ONE-TO-ONE MAPPING ON HARDWARE

One-to-one mapping: a direct mapping from data-flow graph (DFG) onto hardware.
* There is a separate functional unit (FU) in the hardware for each computational node in the DFG.
* Connections between FUs in the hardware are directly derived from the edges of the DFG.
* Specification style is close to RTL.
* The mapping basically results in a data path; controller is relatively simple.

FUNCTION MULTIPLEXING AND TIME FOLDING (1)

* The signal samples to be processed arrive every $T_0$ time units, the iteration period.
* Call the clock period of the hardware clock (assume that there is only one) $T_c$, $T_0 = nT_c$, $n$ being an integer.
* If $n = 1$, one-to-one mapping is the only option.
* If $n > 1$, there are opportunities to save hardware by reusing the same FUs for multiple computations in the DFG (one can e.g. perform 3 additions per iteration on a single adder when $n = 3$).
* $n$ is called the multiplex factor or reuse factor.

FUNCTION MULTIPLEXING AND TIME FOLDING (2)

* When $n > 1$, one can say that the DFG needs to be cut in parts in such a way that each part should be mapped on one and the same data path.
* Each part of the DFG can be considered a separate function, hence the name function multiplexing. Mapping multiple functions on the same hardware will in general require hardware multiplexers in the data path.
* The controller becomes more complex.
* Because one traverses the data path multiple times for one iteration, one could say that the “linear” time in the DFG folds back several time on the hardware, hence the name time folding.
* The issue is, of course: how to do it optimally?

OPTIMIZATION CRITERIA

Most commonly used:
* Time-constrained synthesis: given the iteration period $T_0^*$ use as few processors as possible or as little hardware as possible (typical for DSP).
* Resource-constrained synthesis: given a multiprocessor configuration or a set of hardware resources on chip, minimize $T_0^*$

Another important issue:
* Minimization of power.
TERMINOLOGY

Subtasks:
* Scheduling: determine for each operation the time at which it should be performed such that no precedence constraint is violated.
* Allocation: specify the hardware resources that will be necessary.
* Assignment: provide a mapping from each operation to a specific functional unit and from each variable to a register.

Remarks:
* The subproblems are strongly interrelated; they are, however, often solved separately.
* Scheduling (except for a few versions) is NP-complete heuristics have to be used.

SCHEDULING TERMINOLOGY

* Static scheduling means: mapping to time and processor (functional unit, register, etc.) is identical in all iterations.
* A static schedule is either overlapped (exploiting iteration parallelism) or nonoverlapped.


SOFTWARE PIPELINING EXAMPLE

without software pipeline:
```c
for (i=1; i<n; i++) {
    a[i] = f(x[i]);
    y[i] = g(a[i]);
}
```

with software pipeline:
```c
for (i=1; i<n; i++) {
    a[i] = f(x[i]);
    y[i-1] = g(a[i-1]);
}
```

* In the situation without software pipeline, there is a data dependency between the two statements; they cannot be parallelized.
* In the situation with software pipeline, the loop body can be parallelized at the expense of some initial and termination code. When the parallelization has been implemented, iterations of the original loop are overlapping in time.
THE MINIMAL ITERATION PERIOD $T_0$ ($T_{0\text{ min}}$)

There are four cases:
* Acyclic DFG, nonoverlapped schedule;
* Acyclic DFG, overlapped schedule;
* Cyclic DFG, nonoverlapped schedule;
  * Replace all delay nodes by pairs of input and output nodes.
* Cyclic DFG, overlapped schedule.

For the nonoverlapped cases:
* Compute the critical path, longest path from any input to any output, in the acyclic graph. $T_{0\text{ min}} = \text{length of critical path}$.

EXAMPLE

![Critical path in a cyclic DFG for a nonoverlapped schedule.]

OVERLAPPED SCHEDULING IN A CYCLIC DFG

* Minimal iteration period is given by critical loop.
  Example:

  \[ T_0 \geq \delta(c_1) + \delta(c_2) + \delta(c_3) \]

* When the DFG is acyclic, arbitrarily small iteration periods are possible (just duplicate the hardware as often as necessary; each copy can start any time as there are no feedback loops in the DFG; see later on).

CRITICAL LOOP

For a general DFG, $T_{0\text{ min}}$ is given by:

\[ T_{0\text{ min}} = \max_{\text{all loops } l} \left[ \frac{\sum_{c \in l} \delta(c)}{\sum_{d \in l} \mu(d)} \right] \]


EXAMPLE

\[ T_{0 \min} = 3, \text{ when } \delta(+) = 1 \text{ and } \delta(*) = 2. \]

ON COMPUTING \( T_{0 \min} \)

Remarks:
- Direct use of expression for \( T_{0 \min} \) not efficient (number of loops in graph may grow exponentially with respect to number of nodes).
- Many polynomial-time algorithms have been published; survey in:
- An easy to understand but not very efficient method is based on "matrix multiplication".

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.


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

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

**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.

![Diagram of IDFG](image)

**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:

![Diagram of IDFG](image)

If $\delta(+) = 1$ and $\delta(\times) = 2$, $T_{0_{\text{min}}} = \frac{3}{2} = 2$.

* Using unfolding by 2, one can reach the value $T_{0_{\text{min}}} = \frac{3}{2}$.

* The graph computes the following difference equations, assuming that one multiplies by a factor $a$:

\[
\begin{align*}
s[k] &= i[k] + o[k - 1] \\
o[k] &= as[k - 1]
\end{align*}
\]

**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.

\[
\begin{align*}
s[2k] &= i[2k] + o[2k - 1] \\
s[2k + 1] &= i[2k + 1] + o[2k] \\
o[2k] &= as[2k - 1] \\
o[2k + 1] &= as[2k]
\end{align*}
\]

**LOOP UNROLLING**

* 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.

**Original loop:**

\[
\begin{align*}
& \text{for } (i=0; i<n; i++) \\
& \quad a[i] = f(x[i]); \\
& \quad y[i] = g(a[i]); \\
& \end{align*}
\]

**After unrolling with a factor 2:**

\[
\begin{align*}
& \text{for } (i=0; i<n/2; i++) \\
& \quad a[2*i] = f(x[2*i]); \\
& \quad y[2*i] = g(a[2*i]); \\
& \quad a[2*i+1] = f(x[2*i+1]); \\
& \quad y[2*i+1] = g(a[2*i+1]) \\
& \end{align*}
\]

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 (2)**

* 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:

\[
\begin{align*}
s[2k] &= i[2k] + o[2k - 1] \\
s[2k + 1] &= i[2k + 1] + o[2k] \\
o[2k] &= as[2k - 1] \\
o[2k + 1] &= as[2k]
\end{align*}
\]
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 $T_{0_{\text{min}}} = \frac{3}{2}$.

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 $\delta(\cdot) = 2$, $T_{0_{\text{min}}} = \frac{3}{1} = 3$.

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 $T_{0_{\text{min}}} = \frac{3}{2} = 2$.

* 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).