COMPUTATIONAL PROPERIES OF DSP ALGORITHMS

DSP Algorithms

A DSP algorithm is a computational rule, *f*, that maps an ordered input sequence, $x(nT)$, to an ordered output sequence, $y(nT)$, according to

$$x(nT)\ \text{Æ}\ y(nT) \quad , \quad y(nT) := f(x(nT))$$

Generally, in hard real-time applications the mapping is required to be causal since the input sequence is usually causal.

We stress that the computational rule is an unambiguously specified sequence of operations on an ordered data set.

DSP ALGORITHM		
Sequence of operations	Basic set of arithmetic and logic operations	Ordered data set

A DSP algorithm can be described by a set of expressions. The sign ":=" is used to indicate that the expressions are given in computational order.

$$\begin{cases} x_1(n) := f_1[-, x_1(n-1)\ldots, x_p(n-1), x_p(n-2), \ldots, a_1, b_1, \ldots] \\ x_2(n) := f_2[-, x_1(n-1)\ldots, x_p(n-1), x_p(n-2), \ldots, a_2, b_2, \ldots] \\ x_3(n) := f_3[x_1(n), x_1(n-1)\ldots, x_p(n-1), x_p(n-2), \ldots, a_3, b_3, \ldots] \\ x_N(n) := f_N[x_1(n), x_1(n-1)\ldots, x_p(n-1), x_p(n-2), \ldots, a_N, b_N, \ldots] \end{cases}$$

DSP algorithms can be divided into *iterative processing* and *block processing* algorithms.

An iterative algorithm performs computations on a semi-infinite stream of input data, i.e., input data arrive sequentially and the algorithm is executed once for every input sample and produces a corresponding output sample.

In block processing, a block of output samples is computed for each input block, which results in a large delay between input and output.

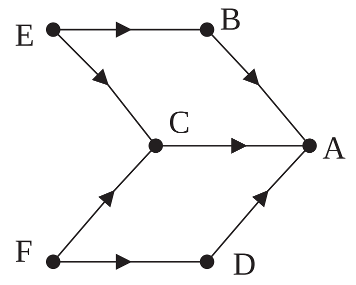
PRECEDENCE GRAPHS

A *precedence graph* describes the order of occurrence of events: *A*, *B*, …, *F*.

The directed branches between the nodes denote the ordering between the events which are *represented* by the nodes.

A directed branch between node *E* and *B* shows that event *E* precedes event *B*.

E is therefore called a *precedent* or predecessor to B.



Node E also precedes event *A*, and therefore node *E* is a second-order precedent to node *A*.

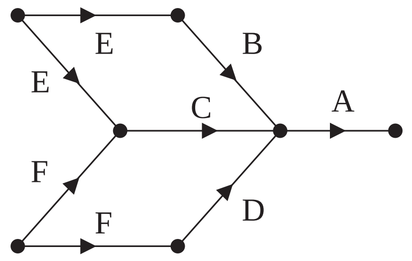
Event *B* is a succedent to event *E*.

An *initial node* has no precedents and a *terminal node* has no succedents, while an *internal node* has both.

If two events are not connected via a branch, then their precedence order is unspecified.

Sometimes, it may be more convenient to let the branches represent the events and the nodes represent the precedence relations.

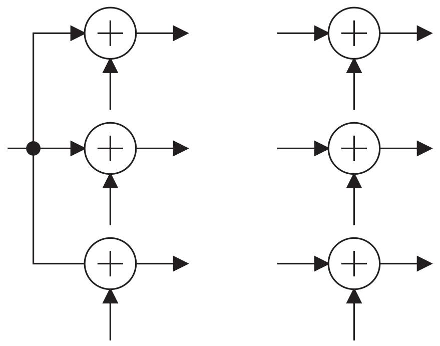
Such precedence graphs are called *AOA* (*activity on arcs*) graphs while the former type of graphs, with activity on the nodes, are called *AON* (*activity on nodes*) graphs.



Parallelism in Algorithms

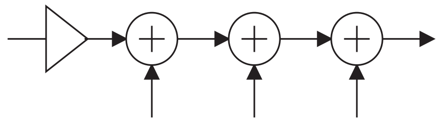
Precedence relations between operations are unspecified in a *parallel algorithm*.

The figure illustrates two examples of parallel algorithms. In the first case, the additions have a common precedent, while in the second case they are independent.



Two operations (algorithms) are *concurrent* if their execution times overlap.

In a *sequential algorithm* every operation, except for the first and the last, has only one precedent and one succedent operation.

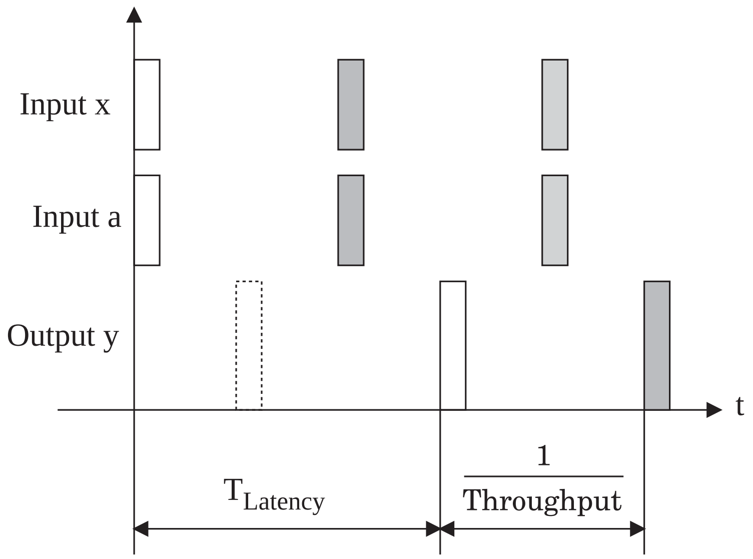


Thus, the precedence relations are uniquely specified for a sequential algorithm.

Latency

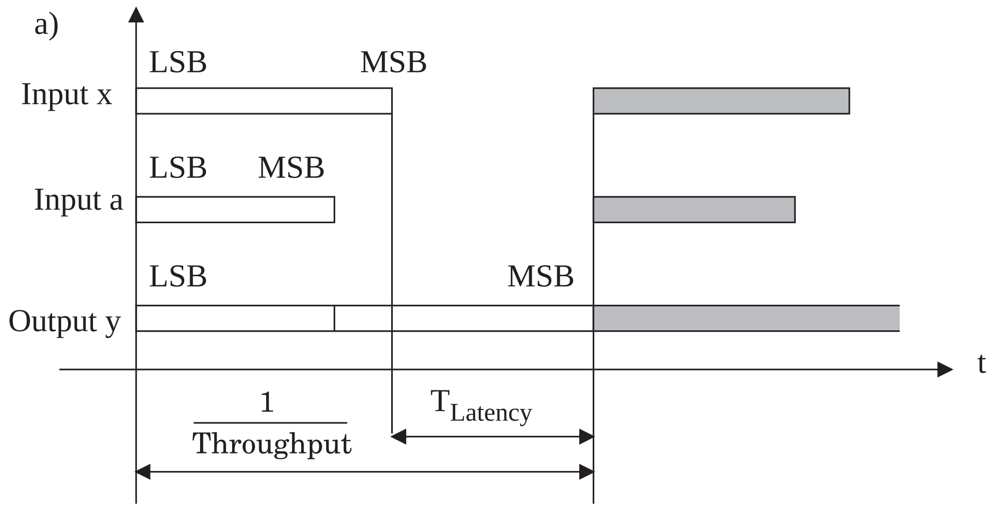
We define *latency* as the time it takes to generate an output value from the corresponding input value.

Example: Bit-parallel multiplication



Note that latency refers to the time between input and output, while the *algorithmic delay* refers to the difference between input and output sample indices.

Example: Bit-serial multiplication



Bit-serial multiplication can be done either by processing the *least significant* or the *most significant bit first*.

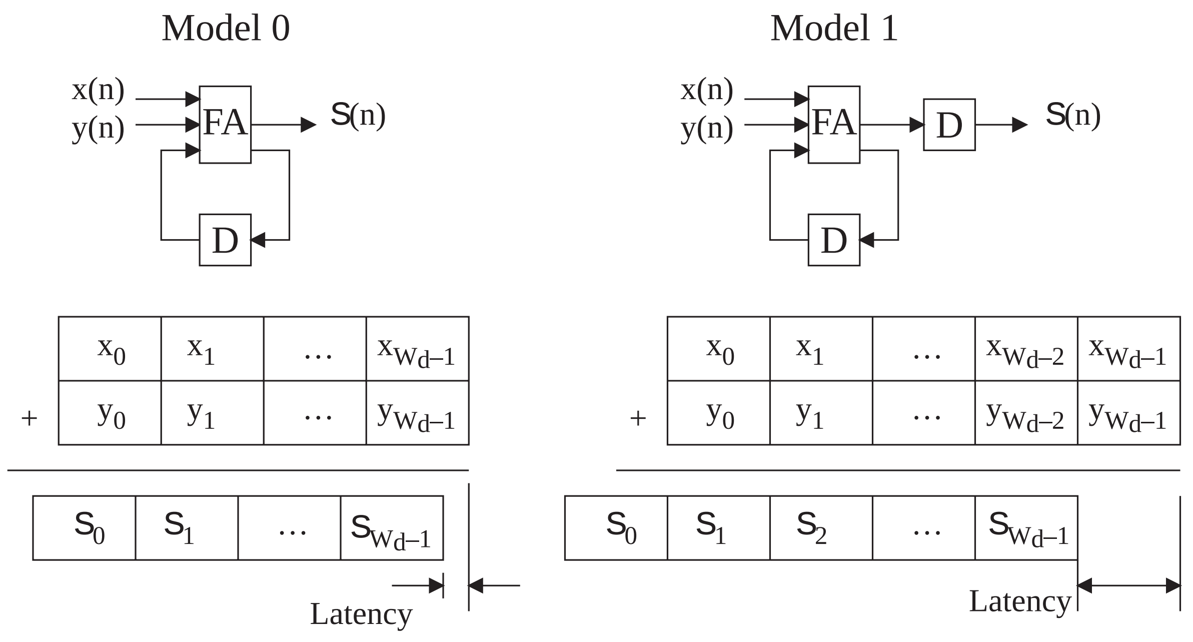
The latency, if the LSB is processed first, is in principle equal to the number of fractional bits in the coefficient.

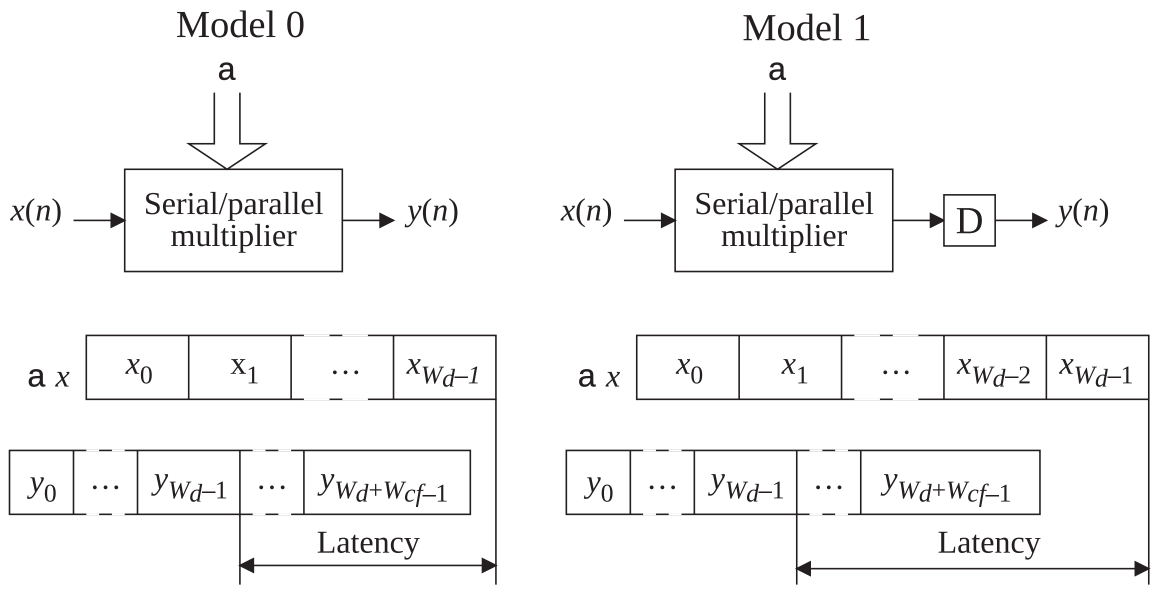
For example, a multiplication with a coefficient $W_c = (1.0011)_{2C}$ will have a latency corresponding to four clock cycles.

A bit-serial addition or subtraction has in principle zero latency while a multiplication by an integer may have zero or negative latency.

However, the latency in a recursive loop is always positive, since the operations must be performed by causal PEs.

Latency models for bit-serial arithmetic.



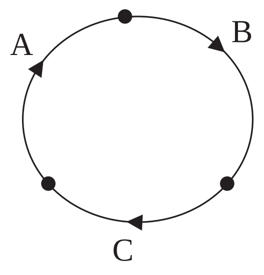


Denoting the number of fractional bits of the coefficient W_{af}, the latencies become W_{af} for latency model 0, and $W_{af} + 1$ for latency model 1.

Sequentially Computable Algorithms

A precedence graph may be contradictory in the sense that it describes an impossible ordering of events.

An example of such a precedence graph is shown in the Figure.



Event *A* can occur only after event *C* has occurred.

However, event *C* can only occur when event *B* has occurred, but event *B* cannot occur until event *A* has occurred.

Hence, this sequence of events is impossible since the sequence can not begin.

In a digital signal processing algorithm, events correspond to arithmetic operations.

In a proper algorithm at least one of the operations in each recursive loop in the signal-flow graph must have all its input values available so that it can be executed.

This is satisfied only if the loops contain at least one delay element, since the delay element contains a value from the preceding sample interval that can be used as a starting point for the computations in the current sample interval.

Thus, there must not be any delay-free loops in the signal-flow graph.

For a sequentially computable algorithm the corresponding precedence graph is called a *directed acyclic graph (DAG)*.

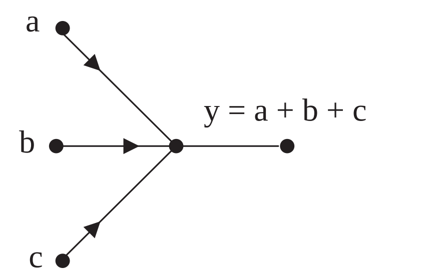
A necessary and sufficient condition for a recursive algorithm to be sequentially computable is that every directed loop in the signal-flow graph contains at least one delay element.

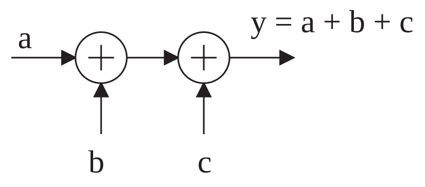
An algorithm that has a delay-free loop is *nonsequentially computable*.

Such an algorithm can be implemented neither as a computer program nor in digital hardware.

Fully Specified Signal-Flow Graphs

In a *fully specified signal-flow graph*, the ordering of all operations as uniquely specified is illustrated in the figure.



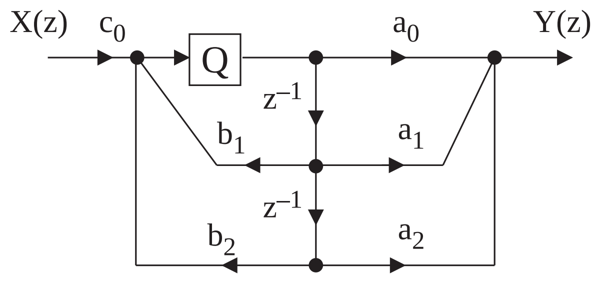


SFGs IN PRECEDENCE FORM

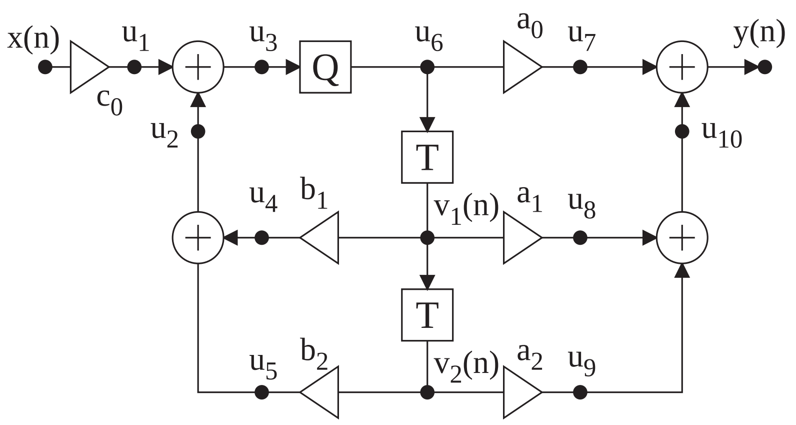
A signal-flow graph in precedence form shows the order in which the node values must be computed.

Example 6.1

The necessary quantization in the recursive loops is shown but not the ordering of the additions. Hence, the signal-flow graph is not fully specified.



We get a fully specified signal-flow graph by ordering the additions. Note that there are several ways to order the additions.



The first step, according to the procedure in Box 6.1, does not apply since, in this case, there are no multiplications by –1.

The second step is to first assign node variables to the input, output, and delay elements.

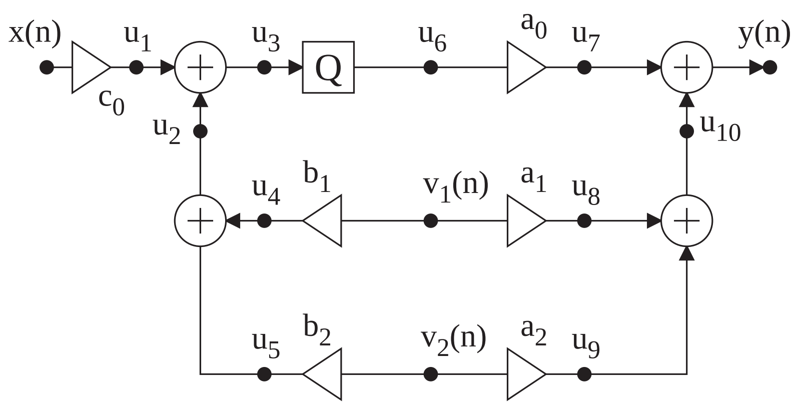
Next we identify and assign node variables to the outputs of the basic operations.

Here we assume that multiplication, two-input addition, and quantization are the basic operations.

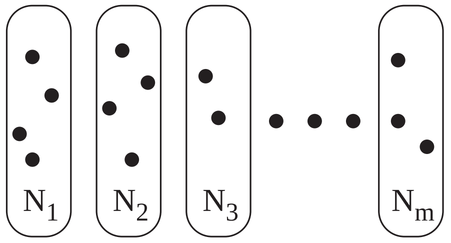
An operation can be executed when all input values are available.

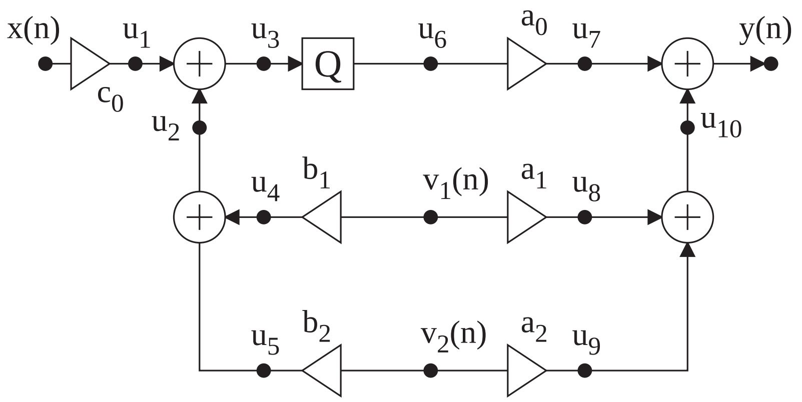
The input, $x(n)$, and the values stored in the delay elements, $v_1(n)$ and $v_2(n)$, are available at the beginning of the sample interval.

Thus, these nodes are the initial nodes for the computations. To find the order of the computations we begin (step 3) by first removing all delay branches in the signal-flow graph.



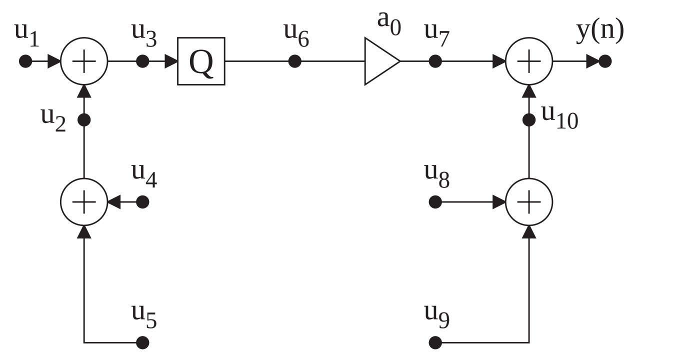
In step four, all initial nodes are identified, i.e., $x(n)$, $v_1(n)$, and $v_2(n)$. These nodes are assigned to node set N_1.



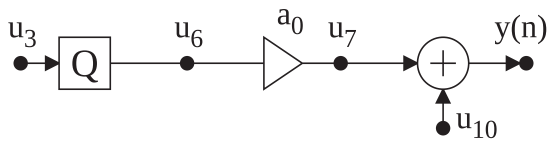


I

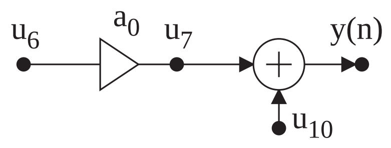
n step five we remove all executable operations, i.e., all operations that have only initial nodes as inputs. In this case we remove the five multiplications by the coefficients: c_0, a_1, a_2, b_1, and b_2. The resulting graph is



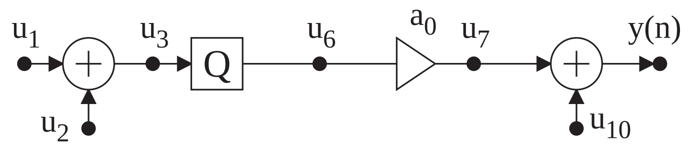
u_1
u_3
Q
u_6
a_0
u_7
y(n)
u_2
u_{10}
u_4
u_8
u_5
u_9



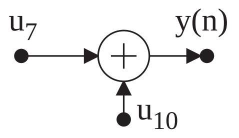
u_3
Q
u_6
a_0
u_7
y(n)
u_{10}



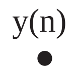
u_6
a_0
u_7
y(n)
u_{10}



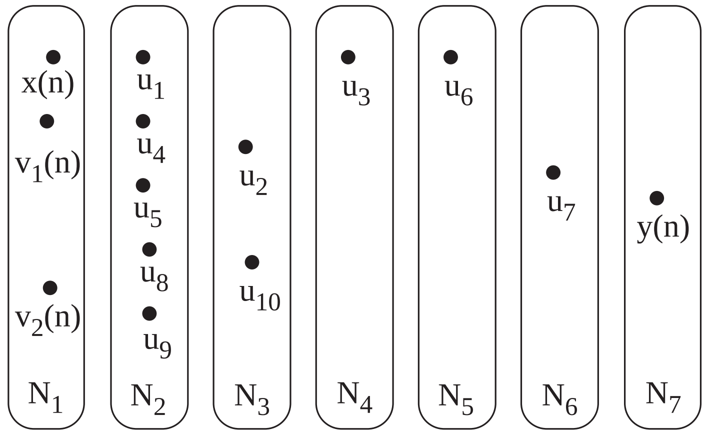
u_1
u_3
Q
u_6
a_0
u_7
y(n)
u_2
u_{10}



u_7
y(n)
u_{10}

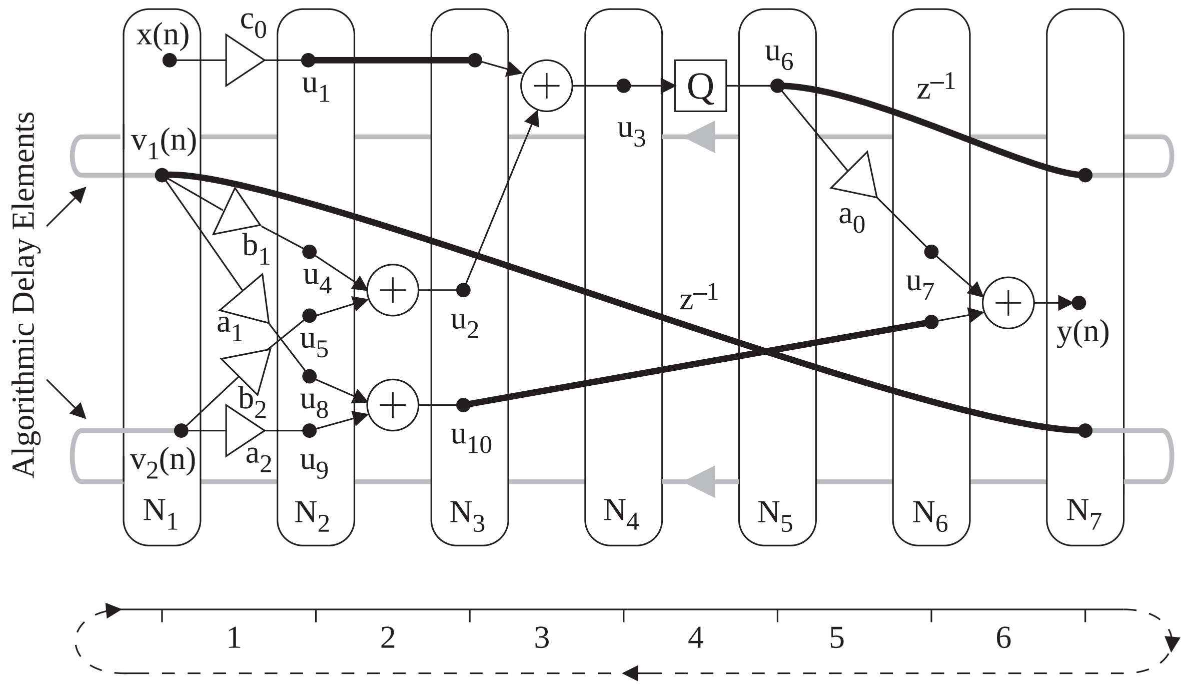


y(n)



x(n)
$v_1(n)$
$v_2(n)$
N_1
u_1
u_4
u_5
u_8
u_9
N_2
u_2
u_{10}
N_3
u_3
N_4
u_6
N_5
u_7
N_6
y(n)
N_7

Finally, we obtain the signal-flow graph in precedence form by connecting the nodes with the delay branches and the arithmetic branches according to the original signal-flow graph.



The computations take place from left to right.

A delay element is here represented by a branch running from left to right and a gray branch running from right to left.

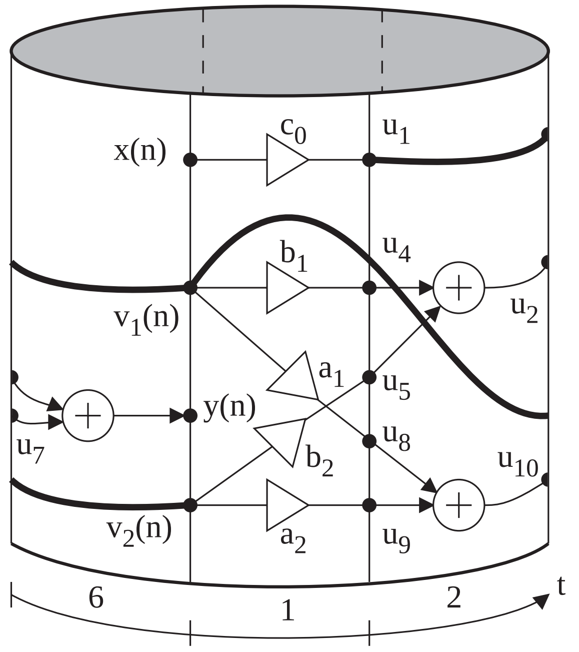
The latter indicates that the delayed value is used as input for the computations belonging to the subsequent sample interval.

If a node value, for example, u_1, is computed earlier than needed, an auxiliary node must be introduced. The branch connecting such nodes represents storing the value in memory.

It is illustrative to draw the precedence form on a cylinder to demonstrate the cyclic nature of the computations.

The computations are imagined to be performed repeatedly around the cylinder.

The circumference of the cylinder corresponds to a multiple of the length of the sample interval.



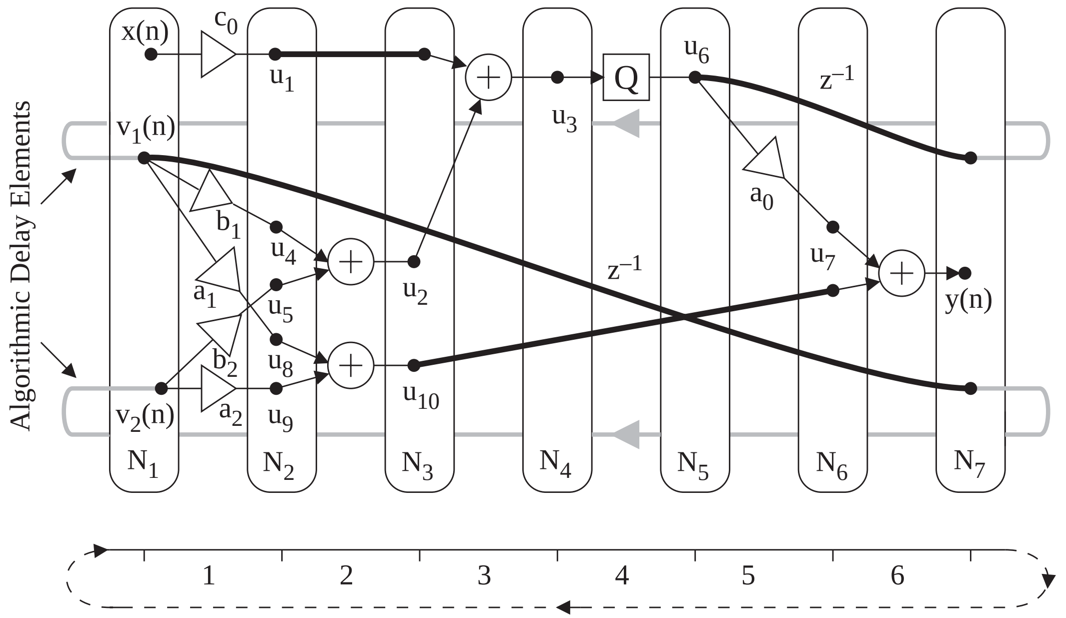
DIFFERENCE EQUATIONS

A digital filter algorithm consists of a set of difference equations to be evaluated for each input sample value.

The difference equations in computable order can be obtained directly from the signal-flow graph in precedence form.

The signal values corresponding to node set N_1 are known at the beginning of the sample interval.

Hence, operations having output nodes belonging to node set N_2 have the necessary inputs and can therefore be executed immediately.



Once this is done, those operations having output nodes belonging to node set N_3 have all their inputs available and can be executed. This process can be repeated until the last set of nodes has been computed.

Node set	**Equations**
N_2	$u_1 := c_0 x(n)$ $u_4 := b_1 v_1(n)$ $u_5 := b_2 v_2(n)$ $u_8 := a_1 v_1(n)$ $u_9 := a_2 v_2(n)$
N_3	$u_2 := u_4 + u_5$ $u_{10} := u_8 + u_9$
N_4	$u_3 := u_1 + u_2$
N_5	$u_6 := [u_3]_Q$
N_6	$u_7 := a_0 u_6$
N_7	$y(n) := u_7 + u_{10}$

Finally, the signal values corresponding to node set N_1, can be updated via the delay elements. This completes the operations during one sample interval.

Node set	Equations
N_{11}	$v_2(n+1) := v_1(n)$
N_{12}	$v_1(n+1) := u_6$

Delay elements connected in series are updated sequentially starting with the last delay element, so that auxiliary memory cells are not required for intermediate values.

Operations with outputs belonging to the different node sets must be executed sequentially while operations with outputs belonging to the same node set can be executed in parallel.

Similarly, updating of node values that correspond to delay elements must be done sequentially if they belong to different subsets of N_{1k}.

The updating is done here at the end of the sample interval, but it can be done as soon as a particular node value is no longer needed.

The system of difference equations can be obtained directly from the table.

In the method just discussed, a large number of simple expressions are computed and assigned to intermediate variables.

Hence, a large number of intermediate values are computed that must be stored in temporary memory cells and require unnecessary store and load operations.

If the algorithm is to be implemented using, for example, a standard signal processor, it may be more efficient to eliminate some of the intermediate results and explicitly compute only those node values required.

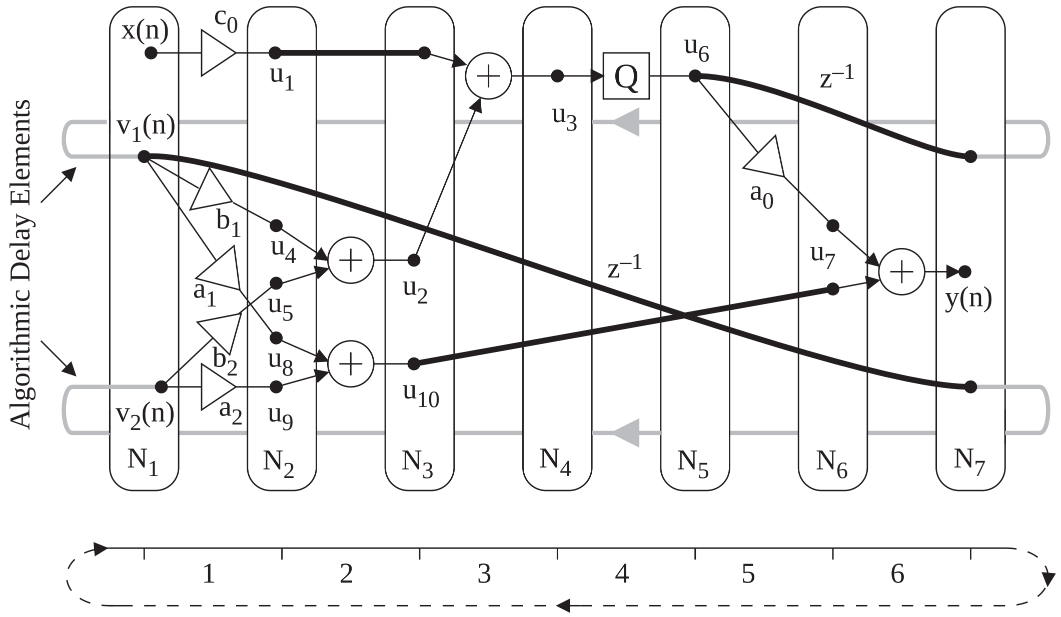
Obviously, the only values that need to be computed explicitly are

- Node values that have more than one outgoing branch
- Inputs to some types of non-commutating operations, and
- The output value.

The only node with two outgoing branches is node u_6.

The remaining node values represent intermediate values used as inputs to one subsequent operation only.

Hence, their computation can be delayed until they are needed.



$$\left\{\begin{aligned} u_3 &:= c_0 x(n) + b_1 v_1(n) + b_2 v_2(n) \\ u_6 &:= [u_3]_Q \\ y(n) &:= a_0 u_6 + a_1 v_1(n) + a_2 v_2(n) \\ v_2(n+1) &:= v_1(n) \\ v_1(n+1) &:= u_6 \end{aligned}\right.$$

In this algorithm, the only operation that is noncommutating with its adjacent operations is the quantization operation.

Generally, the inputs to such operations must be computed explicitly, but in this case u_3 appears in only one subsequent expression.

Hence, u_3 can be eliminated.

We get the following expressions that can be translated directly into a program.

$$\left\{\begin{aligned} u_6 &:= [c_0 x(n) + b_1 v_1(n) + b_2 v_2(n)]_Q \\ y(n) &:= a_0 u_6 + a_1 v_1(n) + a_2 v_2(n) \\ v_2(n+1) &:= v_1(n) \\ v_1(n+1) &:= u_6 \end{aligned}\right.$$

```
Program Direct_form_II;
const
c0 = ....; a0 = ....; a1 = ....; a2 = ....; b1 = ....; b2 = ....; Nsamples = ....;
var
    xin, u6, v1, v2, y : Real;
    i: Longint;

function Q(x: Real): Real;
    begin
        x := Trunc(x*32768);  { Wd = 16 bits => Q = 2^-15 }
        if x > 32767 then x := 32767;
        if x < -32768 then x := -32768;
        x := x/32768;
    end;
begin
    for i := 1 to NSamples do
        begin
            Readln(xin); { Read a new input value, xin }
            u6 := Q(c0*xin + b1*v1 + b2*v2);  { Direct form II }
            y := a0*u6 + a1*v1 + a2*v2;
            v2 := v1;
            v1 := u6;
            Writeln(y);    { Write the output value }
        end;
end.
```

COMPUTATION GRAPHS

It is convenient to describe the computational properties of an algorithm with a *computation graph* that combines the information contained in the signal-flow graph with the corresponding precedence graph for the operations.

The computation graph will later be used as the basis for scheduling the operations. Further, the storage and communication requirements can be derived from the computation graph.

In this graph, the signal-flow graph in precedence form remains essentially intact, but the branches representing the arithmetic operations are also assigned appropriate execution times.

Branches with delay elements are mapped to branches with delay.

Additional branches with different types of delay must often be inserted to obtain consistent timing properties in the computation graph.

These delays will determine the amount of physical storage required to implement the algorithm.

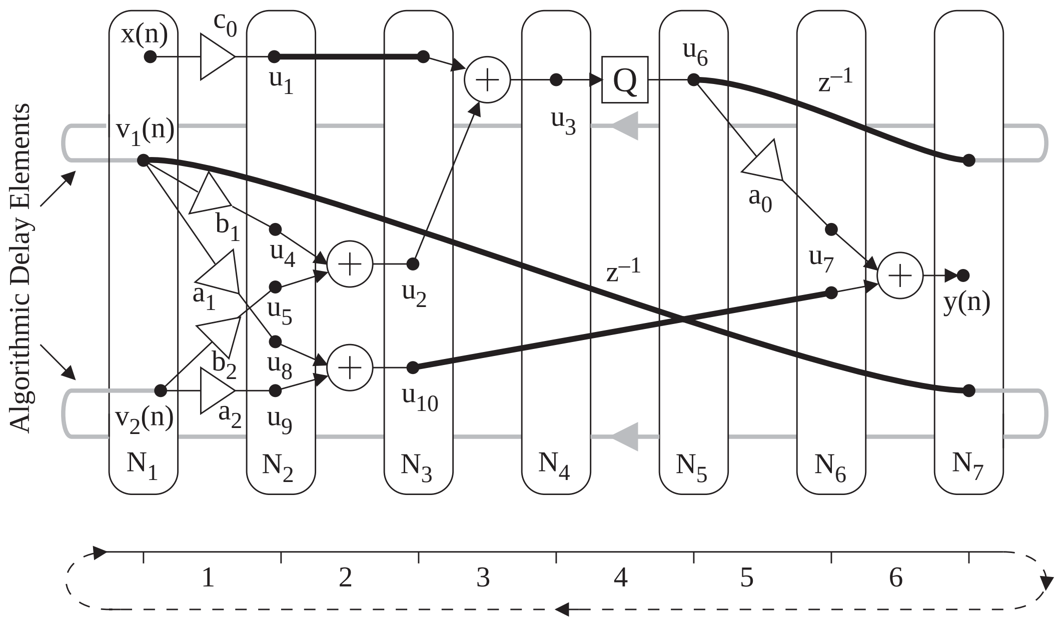
Critical Path

By assigning proper execution times to the operations, represented by the branches in the precedence graph, we obtain the computation graph.

The longest (time) directed path in the computation graph is called the *critical path* (*CP*) and its execution time is denoted T_{CP}.

Several equally long critical paths may exist. For example, there are two critical paths in the computation graph.

The first CP starts at node $v_1(n)$ and goes through nodes u_4, u_2, u_3, u_6, u_7, and $y(n)$ while the second CP begins at node $v_2(n)$ and goes through nodes u_5, u_2, u_3, u_6, u_7, and $y(n)$.



Equalizing Delay

Assume that an algorithm shall be executed with a sample period T and the time taken for the arithmetic operations in the critical path is T_{CP}, where $T_{CP} < T$. Then additional delay, which is here called *equalizing delay*, has to be introduced into all branches that cross an arbitrary vertical cut in the computation graph.

This delay accounts for the time difference between a path and the length of the sample interval. The required duration of equalizing delay in each such branch is

$$T_e = T - T_{CP}$$

The amount of equalizing delay, which usually corresponds to physical storage, can be minimized by proper scheduling of the operations.

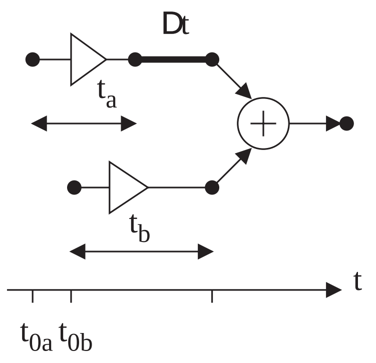
Shimming Delay

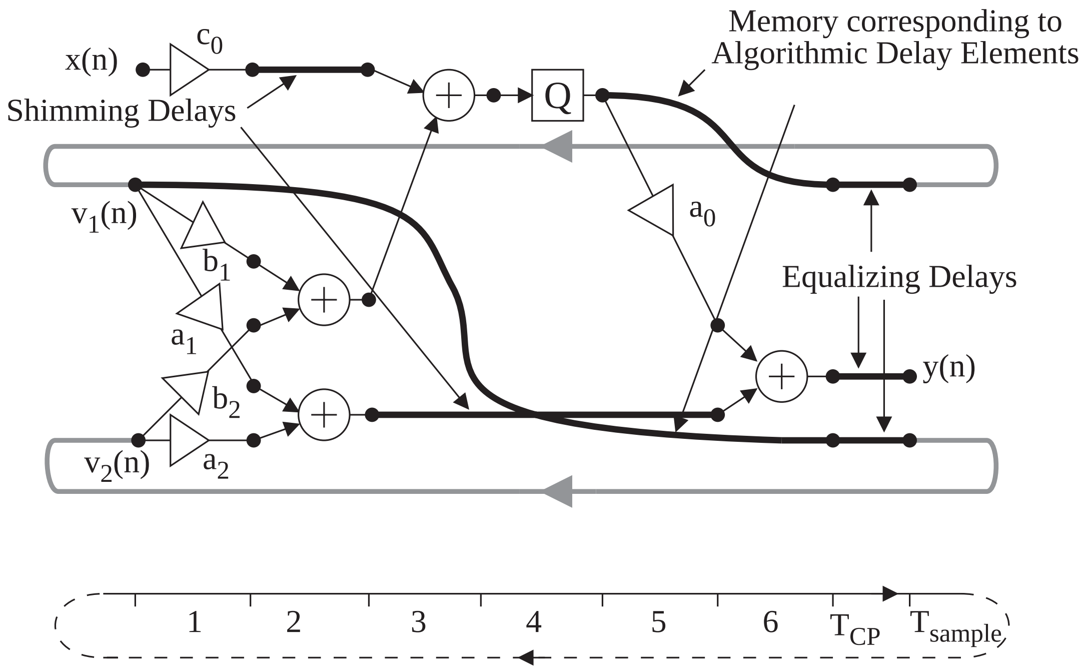
If two paths in a computation graph have a common origin and a common end node, then the data streams in these two paths have to be synchronized at the summation node by introducing a delay in the fastest path.

For example, execution of the two multiplications may be started at time instances t_{0a} and t_{0b}, and the times required for multiplication are t_a and t_b, respectively. This means that the products will arrive at the inputs of the subsequent adder with a time difference

$$Dt = (t_{0b} + t_b) - (t_{0a} + t_s)$$

A delay must therefore be inserted in the upper branch of the computation graph so that the products arrive simultaneously. These delays are called *shimming delays* or *slack*. Shimming delay usually correspond to physical storage.




x(n)
c_0
Shimming Delays
Q
Memory corresponding to Algorithmic Delay Elements
$v_1(n)$
b_1
a_1
b_2
$v_2(n)$
a_2
a_0
Equalizing Delays
y(n)
1
2
3
4
5
6
T_{CP}
T_{sample}

Maximal Sample Rate

The maximal sample rate of an algorithm is determined only by its recursive parts.

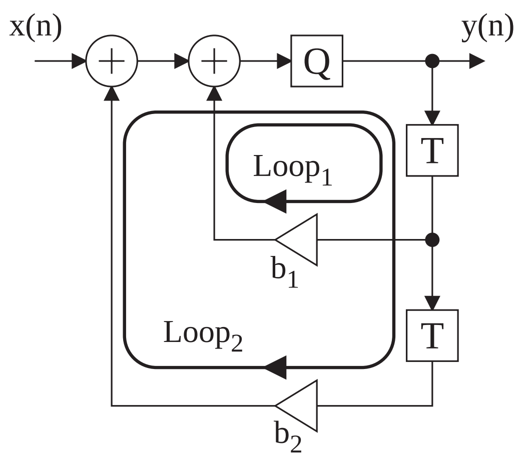
The minimal sample period for a recursive algorithm that is described by a fully specified signal-flow graph is

$$T_{min} = \max_i \left\{ \frac{T_{opi}}{N_i} \right\}$$

where T_{opi} is the total latency of the arithmetic operations, etc. and N_i is the number of delay elements in the directed loop i.

Nonrecursive parts of the signal-flow graph, e.g., input and output branches, generally do not limit the sample rate, but to achieve this limit additional delay elements may have to be introduced into the nonrecursive branches.

The minimal sample period is also referred to as the *iteration period bound*.



Loops that yield T_{min} are called *critical loops*.

The signal-flow graph has two loops. Loop 1 is the critical loop if

$$\frac{T_{b2} + 2T_{add} + T_Q}{2} < \frac{T_{b1} + T_{add} + T_q}{1}$$

otherwise loop 2 is the critical loop.

The iteration period bound is of course not possible to improve for a given algorithm. However, a new algorithm with a higher bound can often be derived form the original algorithm.

We recognize that there are two possibilities to improve the bound.

- Reduce the operation latency in the critical loop.
- Introduce additional delay elements into the loop