategy is often used in DSP chips.

Folding an IIR Filter

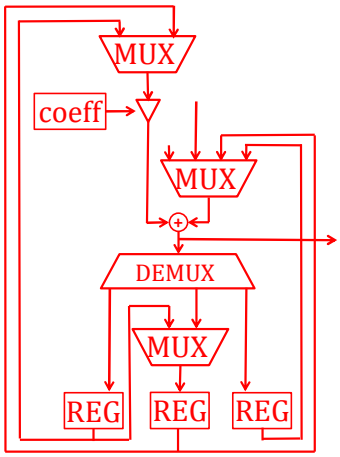
IIR filter: $y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2)$
$+a_1y(n-1) + a_2y(n-2)$



Direct form II realization:

$$w(n) = x(n) + a_1w(n-1) + a_2w(n-2)$$
$$y(n) = b_0w(n) + b_1w(n-1) + b_2w(n-2)$$

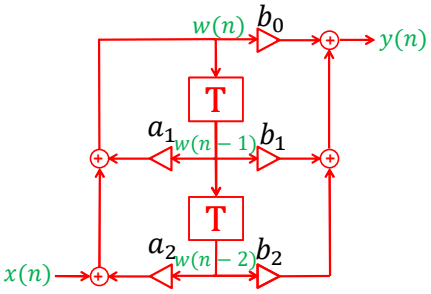
Before folding: 5 multipliers with fixed coefficients, 4 adders, 2 delays, latency 1 clock cycle.



After folding: 1 general multiplier, 1 adder, 3 delays/registers, coefficient memory, 4 multiplexer-demultiplexers, controller, latency 5 clock cycles.

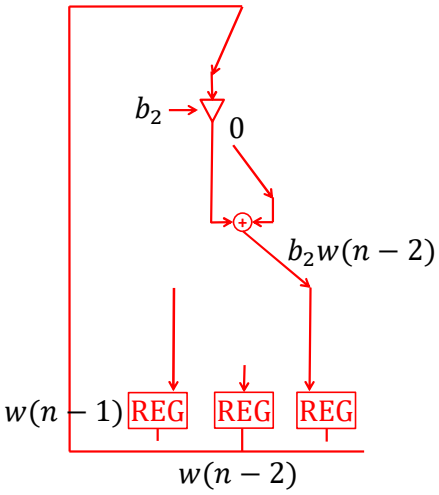
Cycle	Register 1	Register 2	Register 3	compute
0	$w(n-1)$	$w(n-2)$		$b_2 w(n-2)$

- output to register
- store in register



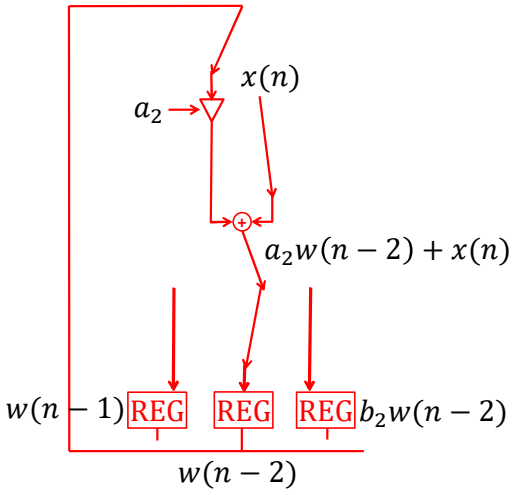
Functioning of the folded architecture:

Cycle 0:



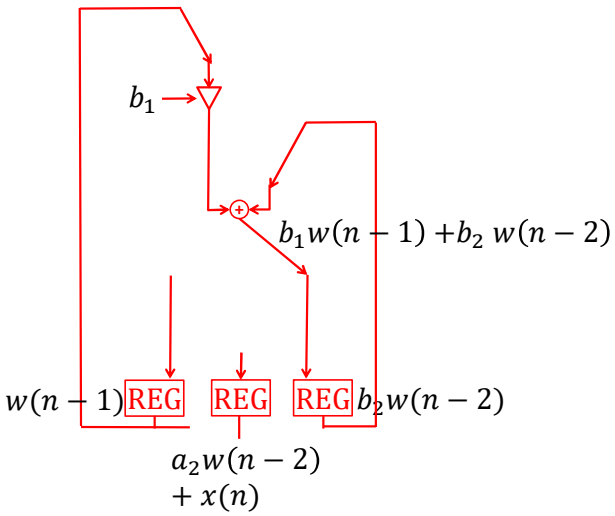
Functioning of the folded architecture:

Cycle 1:



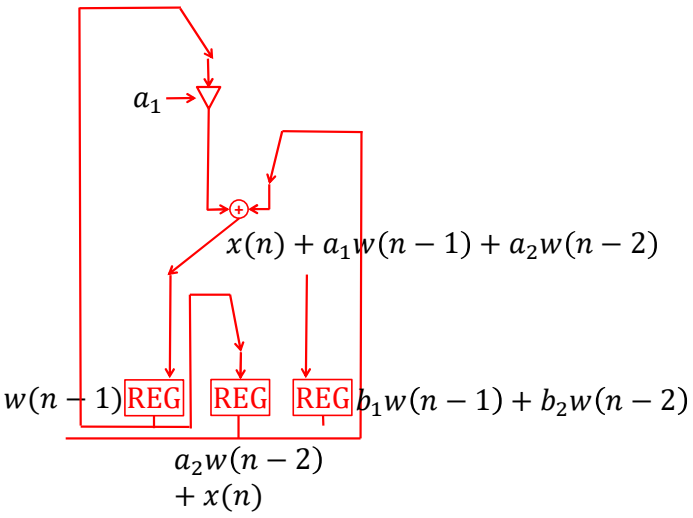
Functioning of the folded architecture:

Cycle 2:



Functioning of the folded architecture:

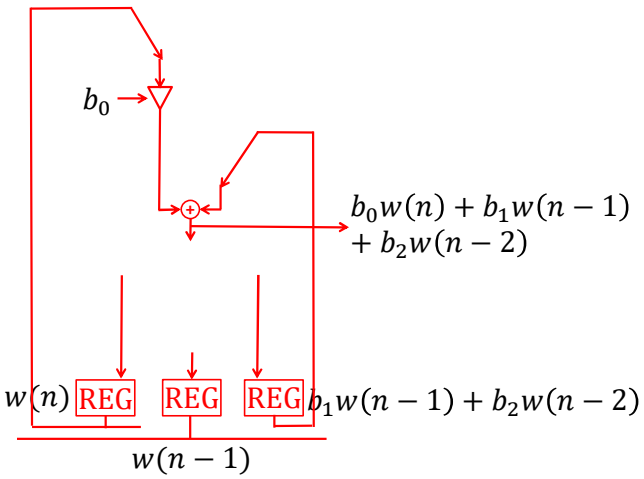
Cycle 3:



$$w(n) = x(n) + a_1 w(n-1) + a_2 w(n-2)$$

Functioning of the folded architecture:

Cycle 4:



$$y(n) = b_0w(n) + b_1w(n-1) + b_2w(n-2)$$