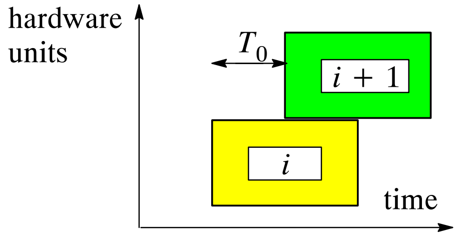
TOPICS

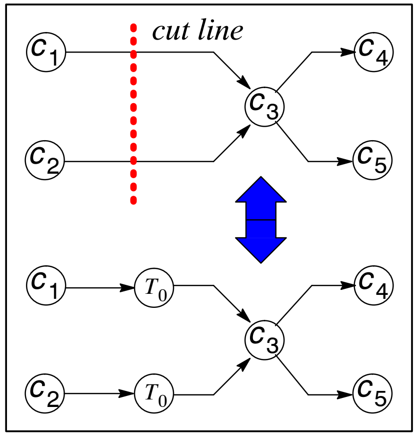
* Pipelining
* Retiming
* Parallel processing
* Loop unrolling
* Unfolding
* Look-ahead transformation

SPEED-UP TECHNIQUES: PIPELINING

Insert delay elements on all edges that are cut by a *cut line* through an edge of the critical path in the DFG.

* Works for acyclic DFGs.
* Schedule becomes overlapped.

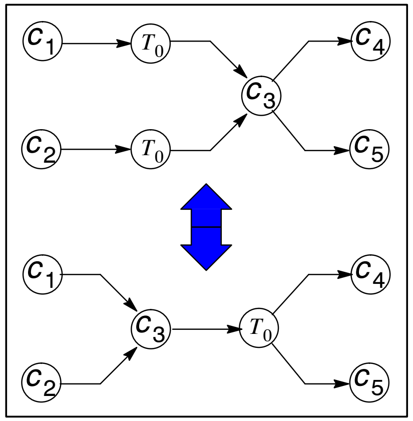




Example

SPEED-UP TECHNIQUES: RETIMING

* Useful for obtaining the minimal T_0 for a nonoverlapped schedule by reduction of critical-path length, both for cyclic and acyclic IDFGs.



Example

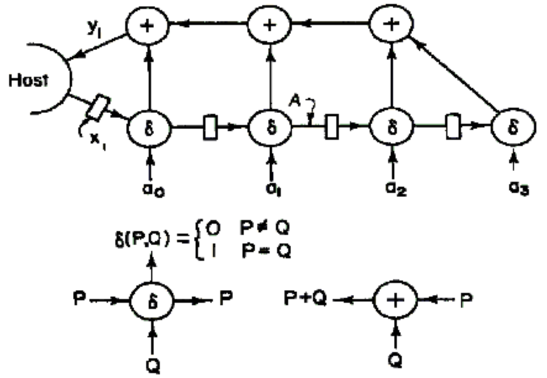
OPTIMAL RETIMING

* It is possible to compute the optimal positions of the delay elements in an efficient way.
* The optimization goal is to minimize the the longest path from any delay element to any other. In other words, to minimize the iteration period of a non-overlapping schedule.

☞ Leiserson, C.E., F.M. Rose and J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming (Preliminary Version)", In: R. Bryant (Ed.), Third Caltech Conference on VLSI, Springer Verlag, Berlin, pp. 87–116, (1983).

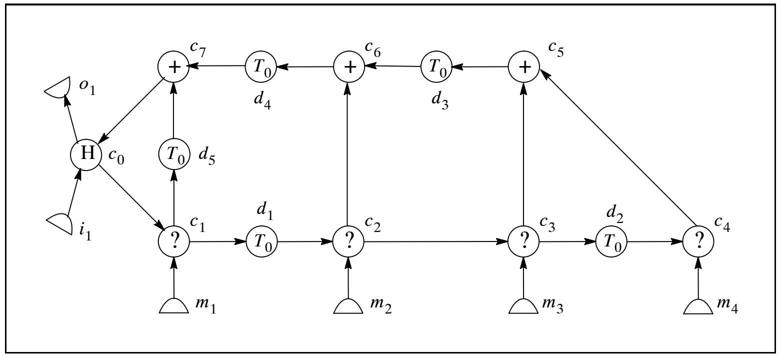
☞ Leiserson, C.E. and J.B. Saxe, Retiming Synchronous Circuitry, Algorithmica, Vol.6, pp. 5–35, (1991).

RETIMING: LEISERSON ET AL. CORRELATOR EXAMPLE



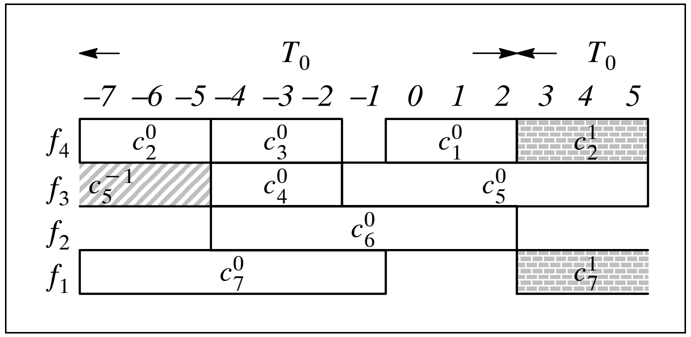
Given: $\delta(+) = 7$ *and* $\delta(?) = 3$; $T_{0_{\min}} = ?$

OPTIMAL RETIMING VS. FASTEST SCHEDULE (1)



$T_{0_{\min}} = 13$ *for non-overlapped schedule when* $\delta(+) = 7$ *and* $\delta(?) = 3$*; however,* $T_{0_{\min}} = 10$ *for an overlapped schedule.*

OPTIMAL RETIMING VS. FASTEST SCHEDULE (2)



Overlapped schedule with $T_{0_{\min}} = 10$.

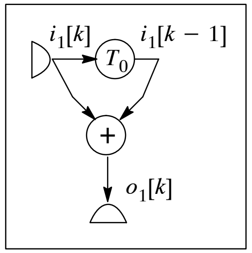
SOME REMARKS ON $T_{0_{\min}}$

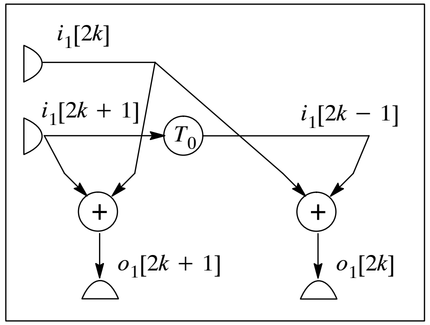
* Retiming does not affect $T_{0_{\min}}$ for overlapped scheduling of IDFG's.
* The T_0 for *nonoverlapped* scheduling obtained after optimal retiming may still be larger than $T_{0_{\min}}$. This is not true when all computational delays are equal to unity.
* $T_{0_{\min}}$ has been defined as an integer; a fractional $T_{0_{\min}}$ makes sense when *unfolding* is applied (unfolding creates a new DFG of multiple copies of the original one; see later).

☞ Chao, L.F. and E.H.M. Sha, "Rate-Optimal Static Scheduling for DSP Data-Flow Programs", 3rd Great Lakes Symposium on VLSI Design, Automation of High-Performance VLSI Systems, pp 80–84, (March 1993).

SPEED-UP TECHNIQUES: PARALLEL PROCESSING

* Works for acyclic IDFGs.
* Duplicate the IDFG as often as desired speed-up factor.
* Allows any arbitrary speed-up, but is proportionally expensive.





process 2 inputs at a time

LOOP UNROLLING

Loop unrolling is the process of explicitly describing multiple iterations of some loop in order to create more parallelism.

Original loop:

```
for (i=0; i<n; i++) {
  a[i] = f(x[i]);
  y[i] = g(a[i]);
}
```

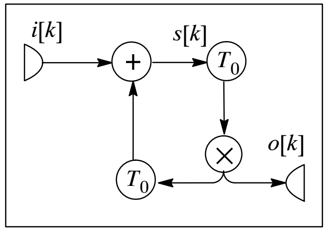
After unrolling with a factor 2:

```
for (i=0; i<n/2; i++) {
 a[2*i] = f(x[2*i]);
 y[2*i] = g(a[2*i]);
 a[2*i+1] = f(x[2*i+1]);
 y[2*i+1] = g(a[2*i+1])
}
```

For an IDFG, loop unrolling by a factor n amounts to converting it into an acyclic graph (by cutting the delay nodes) and concatenating n copies of the acyclic graph.

UNFOLDING (1)

* A technique for the duplication of cyclic IDFGs in combination with processing multiple inputs at a time. Cycles in the graph are preserved as opposed to loop unrolling.
* Consider the following IDFG:



* If $\delta(+) = 1$ and $\delta(*) = 2$, $T_{0_{\min}} = \left\lceil \frac{3}{2} \right\rceil = 2$.
* Using unfolding by 2, one can reach the value $T_{0_{\min}} = \frac{3}{2}$.
* The graph computes the following difference equations, assuming that one multiplies by a factor a:

$$s[k] = i[k] + o[k-1]$$
$$o[k] = as[k-1]$$

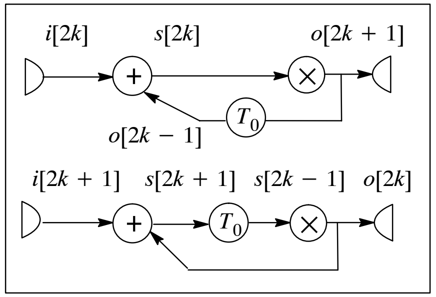
UNFOLDING (2)

* The precise unfolding algorithm will not be given here; it amounts to duplicating all vertices in the IDFG such that n copies of each vertex is created (n is the unfolding factor) and then to connecting these vertices with edges having an appropriate number of delay elements. The unfolded graph can also be reconstructed from the equations.
* The method will be illustrated using the example IDFG and unfolding factor of two, meaning that two inputs will be available per iteration and two outputs will be produced. The equations:

$$s[2k] = i[2k] + o[2k-1]$$
$$s[2k+1] = i[2k+1] + o[2k]$$
$$o[2k] = as[2k-1]$$
$$o[2k+1] = as[2k]$$

UNFOLDING (3)

* The example IDFG after unfolding:

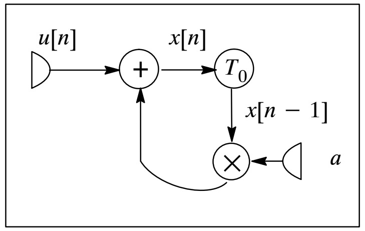


* Note that the unfolded IDFG has two loops with one delay element each and a computational duration of 3. Because a delay element creates an offset of two indices (2 inputs are processed in each iteration), the effective iteration period bound is equal to $T_{0_{\min}} = \frac{3}{2}$.

LOOK-AHEAD TRANSFORMATION (1)

* Consider the following computation:

$$x[n] = ax[n-1] + u[n]$$



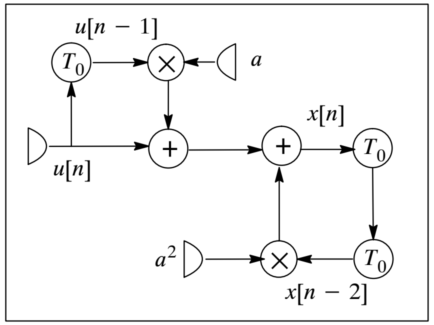
* It has one multiplication and one addition in the critical loop with one delay element. If $\delta(+) = 1$ and $\delta(*) = 2$, $T_{0_{\min}} = \left\lceil\frac{3}{1}\right\rceil = 3$.

LOOK-AHEAD TRANSFORMATION (2)

* Apply look-ahead transformation (think of the principle of look-ahead addition):

$$x[n] = a(ax[n-2] + u[n-1]) + u[n]$$
$$x[n] = a^2x[n-2] + au[n-1] + u[n]$$

* The new equation has one multiplication and one addition in the critical loop with two delays leading to $T_{0_{\min}} = \left\lceil\frac{3}{2}\right\rceil = 2$.
* The transformation can affect the original computation (finite word length effects).



RELATION WITH RTL SYNTHESIS

* Multicycle operations are not so common in RTL synthesis (one normally defines a clock period for the registers and all combinational logic should execute in this period).
* RTL synthesis programs such as the *Synopsys Design Compiler* do support multicycle operations, by the way.
* Presented theory becomes less interesting when all computations have a unit delay:
  + Non-overlapped scheduling after optimal retiming gives fastest implementation.
* Theory of transformations is still applicable to combinational logic in case of one-to-one mapping (think e.g. of converting the *ripple-carry adder* to the *look-ahead adder* by means of the *look-ahead* transformation).