IMPLEMENTATION OF DIGITAL SIGNAL PROCESSING (IDSP)

Architecture Synthesis and (Overlapped) Scheduling

© Sabih H. Gerez, University of Twente, The Netherlands

TOPICS

* Hardware models
* Terminology
* Minimal iteration period
* Mobility-based scheduling
* Assignment

HIGH-LEVEL SYNTHESIS (HLS)

VHDL synthesis:
* Starts from a register-transfer level (RTL) description; circuit behavior in each clock cycle is fixed.
* Uses logic synthesis techniques to optimize the design.
* Generates a standard-cell netlist.

High-level synthesis (also called architectural synthesis):
* Starts from an abstract behavioral description.
* Generates an RTL description.

STATUS

• Sign-up for teams of 2:
  – Matchmaking in the break?

• Pre-knowledge reparation: who has started?
  – Connection issues?
  – Other questions?

•Forgot to mention last week:
  – A final oral session is part of the exam. We will go through your reports in this session and you should be prepared to answer questions on your work. The grading of your report can then be corrected by plus or minus one point.
HARDWARE MODELS FOR HIGH-LEVEL SYNTHESIS

All HLS systems need to restrict the target hardware. The search space is too large, otherwise.

All synthesis systems have their own peculiarities; but most systems generate synchronous hardware and build it with the following parts:

* **functional units**: they can perform one or more computations, e.g. addition, multiplication, comparison, ALU.

* **registers**: they store inputs, intermediate results and outputs; sometimes several registers are taken together to form a register file.

* **multiplexers**: from several inputs, one is passed to the output.

* **busses**: a connection shared between several hardware elements, such that only one element can write data at a specific time.

* **three-state (tri-state) drivers** control the exclusive writing on the bus.

Parameters defining the hardware model for the synthesis problem:

* **clocking strategy**: e.g. single or multiple phase clocks.

* **interconnect**: e.g. allowing or disallowing buses.

* **clocking of functional units**: allowing or disallowing of:
  + multicycle operations;
  + chaining;
  + pipelined units.
EXAMPLE OF A HLS HARDWARE MODEL

HARDWARE CONCEPTS: DATA PATH + CONTROL

Hardware is normally partitioned into two parts:
* the data path: a network of functional units, registers, multiplexers and buses. The actual “computation” takes place in the data path.
* control: the part of the hardware that takes care of having the data present at the right place at a specific time, of presenting the right instructions to a programmable unit, etc.

Often high-level synthesis concentrates on data-path synthesis. The control part is then realized as a finite state machine or in microcode.

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?

FOLDING AN FIR FILTER

How would this DFG fold on hardware with a single *multiply-add-accumulate* (MAC)?

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 in high-level synthesis:
* *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.
* Traditionally, scheduling precedes assignment. However, starting with assignment may give better results (more regular data path).
* Scheduling (except for a few versions) is NP-complete $\Rightarrow$ 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 interiteration parallelism) or nonoverlapped.


SOFTWARE PIPELINING EXAMPLE

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

with software pipeline: 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.

EXAMPLE DATA-FLOW GRAPH

The second-order filter section made acyclic (delay elements replaced by output-input pairs):
EXAMPLE SCHEDULE

The schedule and operation assignment with an allocation of one adder and one multiplier:

\[
\begin{array}{cccc}
\times & c_3 & c_4 & c_5 & c_6 \\
+ & c_1 & c_2 & c_7 & c_8 \\
\end{array}
\]

EXAMPLE DATA PATH

The resulting data path after register assignment (the specification of a controller completes the design):

IDFG NOTATION

IDFG(V,E) with:

- V: the vertex set
  \[ V = C \cup D \cup I \cup O \]
- C: set of computational nodes
- D: set of delay nodes
- I: set of input nodes
- O: set of output nodes
- E: the edge set
- \( \delta(c), c \in C \) gives the duration of a computation (atomic, non-preemptive, restricted library)
- \( \mu(d), d \in D \) gives the multiplicity of a delay node

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. \( T_{0_{\text{min}}} \) = “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, \( T_{0_{\min}} \) is given by:

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


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".

PROBLEM FORMULATION (1)
* Input consists of:
  + a DFG $G(V, E)$; $C$ is the set of computational nodes ($C \subseteq V$); $D$ is the set of delay nodes ($D \subseteq V$).
  + a library $F$ of functional units (FUs) or resource types.

* $\Omega$ is the set of all operations; the operation type of a computational node is given by $\gamma : C \rightarrow \Omega$.
* The delay for the fastest execution of an operation is given by $\delta : C \rightarrow N$.
* The multiplicity of a delay element is given by $\mu : D \rightarrow N$.
* An FU can execute one or more operations given by $\Gamma : F \rightarrow 2^\Omega$ ($2^\Omega$ is the power set of $\Omega$, the set of all its subsets).

FORWARD PRECEDENCE RELATIONS
* Inequalities for starting time derived from edges connecting two computational nodes.

\[\forall c_i, c_j \in C, (c_i, c_j) \in E : \sigma(c_j) \geq \sigma(c_i) + \delta(c_i)\]

Example:

$\sigma(c_7) \geq \sigma(c_2) + \delta(c_2)$
$\sigma(c_9) \geq \sigma(c_8) + \delta(c_8)$
$\sigma(c_8) \geq \sigma(c_{10}) + \delta(c_{10})$

BACKWARD PRECEDENCE

For all paths \( R \) connecting two nodes \( c_i, c_j \) via delay nodes:

\[
\sigma(c_j) \geq \sigma(c_i) + \delta(c_i) - T_0 \sum_{d \in R} \mu(d)
\]

Example:

\[
\begin{align*}
\begin{array}{c}
\circ \ 1 \ c_2 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_1 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_9 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_8 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_7 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_6 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_5 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_4 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_3 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_2 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_1 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_9 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_8 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_7 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_6 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_5 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_4 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_3 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_2 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_1 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_9 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_8 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_7 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_6 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_5 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_4 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_3 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_2 \\
\downarrow 0
\end{array} & \quad \begin{array}{c}
\circ \ 1 \ c_1 \\
\downarrow 0
\end{array} \end{array}
\end{align*}
\]

\[
\begin{align*}
\sigma(c_9) & \geq \sigma(c_2) + \delta(c_2) - T_0 \\
\sigma(c_{10}) & \geq \sigma(c_2) + \delta(c_2) - 2T_0
\end{align*}
\]

THE INEQUALITY GRAPH

* Given \( T_0 \), all inequalities can be written as:

\[
\sigma(c_a) - \sigma(c_b) \geq \kappa_{ab}
\]

where \( \kappa_{ab} \) is a constant.

* For all input nodes, \( i \in I \), set \( \sigma(i) = 0 \).

* For all output nodes, \( o \in O \), set \( \sigma(o) = \lambda \), where \( \lambda \) is the latency.

* Create an inequality graph with a vertex for each computational, input and output node in the DFG.

* All inequalities can be represented in the graph with edges having weight \( \kappa_{ab} \).

SOLVING THE INEQUALITIES

* **Lower bounds** for the node scheduling times \( \tilde{\sigma}_-(v) \) are found by solving the longest-path problem, with the input nodes as the sources.

* **Upper bounds** \( \tilde{\sigma}_+(v) \) are found analogously, by computing the longest paths traversing the graph in the reverse direction starting from the output nodes.

* Scheduling all operations on their lower bounds, gives an ASAP (as soon as possible) schedule and the upper bounds lead to an ALAP (as late as possible) schedule.

* A partial schedule \( \tilde{\sigma} : V \rightarrow [\mathbb{Z}, \mathbb{Z}] \) assigns a scheduling range or time frame to each \( v \in V \):

\[
\tilde{\sigma}(v) = [\tilde{\sigma}_-(v), \tilde{\sigma}_+(v)].
\]

* The initial scheduling range is given by ASAP/ALAP schedule.
MOBILITY-BASED SCHEDULING

* The difference $\bar{\sigma}_+(v) - \bar{\sigma}_-(v)$ is called $v$'s mobility (or freedom).
* It can be used as the basis of many scheduling heuristics (most scheduling problems are NP-complete).
* Finding a schedule can be seen as the generation of a sequence of partial schedules until all mobility has disappeared.

Principles of a general mobility-based heuristic:
* determine scheduling ranges of all operations;
* decide to shorten scheduling range of at least one operation;
* update ranges of all affected operations;
* repeat until all operations have mobility zero.

EXAMPLE HEURISTIC

Hybrid genetic algorithm:
* Design a simple heuristic that can be parametrized by a permutation of all operations and that quickly generates a schedule.
* Use a genetic algorithm to find the permutation that leads to the best schedule.

Principles of a general mobility-based heuristic:
* determine scheduling ranges of all operations;
* decide to shorten scheduling range of at least one operation;
* update ranges of all affected operations;
* repeat until all operations have mobility zero.

UPDATING THE RANGES

The best known method requires $O(|C|)$ time per update.
* Precompute all-pairs longest paths in inequality graph using e.g. the Floyd-Warshall or Johnson algorithm and store the path lengths in a distance matrix $D_{T_0}$.
* Suppose that operation $v$ has been scheduled at time $\sigma(v)$; then for all $u \in C$:

$$\bar{\sigma}_-(u) \leftarrow \max(\bar{\sigma}_-(u), \sigma(v) + D_{T_0}[v, u])$$

$$\bar{\sigma}_+(u) \leftarrow \min(\bar{\sigma}_+(u), \sigma(v) - D_{T_0}[u, v])$$

LIST SCHEDULING

* A resource-constrained scheduling method.
* Start at time zero and increase time until all operations have been scheduled.
* The ready list $L_t$ contains all operations that can start their execution at time $t$ or later.
* If more operations are ready than there are resources available, use some priority function to choose, e.g. the longest-path to the output node $\Rightarrow$ critical-path list scheduling.
LIST SCHEDULING EXAMPLE

Subtasks in assignment:
* operation-to-FU assignment
* value grouping
* value-to-register assignment
* transfer-to-wire assignment
* wire to FU-port assignment

In general: task-to-agent assignment

COMPATIBILITY AND CONFLICT GRAPHS

Clique partitioning gives an assignment in a compatibility graph.

Graph coloring gives an assignment in the complementary conflict graph.

ASSIGNMENT

* After scheduling is completed, all operations \( \nu \) are known to occupy a time interval \([\sigma(\nu), \sigma(\nu) + \delta(\nu)]\) of a processor.
* The goal of assignment is to use as few processors as possible.
* All times should be taken modulo \( T_0 \) because of the iterative computation.
* If none of the intervals cross the \( T_0 \) boundary, the problem is equivalent to interval-graph coloring and can be solved optimally in polynomial time using the left-edge algorithm.

NONOVERLAPPED ASSIGNMENT EXAMPLE

Consider the intervals $i_1 = [1, 4], i_2 = [12, 15], i_3 = [7, 13], i_4 = [3, 8], i_5 = [5, 10], i_6 = [2, 6]$ and $i_7 = [9, 14]$, with $T_0 = 16$.

\[(a) \quad (b)\]

Corresponding interval graph (a) and its optimal coloring (b).

OVERLAPPED ASSIGNMENT

* Intervals cross $T_0$ boundary and the problem becomes a circular-arc graph coloring problem which is NP-complete.
* Even in nonoverlapped scheduling register lifetimes cross the $T_0$ boundary!
* Powerful exact (Stok and Jess, 1992) and heuristic (Hendren et al., 1993) solutions have been reported.

CIRCULAR-ARC GRAPH COLORING EXAMPLE

\[(a) \quad (b) \quad (c)\]

A set of circular arcs (a), its corresponding graph (b) and its optimal coloring (c).