SCHEDULING

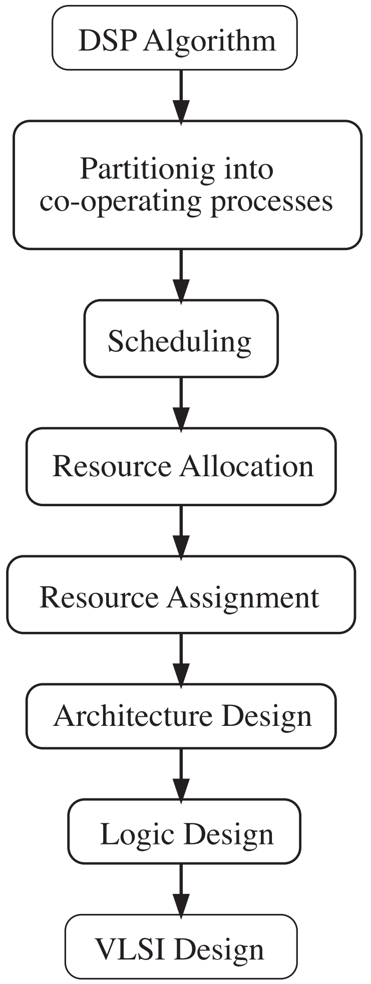
So much to schedule, so little time.

A process that is allowed to be halted while it is executed and continued in another PE, or in the same PE at a later stage, is called a *preemptive process*, in contrast to the opposite, a *non-preemptive process*.

A preemptive process can be split into smaller processes.

Processes to be scheduled are characterized by their start or release time, duration, or deadline, and they may arrive at random time instances or periodically.

Hard real-time processes are processes for which the deadline must be met.



If all processes are known in advance, it is possible to perform scheduling before run time, yielding a *static schedule* that will not change.

Static schedules need only be done once. Conversely, a *dynamic schedule* must be used whenever the process characteristics are not completely known.

The following five criteria of optimality are commonly used for static scheduling.

Sample period

Power consumption

Resource optimal

- Processor optimal

- Memory optimal

Delay optimal

SCHEDULING FORMULATIONS

The design problem of interest here is to find an operation schedule that allows the operations to be mapped to a minimum-cost hardware structure that meets the performance constraints.

In this section we will discuss the following formulations of the scheduling problems:

- Single interval formulation
- Block formulation
- Loop-folding
- Periodic formulation

The aim of the scheduling is to minimize a *cost function*.

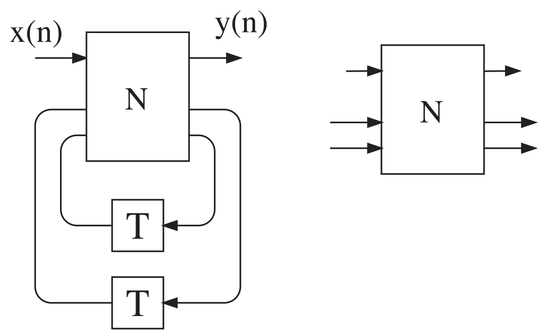
The most commonly used cost function is the number of processors.

However, it is also important to include other hardware resources that consume significant amounts of chip area and power.

Single Sample Interval Scheduling Formulation

The starting point for scheduling is the computation graph derived from the fully specified signal-flow graph.

The first step is to extract all delay elements. The remaining part, which contains only the arithmetic operations and delays, is denoted by *N*. The computation graph *N* is a *DAG* (*directed acyclic graph*).

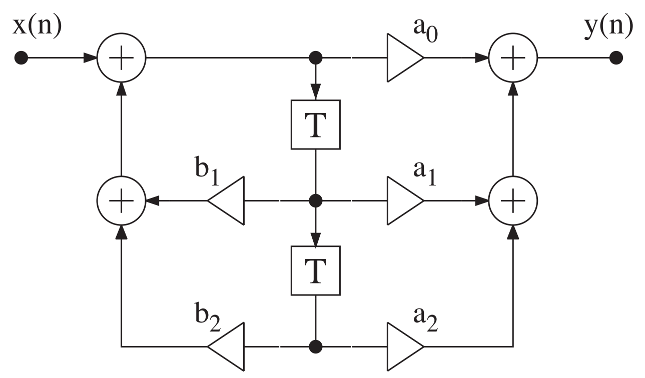


It appears that an analysis of this graph, which represents the arithmetic operations during one sample interval, would reveal the computational properties of the algorithms, such as the parallelism and the maximal sample rate, etc.

However, this is not the case. In fact, it is not sufficient to consider the operations during one sample interval only. We will demonstrate the shortcomings of considering only one sample interval with an example.

Example 7.1

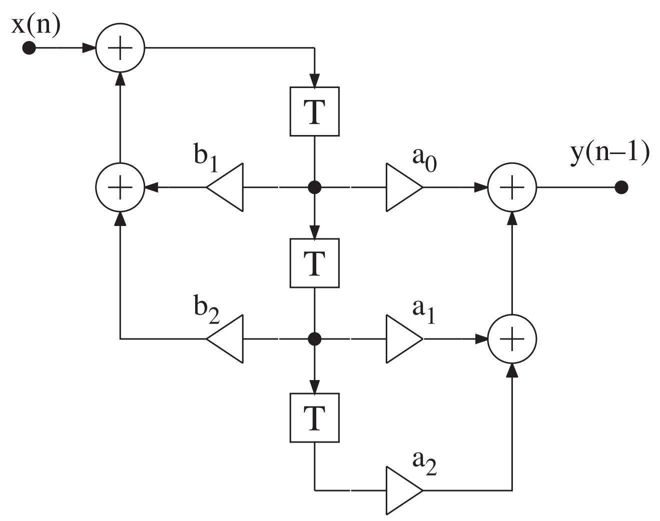
Consider the second-order section in direct form II. The basic operations are multiplication and addition. T_{mult} = 4 time units and T_{add} = 1 time unit. The required sample period is $T_{sample} = 10$ time units.

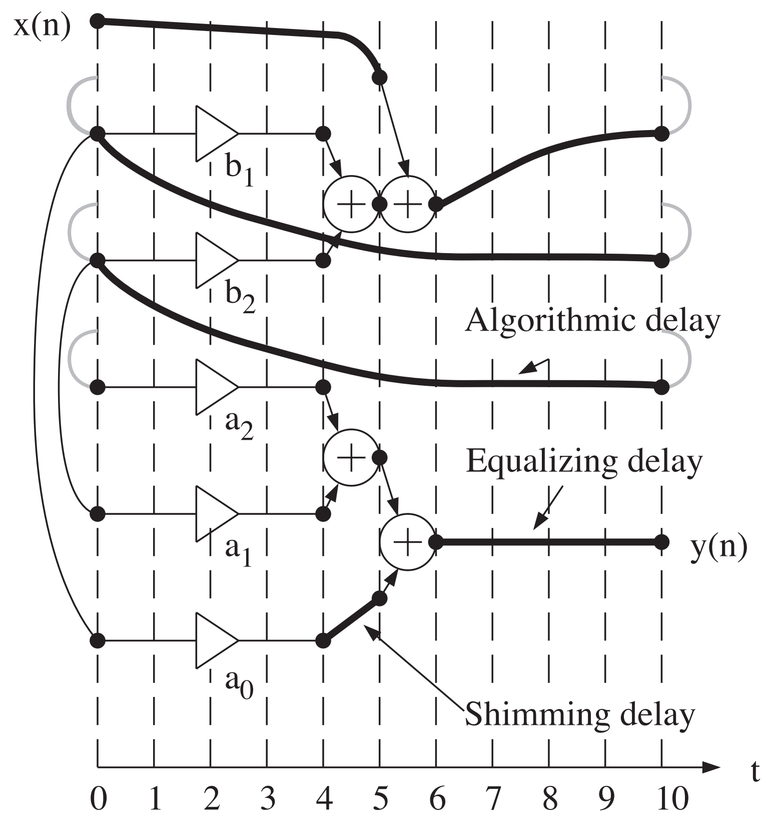


The critical loop and the critical path of this algorithm are $T_{mult} + 2T_{add} = 6$ and $2T_{mult} + 3T_{add} = 11$ time units, respectively.

Hence, to achieve the desired sample rate, we must either use interleaving or introduce delays to divide the critical path into smaller pieces (pipelining).

We choose to introduce pipelining.

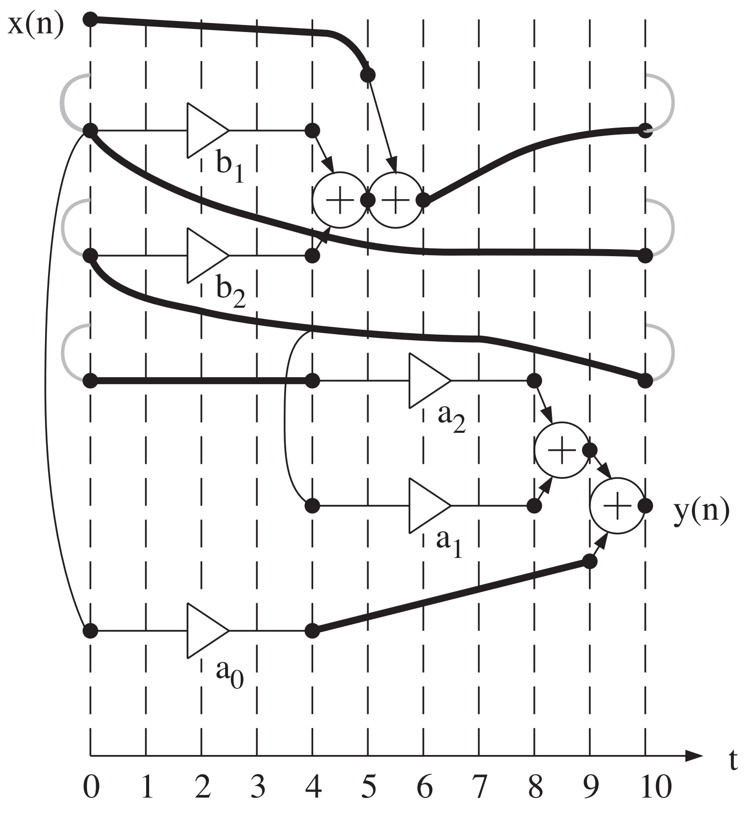




Computation graph

The computation graph can be directly interpreted as a schedule for the arithmetic operations. Five multipliers and two adders are needed. The number of time units of storage is 34.

Clearly, a better schedule can be found. The operations can be scheduled to minimize the number of concurrent operations of the same type. The scheduling is done by interchanging the order of operations and delays in the same branch.



The amount of storage has increased to 38 time units while the numbers of multipliers and adders have been reduced to three and one, respectively.

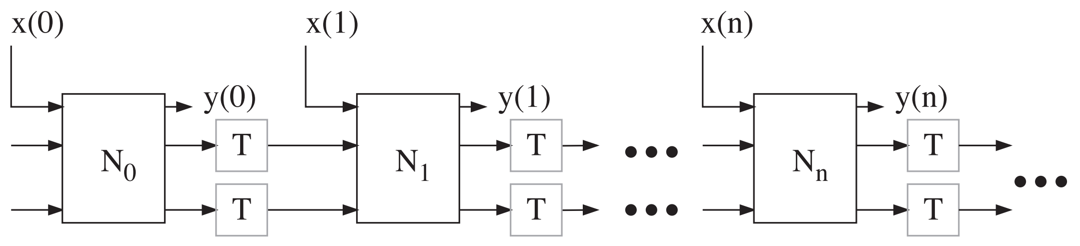
It is obvious that there are many possible schedules that require the same number of processing elements, but have different requirements on storage and communication resources, i.e., communication channels between processing elements and memories.

Note that the average number of multipliers required is only $(5 \cdot 4)/10 = 2$.

This suggests that it may be possible to find a more efficient schedule that uses only two multipliers.

However, it is not possible to find such a schedule while considering the operations during a single sample interval only.

Block Scheduling Formulation



The computation graph contains a number of delay-free paths of infinite length, since the delay elements just represent a renaming of the input and output values.

The longest average path is the *critical path, CP*. Note that the input $x(n)$ and the output $y(n)$ do not belong to the critical path. The average computation time of the *CP* is equal to the iteration period bound.

The block scheduling formulation allows the operations to be freely scheduled across the sample interval boundaries. Hence, inefficiency in resource utilization due to the artificial requirement of uniform scheduling boundaries can be reduced. The price is longer schedules that require longer control sequences.

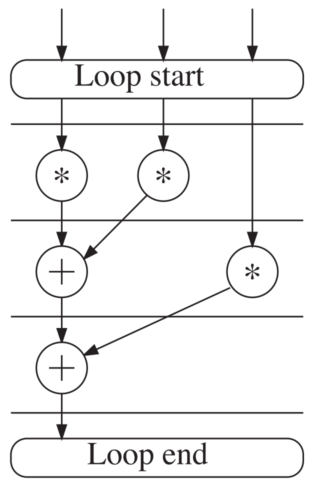
Loop-Folding

Operations inside a (possibly infinite) “for”-loop correspond to the operations within one sample interval in a DSP algorithm.

Loop-folding techniques have been used for a long time in software compilers to increase the throughput of multiprocessor systems.

Loop-folding can be used to increase the throughput or reduce the cost of the implementation.

Consider the kernel of a loop containing three multiplications and two additions.



The loop has a critical path of length three and the operations are scheduled in three different control steps, i.e., the sample period is three time units.

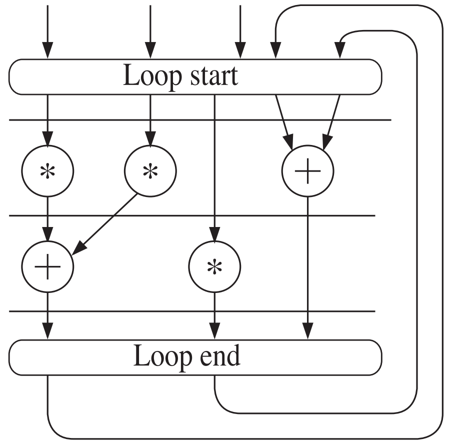
We need two multipliers and one adder. It is not possible to reschedule operations within the loop to further reduce the number of resources.

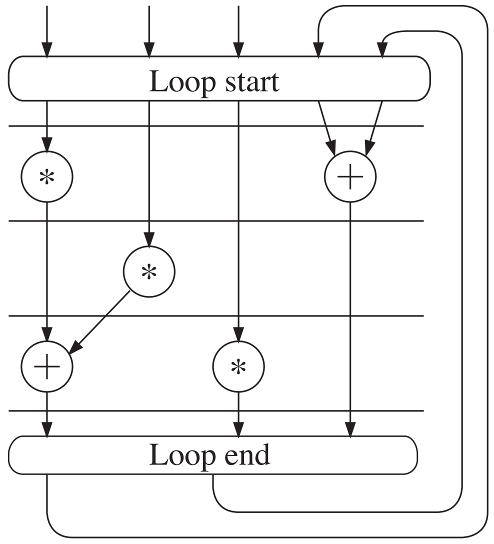
The operations are divided into two sections so that the critical paths are broken into pieces of length one and two.

There are two main options: we can either *minimize loop time* (sample period) or *minimize the resources*.

In the second case, *loop-unfolding*, the number of PEs is minimized.

Only one multiplier and one adder are required. The drawback is that latency from input to output has been increased from the original three to six.





Loop-folding is equivalent to recognizing that some operations belonging to a sample interval can be started earlier than others and be executed at the same time as operations from previous intervals.

We do not have to execute all operations in an interval before we start to execute the operations belonging the next interval.

This resembles scheduling a cyclic graph instead of a DAG.

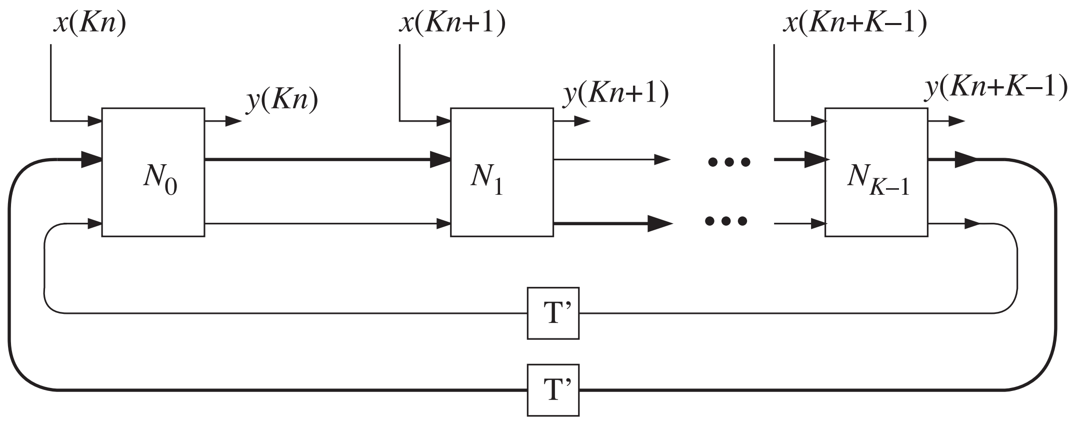
Loop-unfolding is a simplified version of the more general cyclic scheduling formulation which is described in the next section.

Cyclic Scheduling Formulation

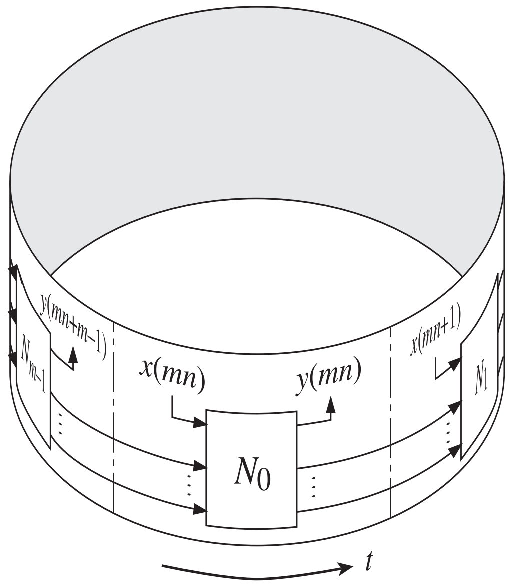
In the single interval and blocking formulations we did not take into account the fact that the schedule is inherently periodic.

It is therefore convenient to connect K computation graphs in a closed loop. This cyclic formulation will under certain conditions result in a maximally fast schedule.

This is the most general scheduling formulation and it can be used to find resource-optimal schedules using arbitrary constraints on, for example, sample rate and latency.



It may help to imagine the computation graphs drawn on a cylinder of circumference equal to the scheduling period (which is a multiple of the sample period).



Unfortunately, it is not possible to determine the best choice of K in the general case.

In order to attain the minimum sample period, it is necessary to perform cyclic scheduling of the operations belonging to several successive sample intervals if

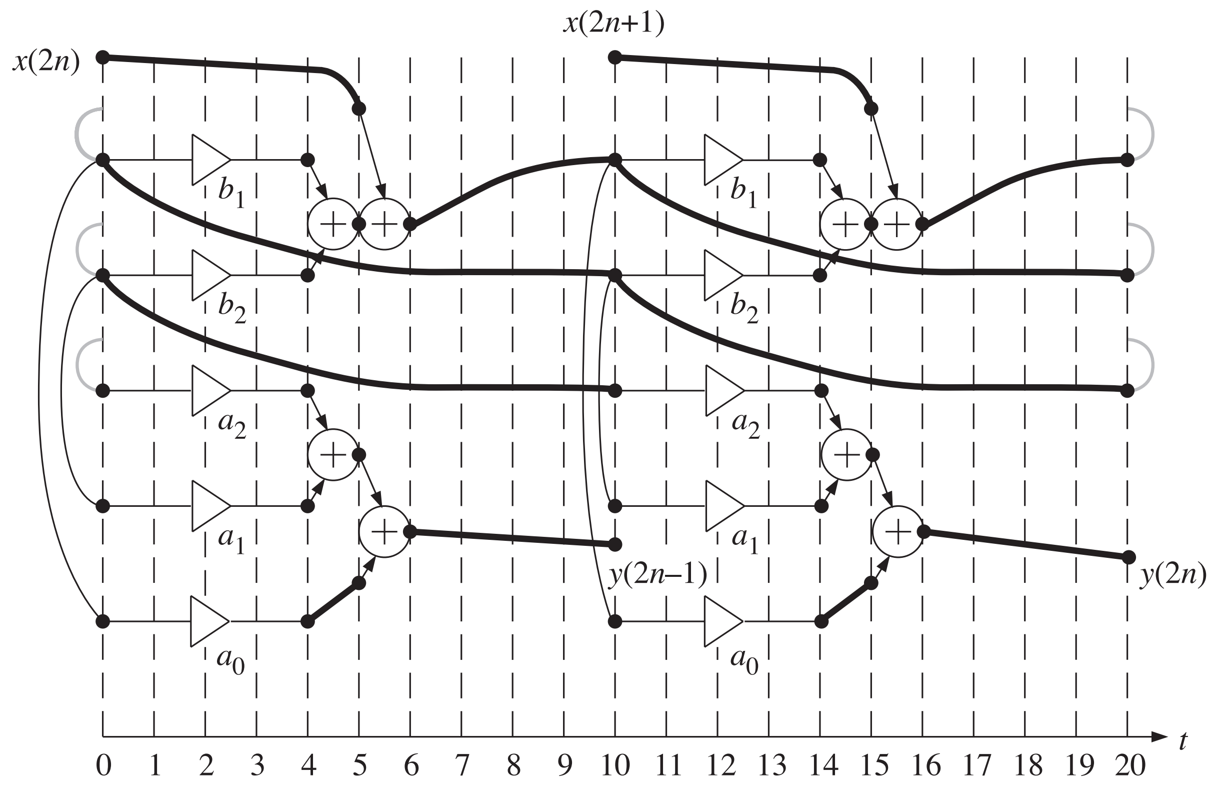
The computation time for a PE is longer than T_{min} or

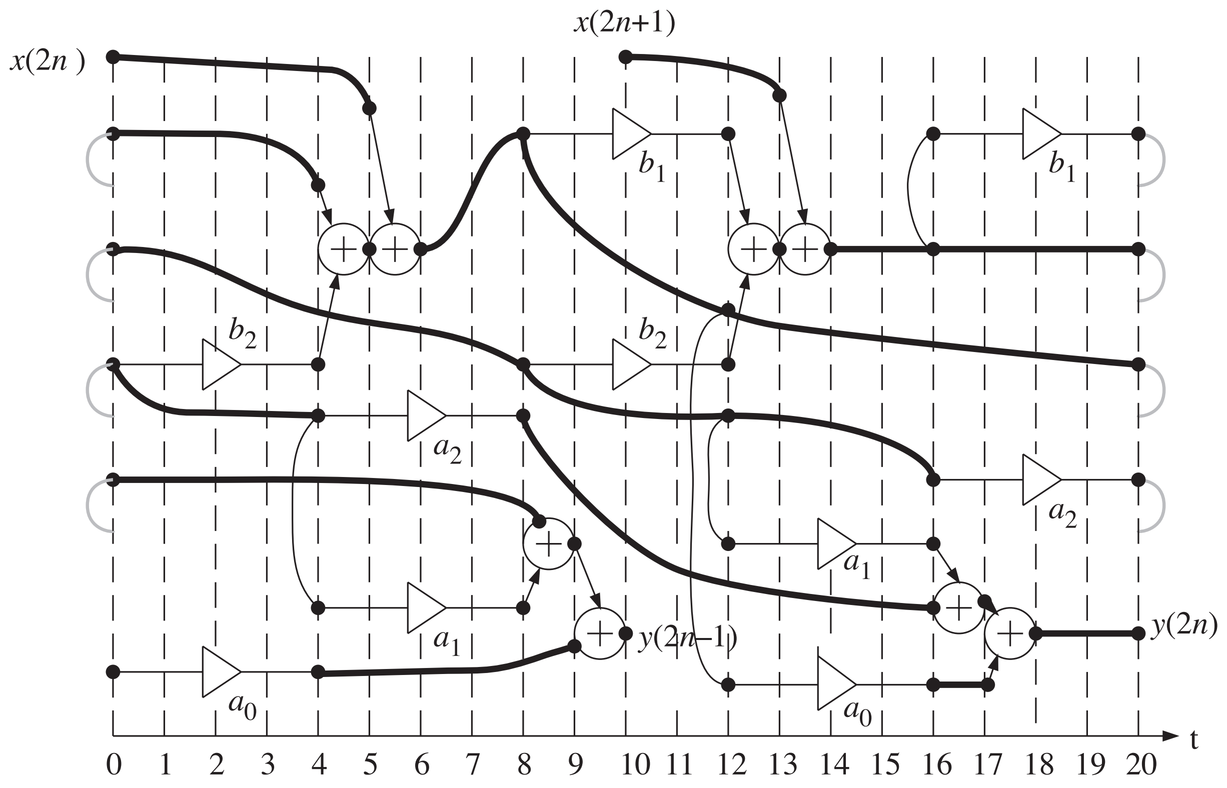
The critical loop(s) contains more than one delay element.

Generally, the critical loop should be at least as long as the longest execution time for any of the PEs in the loop.

For bit-serial arithmetic, the execution times of the processing elements are normally longer than the minimum sample period.

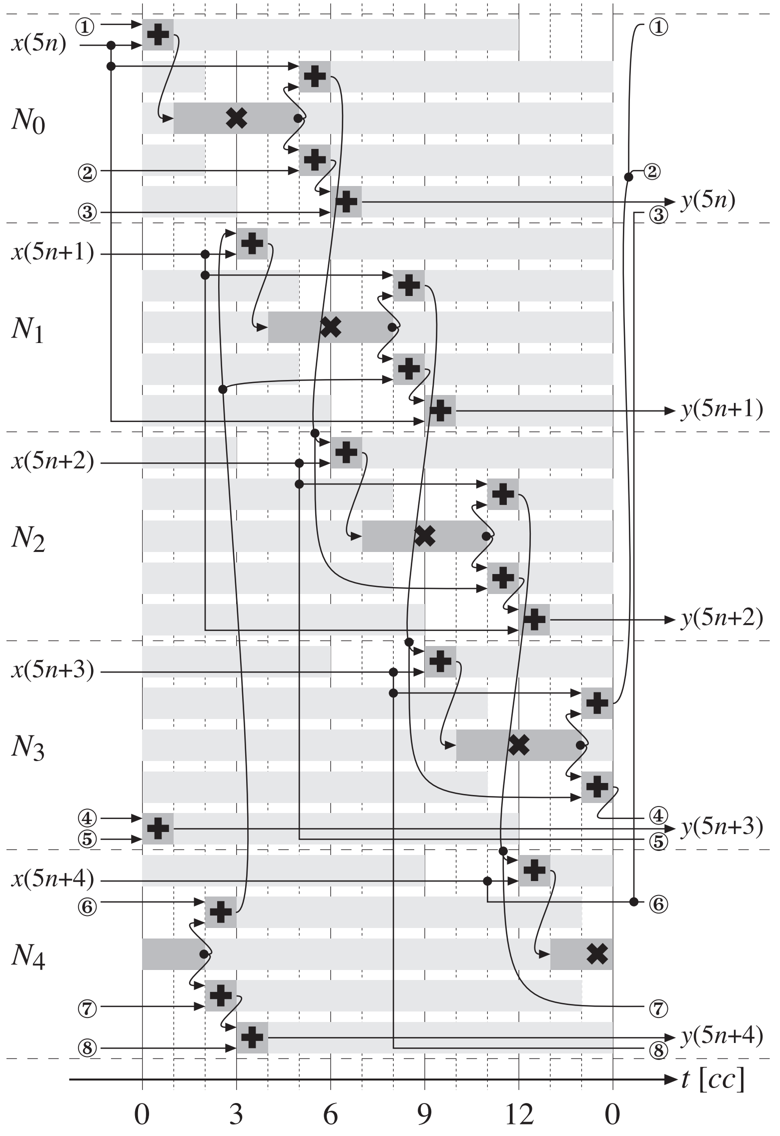
Example 7.2


x(2n)
x(2n+1)
b1
b2
a2
a1
a0
y(2n–1)
y(2n)
t
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

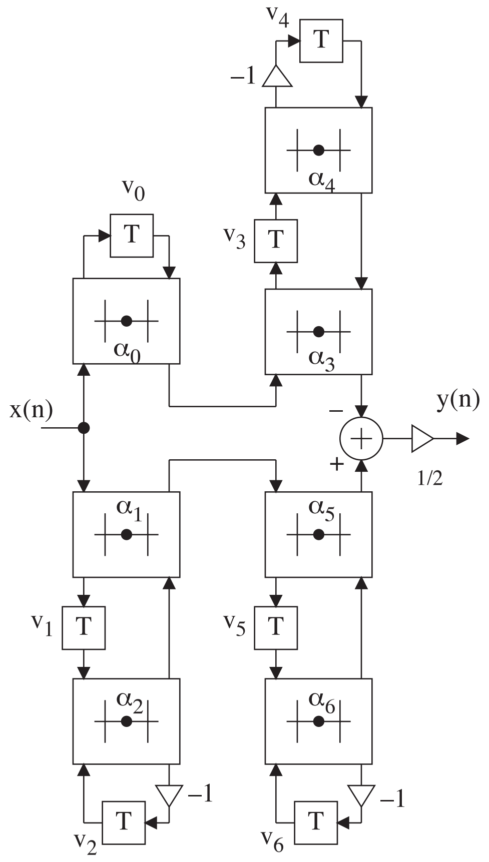


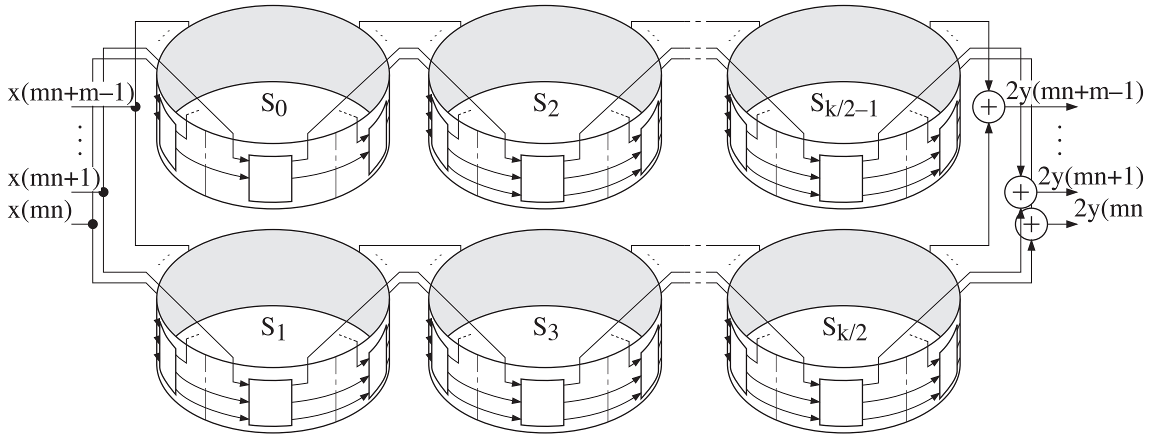
x(2n)
x(2n+1)
b1
b2
a2
a1
a0
y(2n–1)
y(2n)
t
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Example 7.3


x(5n)
N0
x(5n+1)
N1
x(5n+2)
N2
x(5n+3)
N3
x(5n+4)
N4
①
②
③
④
⑤
⑥
⑦
⑧
y(5n)
y(5n+1)
y(5n+2)
y(5n+3)
y(5n+4)
t [cc]
0
3
6
9
12
0

Scheduling of Lattice Wave Digital Filters





SCHEDULING ALGORITHMS

Scheduling algorithms are basically combinatorial optimization techniques.

They search the solution space for a solution with favorable properties. Combinatorial optimization methods can be characterized as

- Heuristic methods
  - Constructive methods
  - Iterative methods
- Non-heuristic methods

Heuristic methods use a set of rules that limit the search in the solution space in order to shorten the required search time.

This is often necessary since most scheduling problems have been shown to be NP-complete. A drawback of heuristic methods is that they may get stuck in local optima.

Constructive methods construct a solution.

Iterative methods produce better solutions iteratively from older ones.

Constructive methods can be used to find an initial solution for an iterative method. Both these types of methods traverse only a part of the solution space and yield suboptimal solutions.

Conversely, *nonheuristic methods* will find the optimal solution, but they will, in general, require excessive run times since they traverse the whole solution space.

Exhaustive searches can be made effectively by depth-first or breadth-first approaches.

The following optimization techniques are commonly used for scheduling:

ASAP and ALAP scheduling

Earliest deadline and slack time scheduling

Linear programming methods

Critical path list scheduling

Force-directed scheduling

Cyclo-static scheduling

Simulated annealing

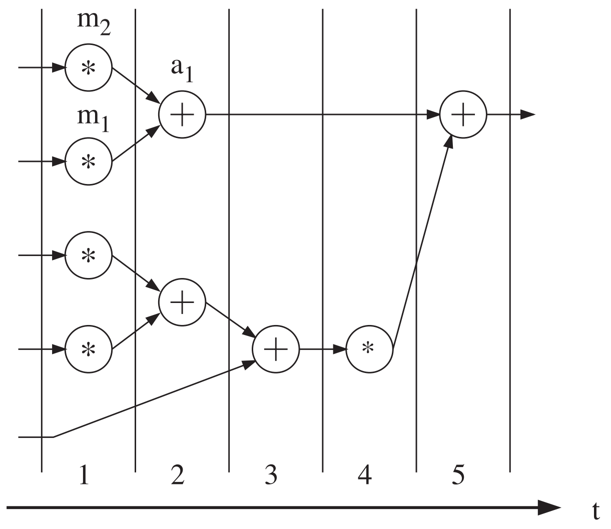
Genetic algorithms

ASAP Scheduling

ASAP (as soon as possible) scheduling simply schedules operations as soon as possible.

A computation can be performed as soon as all of its inputs are available—i.e., all predecessors have been performed.

The aim is to obtain the shortest possible execution time without considering resource requirements.



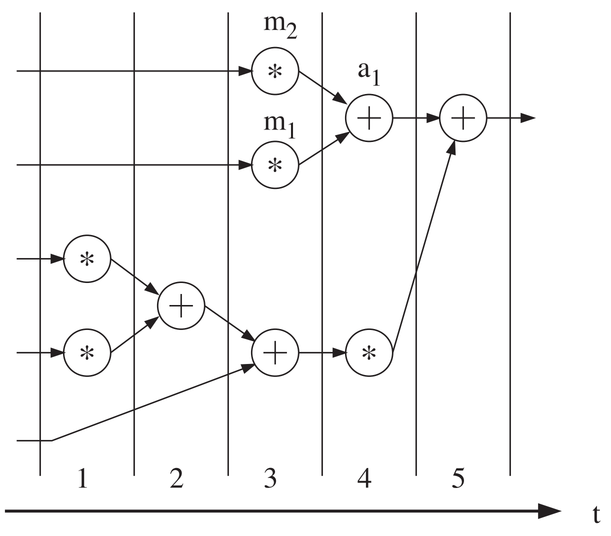
ALAP Scheduling

ALAP (as late as possible) is a method similar to ASAP.

In ALAP the operations are scheduled as late as possible.

ASAP and ALAP scheduling are often used to determine the time range in which the operations can be scheduled.

This range is often referred to as *life-span*.



For example, multiplications m_1 and m_2 and addition a_1 are the only operations that can be rescheduled.

The reduction in life-spans of the operations results in a significant reduction of the size of the scheduling problem.

Earliest Deadline and Slack Time Scheduling

These methods can be used to schedule periodically or randomly occurring processes.

The *earliest deadline* scheduling algorithm schedules, in each time step, the processes whose deadline is closest.

In general, this scheduling technique requires that the execution of the process may be preempted.

The *slack time* algorithm resembles the earliest deadline algorithm.

In each time slot, it schedules the process whose slack-time is least. Slack-time is the time from the present to the deadline minus the remaining processing time of a process.

Linear Programming

The scheduling problem is a combinatorial problem that can be solved by *integer linear programming* (*ILP*) methods.

These methods (for example, the *simplex method* and the *interior point methods*) find the optimal value of a linear cost function while satisfying a large set of constraints.

The precedence relations for the operations to be scheduled represent a set of inequalities in the form of nonfeasible time intervals.

Using linear programming methods, the optimal solution can be obtained. Problems with a few thousand constraints and several thousand variables can be tackled on a small PC while workstations can often handle problems with variables in the tens of thousands or even greater.

However, very large problems with many operations might be intractable due to NP-completeness of the scheduling problem and the cost function is being limited to a linear function.

Linear programming methods are nonheuristic.

Critical Path List Scheduling

Critical path list scheduling is related to ASAP scheduling.

It is one method in the class of list scheduling methods.

The first step in list scheduling methods is to form an ordered list of operations to be performed.

Secondly, the operations are picked one by one from this list and assigned to a free resource (PE).

In critical path list scheduling, the list is formed by finding the critical paths in the DAG of the algorithm.

Operations are ordered according to path length from the start nodes in the DAG.

Other forms of list scheduling methods are obtained by using different heuristics for scheduling the operations that do not belong to the critical paths.

Force-Directed Scheduling

The force-directed scheduling method attempts to distribute the operations in time so that the resources required are minimized under a fixed execution time constraint.

Cyclo-Static Scheduling

The theory of cyclo-static processor schedules was developed by Schwartz and Barnwell. The method is suitable for systolic arrays.

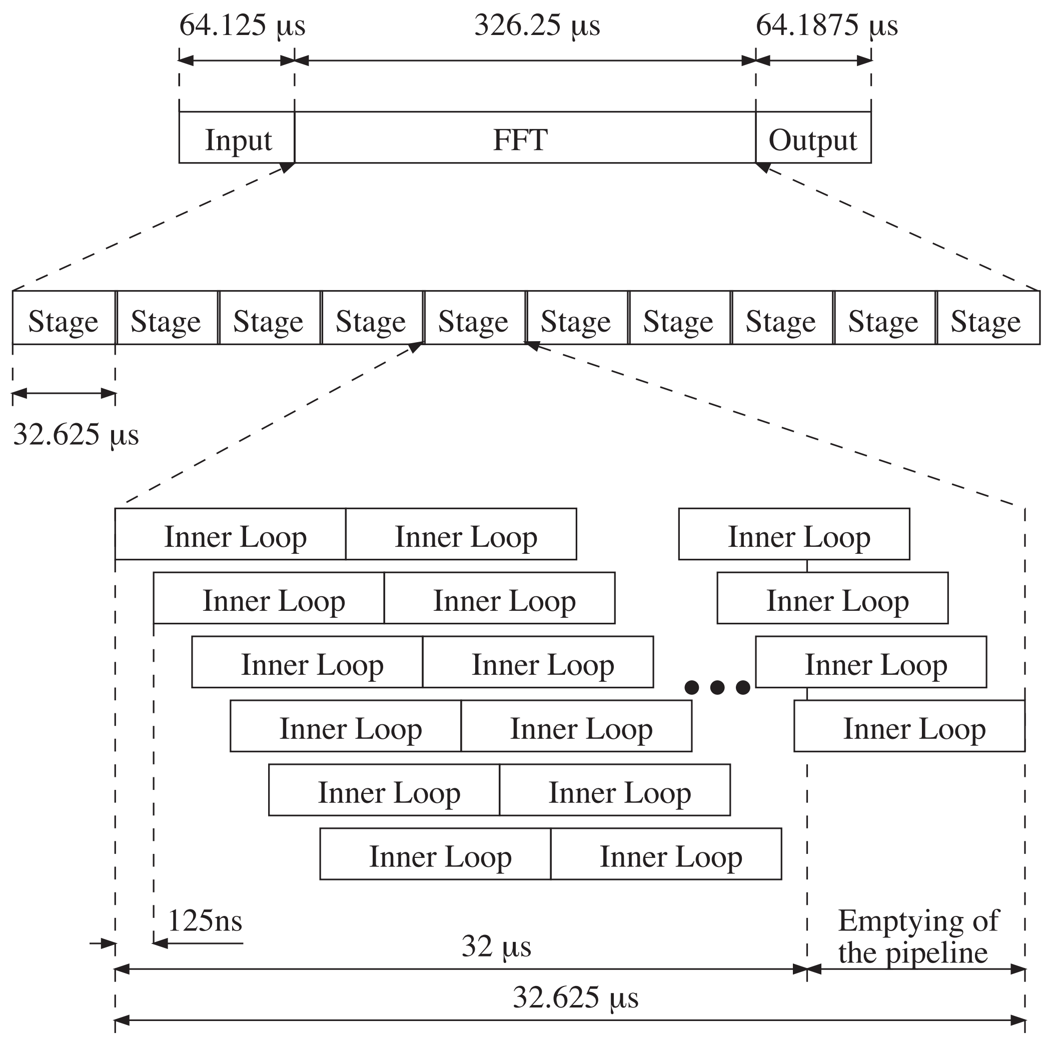
Maximum Spanning Tree Method

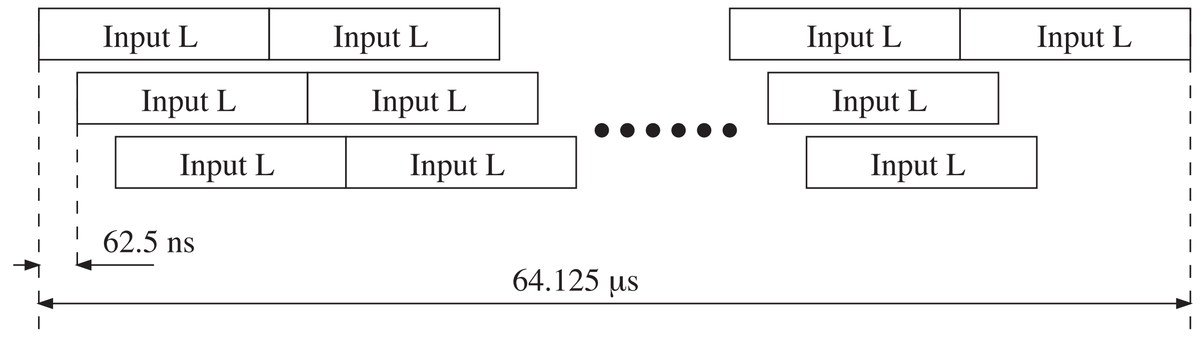
The maximum spanning tree method, which was developed by Renfors and Neuvo, can be used to achieve rate optimal schedules.

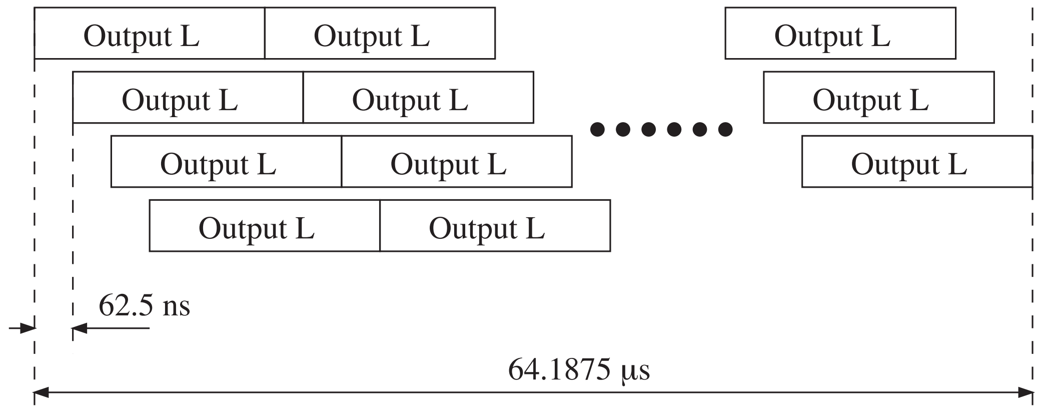
The method is based on graph-theoretical concepts. The starting point is the fully specified SFG.

Simulated Annealing and Genetic Algorithms

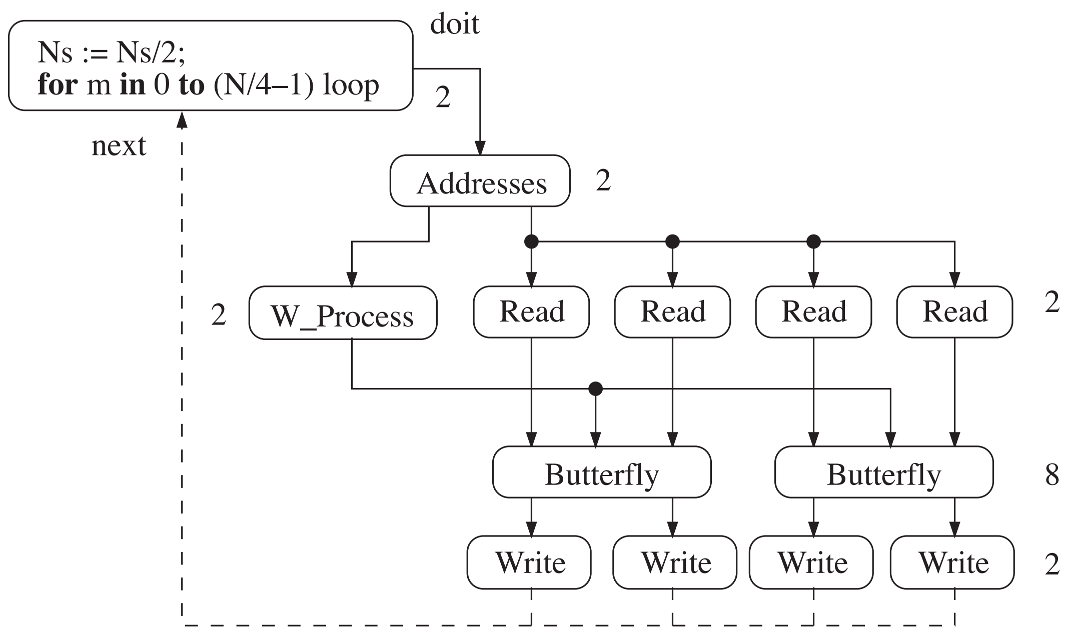
FFT PROCESSOR, CONT.

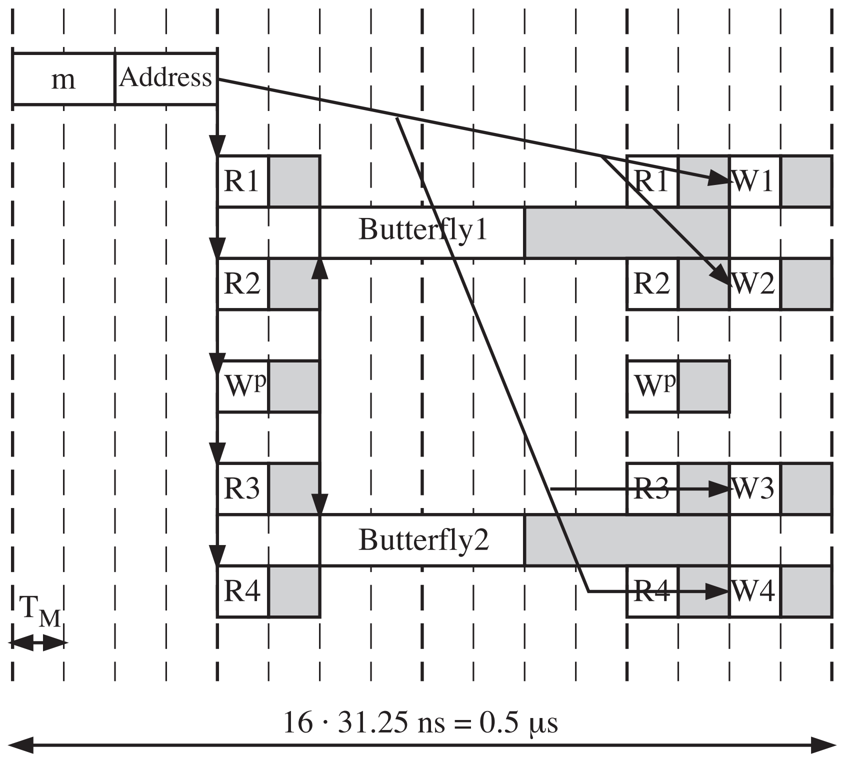



Input L
Input L
Input L
Input L
Input L
Input L
Input L
Input L
Input L
Input L
62.5 ns
64.125 µs

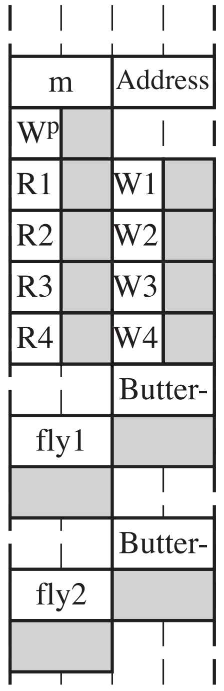

Output L
Output L
Output L
Output L
Output L
Output L
Output L
Output L
Output L
Output L
Output L
62.5 ns
64.1875 µs

Scheduling of Inner Loops



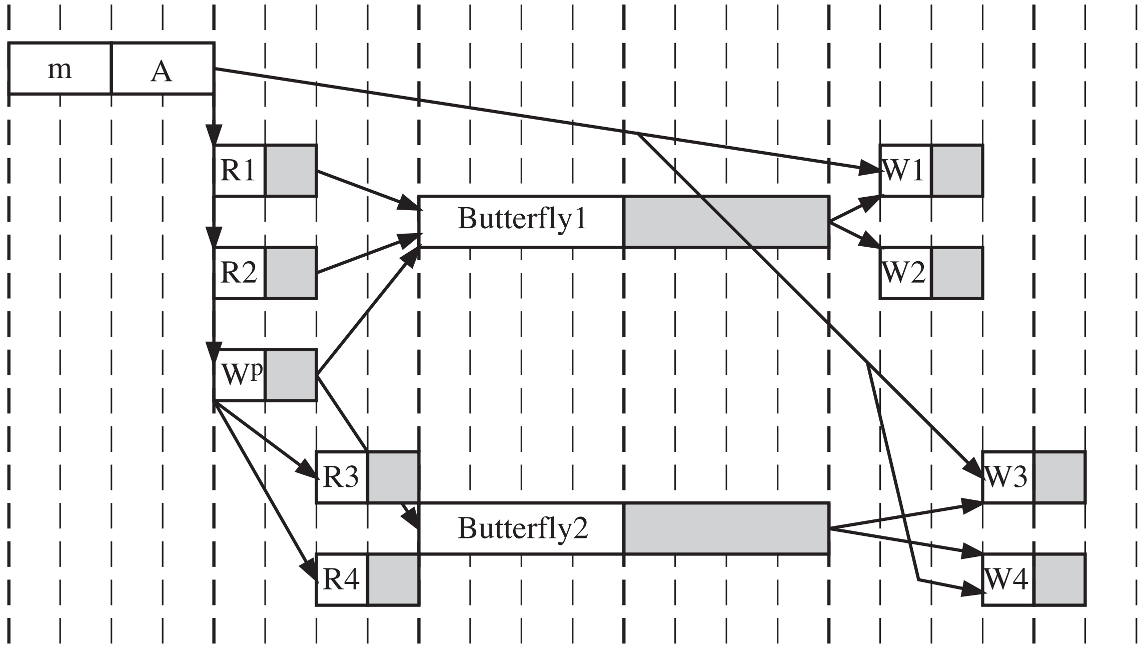


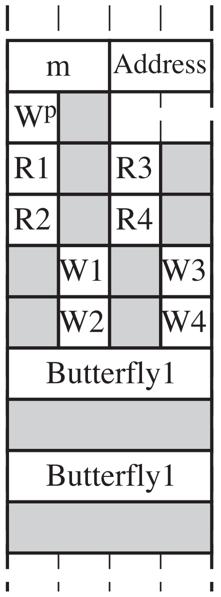
ASAP schedule of the inner loop



Folded schedule

Obviously, this memory access schedule is unacceptable since there are four concurrent read or write operations. This schedule therefore requires four memories. The memory accesses are not evenly distributed within the folded schedule.





Improved folded schedule