HIGH-LEVEL SYNTHESIS

*High-level synthesis:* the automatic addition of structural information to a design described by an algorithm.

*High-level synthesis visualized on Gajski's Y-chart*
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 \textit{synchronous} hardware and build it with the following parts:

* \textit{functional units}: they can perform one or more computations, e.g. addition, multiplication, comparison, ALU.
HARDWARE MODELS (Continued)

* **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.

\[
o = i_k , \quad k = 2c_1 + c_0
\]
HARDWARE MODELS (Continued)

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

![Diagram of busses]

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

![Diagram of three-state driver]

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

Parameters defining the hardware model for the synthesis problem:

* clocking strategy: e.g. one- or two-phase clocks.

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

* clocking of functional units: allowing or disallowing of:
  + multicycle operations;
  + chaining;
  + pipelined units.
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.
INPUT FORMAT

The algorithm, that is the input for a high-level synthesis system, is often provided in textual form either:

* in a *conventional programming language*, such as Pascal, or
* in a *hardware description language*, which is more suitable to express the parallelism present in hardware.

The description has to be parsed and transformed into an *internal representation*; here conventional compiler techniques can be used.
INTERNAL REPRESENTATION

Most systems use some form of a *data-flow graph*. A DFG may or may not contain information on control flow.

A data-flow graph is built from:
* **vertices (nodes):** representing computation, and
* **edges:** representing *precedence* relations.
DATA FLOW

\[
x := a \times b; \quad y := c + d; \\
z := x + y;
\]
TOKEN FLOW IN A DFG

* A node in a DFG *fires* when *tokens* are present at its inputs.

* The input tokens are *consumed* and an output token is *produced.*
CONDITIONAL DATA FLOW

By means of two special nodes:

* **selector**

![Selector Diagram]

* **distributor**

![Distributor Diagram]
Selector and distributor nodes can be used to describe iteration.

While $a > b$
do

\[ a := a - b; \]
Loops require careful placement of initial tokens on edges.
**IMPLICIT ITERATIVE DATA FLOW**

* Iteration implied by stream of input tokens arriving at regular points in time.
* Initial tokens act as buffers.

* Delay elements instead of initial tokens.
ITERATIVE DFG EXAMPLE

A second-order filter section.
HIGH-LEVEL TRANSFORMATIONS

Restructuring data and control flow graphs prior to the actual mapping onto hardware.

Examples:
* Replacing “division by 2” by “shift”.
* Loop unrolling.
* Replacing chain of adders by a tree.
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.

  For an edge \( (v_i, v_j) \): \( t_j \geq t_i + \delta_i \)

* **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 \( \Rightarrow \) heuristics have to be used.
EXAMPLE

The second-order filter section made acyclic:

The schedule and operation assignment with an allocation of one adder and one multiplier:
EXAMPLE (Continued)

The resulting data path after register assignment:
OPTIMIZATION CRITERIA

Not surprisingly, the objective function to be optimized is similar to the one for the design of VLSI circuits in general. It is a combination of:

* speed: how long does it take to perform the intended computation,
* area: the addition of the areas of all functional units and registers, sometimes taking the area of interconnection into account.

Often optimization is constrained:

* Optimize area when the minimum speed is given ⇒ time-constrained synthesis.
* Optimize speed when maximum area is given ⇒ area-constrained synthesis.
* Optimize speed when a maximum for each resource type is given ⇒ resource-constrained synthesis.
SCHEDULING METHODS

* As soon as possible (ASAP):
  + an operation can be scheduled when all its predecessors have been scheduled.
  + very simple, as data-flow graph need only to be traversed from inputs(s) to output(s).

* As late as possible (ALAP):
  + similar to ASAP, but scheduling is now performed from output(s) to input(s).

* List scheduling:
  + similar to ASAP; however, now an extra criterion is used for choosing between all operations that have all their predecessors scheduled.
SCHEDULING METHODS
(Continued)

* Critical-path list scheduling:
  + sorting criterion is the length of the longest path from operation to output.
  + gives good results in practice!

* Freedom-based scheduling:
  + compute both ASAP and ALAP schedules (longest-path computations from inputs to operation and from operation to outputs).
  + the difference in scheduling position gives the freedom or mobility of an operation (operations in the critical path have mobility zero).
  + take advantage of mobility to find a good position within scheduling range.
FORCE-DIRECTED SCHEDULING

* extension of freedom-based methods.
* computes so-called distribution graph to model the global effects on hardware resources.

\[
\begin{array}{c}
  a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_3 \\
  \times \quad \times \\
  a_2 \rightarrow a_3 \\
  a_1 \quad a_2 \quad a_3 \\
\end{array}
\]

* a partial schedule \( \tau \), is a schedule where some transfers have a fixed time positions and others have a scheduling range.
* the resource utilization for resource \( r \) of a partial schedule \( \tau \) at time \( t \) is given by the distribution function \( \hat{\theta}_r(\tau, t) \).
FORCE-DIRECTED SCHEDULING (Ctd.)

* When fixing an operation’s time, one makes a transition from partial schedule $\hat{\tau}$ to partial schedule $\hat{\sigma}$.

* For every operation a force $F_r(\hat{\tau}, \hat{\sigma})$, is computed for every possible position within an operation’s range.

* Basic force function:

$$F_r(\hat{\tau}, \hat{\sigma}) = \sum_{t \in \text{range}(op)} \hat{\theta}_r(\hat{\tau}, t)(\hat{\theta}_r(\hat{\sigma}, t) - \hat{\theta}_r(\hat{\tau}, t))$$

* The lowest force determines which operation is scheduled and at which position.
THE ASSIGNMENT PROBLEM

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

Assumption: assignment follows scheduling.
* Then the claim of a task on an agent is an interval \( \Rightarrow \) minimum resource utilization can be found by left-edge algorithm.
* In case of iterative algorithm, interval graph becomes circular-arc graph \( \Rightarrow \) optimization is NP-complete.
ASSIGNMENT BY CLIQUE PARTITIONING

* Tasks are *compatible* when they can be executed on the same agent simultaneously ⇒ *compatibility graph.*

* A *clique* is a maximal complete subgraph.

* *Clique partitioning* divides the vertices of a graph into complete subgraphs.

A graph and its cliques
TSENG AND SIEWOREK’S CLIQUE-PARTITIONING ALGORITHM

\[ k := 0; \]
\[ G_c^k(V_c^k, E_c^k) := G_c(V_c, E_c); \]
\[ \text{while } E_c^k \neq \emptyset \text{ do} \]
\[ \text{begin} \]
\[ \quad \text{“find } v_i, v_j \text{ with largest set of common neighbors”;} \]
\[ \quad N := \text{“set of common neighbors of } v_i \text{ and } v_j”; \]
\[ \quad s := i \cup j; \]
\[ \quad V_c^{k+1} := V_c^k \cup \{v_s\} \setminus \{v_i, v_j\}; \]
\[ \quad E_c^{k+1} := \emptyset; \]
\[ \quad \text{for each } (v_m, v_n) \in E_c^k \text{ do} \]
\[ \quad \quad \text{if } v_m \neq v_i \land v_m \neq v_j \land v_n \neq v_i \land v_n \neq v_j \]
\[ \quad \quad \text{then } E_c^{k+1} := E_c^{k+1} \cup \{(v_m, v_n)\}; \]
\[ \quad \text{for each } v_n \in N \text{ do} \]
\[ \quad \quad E_c^{k+1} := E_c^{k+1} \cup \{(v_n, v_s)\}; \]
\[ \quad k := k + 1 \]
\[ \text{end} \]
CLIQUE-PARTITIONING EXAMPLE

(a) $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$
(b) $v_2, v_3, v_4, v_5, v_6, v_7$
(c) $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$
(d) $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$
(e) $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$
(f) $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$
HIGH-LEVEL TRANSFORMATIONS

Restructuring data and control flow graphs prior to the actual mapping onto hardware.

Examples:
* Replacing “division by 2” by “shift”.
* Loop unrolling.
* Replacing chain of adders by a tree.