HIGH-LEVEL SYNTHESIS OF DIGITAL CIRCUITS

Giovanni De Micheli

Technical Report No. CSL-TR-92-551

November 1992

This research was sponsored by NSF and DEC under a PYI award.

HIGH LEVEL SYNTHESIS OF DIGITAL CIRCUITS

Giovanni De Micheli

Technical Report: CSL-TR-92-551

November 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

This tutorial surveys the state of the art in high-level synthesis of digital circuits. It will be published in *Advances in Computers*, Volume 35, edited by M. Yovits.

Keywords and Phrases: high-level synthesis, scheduling, binding, datath and control synthesis.

Contents

1 Introduction.

As digital hardware systems become larger and more complex, engineers need even more powerful design tools. Synthesis systems have been shown effective in providing an automated way, or a computer-assisted environment, for integrated circuit design.

Several advantages stem from using synthesis tools, starting from behavioral circuit models. First, high-level modeling languages allow designers to conceptualize circuits in a self-documenting form, that can be fairly independent of the target technology and design style. This provides design portability and support for incremental changes in later design revisions. At the same time, it makes circuit design available to a larger base of engineers, who master system design issues without being expert in specific circuit technologies.

The second advantage provided by synthesis tools is correctness and optimality. While design correctness is relying in part on the software implementation of algorithms, and therefore hard to claim, it is obvious that the reliability in handling large scale designs is higher when performed by automated means than when done by human beings. Design optimization at various levels is coupled to synthesis. High-level optimization is extremely critical in achieving the best circuit implementation, because it affects the macroscopic circuit parameters.

Eventually, synthesis tools provide a shortening of design time, reducing both the design cost and the time to market. Both factors are crucial in the competitive marketplace of integrated circuits.

Computer-aided synthesis of digital circuit has been introduced gradually over the last two decades. At first physical design tools, and later logic synthesis and optimization programs, became common in the design flow of integrated circuits. Recently, high-level synthesis techniques have been proposed and used for research as well as for some product-level design. While these techniques are not yet used ubiquitously, it is likely that they will have positive impact on digital design methodology.

High-level synthesis is a broad term to define circuit synthesis from models that are more abstract and general than logic ones. The circuit modeling problem is strongly related to high-level synthesis, because it defines the boundary of this domain. High-level modeling is done by means of *Hardware Description Languages* (HDLs), as mentioned in Section 2.1. The lack of standardization of HDLs suitable for synthesis has been a major impediment in the diffusion of high-level synthesis. We shall consider in Section 2.2 modeling issues with particular reference to abstract circuit models, based on graphs, that can serve as common basis for synthesis and that decouple synthesis and optimization from the particular features of any given language.

High-level optimization is coupled to synthesis. It is customary to gather optimization techniques into two classes. The former groups those optimizations that are independent on the circuit structure and that parallel the techniques used in software optimizing compilers. Such techniques are described in Section 3.

The latter class consists of the algorithms for creating and optimizing the *data-path* and the *control-unit*. Data-path synthesis consists of binding the operations of the data-flow model to time slots and to computational units. The time-binding is often called *scheduling* of the operations. The physical binding is often called *resource binding* and it may involve *resource sharing*. Control synthesis corresponds to interpreting the control-flow of the model and in constructing a control unit that activates the resources at the appropriate time and with the appropriate inputs and destinations. Scheduling, resource binding and control synthesis are described in Sections 5, 6 and 7 respectively. Structural synthesis and the related tasks are described first as applied to non-pipelined circuits, for the sake of simplicity. Extensions to pipelined models are reported in Section 8.

We conclude this chapter by giving a short history of high-level synthesis and by describing and comparing high-level synthesis systems in Section 9. This review will give the reader an idea about the state of the art in the field, the success achieved by high-level synthesis and the present difficulties.

2 Circuit modeling.

Circuit modeling plays a fundamental role in defining the synthesis task as well as in capturing essential features of a design. We consider in the sequel circuit models at both the *functional* and *logic* abstraction levels with *behavioral* and *structural* flavors. At the former level, a circuit behavior can be captured by a set of *tasks* and a *partial order* on the set of tasks. The tasks may be general in nature, involving arithmetic or logic functions. Similarly, a circuit structure can be described by a partition into functional units and their interconnection. At the logic level, a circuit behavior can be modeled in terms of *states* and *transitions*. A circuit structure can be stated in terms of *logic gates* and their interconnection. Circuit behavior at the functional level, called shortly circuit behavior, is the staring point for high-level synthesis.

Hierarchical models are often used to simplify the representation. The hierarchy can be used to render a model *modular*, by encapsulating some of its portions, as well as making possible the multiple usage of a (sub)-model by means of a *model* call. The use of hierarchical models has been used extensively in software (e.g. subroutines).

2.1 Modeling languages.

Hardware Description Languages are often used to represent the circuit behavior or its structure. Modern HDLs, such as VHDL [37], Verilog [53] and UDL/I [25], support both a behavioral and a structural modeling style. We will consider the former only, because high-level synthesis is not required for structural models.

A natural question to ask is why standard programming languages, like C, cannot be used to model hardware behavior. They can indeed, but in a restricted domain. Functional models of processors can be defined as C programs, compiled and executed for validating the correctness. It is questionable though how useful they can be for synthesis.

There are several differences between standard programming languages and modern HDLs. The major ones stem from the fact that hardware circuits can always provide parallel streams of execution and that the precise timing of the execution of the operations may be very important in hardware. HDLs have explicit definition of I/O ports and provide some specification means for determining when some operations are executed and by which unit. On the other hand, HDLs do not support complex memory reference mechanisms like pointers and structures.

Whereas the syntax of HDLs varies widely, most of them are procedural, with an imperative semantics. Therefore the designer models a set of tasks by sequencing assignments to variables by means of control-flow constructs, such as *branching*, *iteration* and *model call*.

Digital circuits perform Boolean operations and their semantics is the same in hardware and in software. Similarly, the semantics of the HDL control-flow constructs parallels that of common programming languages. The interesting differences between HDL and programming language semantics are related to the data types and to the timing of the operations.

The fundamental data-type in HDLs is the Boolean variable. When considering languages with imperative semantics, variables may store information. They may be assigned multiple values, and they retain the last value until the next assignment occurs. Therefore a hardware mechanism has to be associated with the storage, unless the information of a variable is readily consumed. In this case, variables correspond to wires in the circuit. Otherwise, they relate to more complex structures, such as registers (with possibly multiplexed inputs and enables). This side-effect, i.e. modeling implicitly storage in imperative HDLs, is not a desirable feature, and complicates synthesis.

Other variables, often called *meta-variables*, can be used to simplify the representation. For example, meta-variables can be integers that address elements of a vector. Iteration on meta-variables is permitted. Note that meta-variables do not have a hardware counterpart, and they are expanded in an early synthesis stage.

The timing semantics of HDLs is currently subject of wide discussion. Some HDLs, like VHDL and Verilog, were conceived for circuit specification and simulation. Therefore their constructs are geared toward the efficient support of event-driven simulation. Some constructs do not even have a hardware correspondence. Models in VHDL and Verilog specify circuit behavior as a set of sequential statements. Since the languages do not specify a timing semantics, *synthesis policies* are used to interpret the timing behavior. This has the unfortunate side-effect of linking the semantics of hardware models to a policy and hence to a synthesis tool, at the expense of its generality. Conversely, the UDL/I language as a formal hardware semantics. The timing of the operations in a UDL/I model is linked to the states of an automaton, hence prescribing one execution interval for each operation. Preciseness is achieved at the expense of sacrificing some degrees of freedom in interpreting the model and in lowering the overall abstraction level.

2.2 Abstract models.

Abstract models capture the essential features of behavioral models, and decouple them from the language. Behavioral models at the functional level of abstraction are specified in terms of *tasks* and their *dependencies*. A task is often called an *operation*. For the sake of generality we assume that tasks can be also No-Operations (NOPs), i.e. fake operations that execute instantaneously with no side effect and that can be used as placeholders. Dependencies arise from several reasons. First, availability of data. When an input to an operation is the result of another one, then the former operation depends on the latter. Second, serialization constraints. A task may have to follow a second one regardless of data-dependency. A simple example is provided by the two following operations: loading data on a bus and raising a flag. The circuit model may require that the flag is raised after the data is loaded. Third, dependencies may arise because two tasks share the same resource, that can service only one task at a time. Thus one task has to perform before the other. Note though that in general dependencies due to resource sharing are not part of the original circuit specification, because the way in which resources are exploited is related to the circuit implementation.

Many different models have been proposed to model the circuit behavior in terms of graphs. We consider in this chapter only one model, called *sequencing graph*, that is a hierarchical combination of *data-flow* graphs. The data-flow graph entities model the data-flow while the hierarchical linkage of the entities models the control flow.

Let us consider first a flat sequencing graph, i.e. a data-flow model. It represents operations and data dependencies. It is a directed graph $G(V, E)$ whose vertex set $V = \{v_i; i = 1, 2, \ldots, n_{ops}\}$ is in one to one correspondence with the set of tasks. The directed edge set $E = \{(v_i, v_j); i, j = 1, 2, \ldots, n_{ops}\}$ is in correspondence with the transfer of data from an operation to another one. Data-flow graphs are acyclic and can be made *polar* by adding two vertices, called *source* and *sink*, that represent first and last tasks. They correspond to No-Operations and are labeled by v_0 and v_N respectively. Therefore the graph has $n_{ops} + 2$ vertices and subscript N is interchangeable with $n_{ops} + 1$. Appropriate edges are added to link the source and sink to the other vertices. We say that vertex v_i is a predecessor (or immediate predecessor) of v_j when there is a path (or an edge) with tail v_i and head v_j. Similarly, we say that vertex v_i is a successor (or immediate successor) of v_j when there is a path (or an edge) with

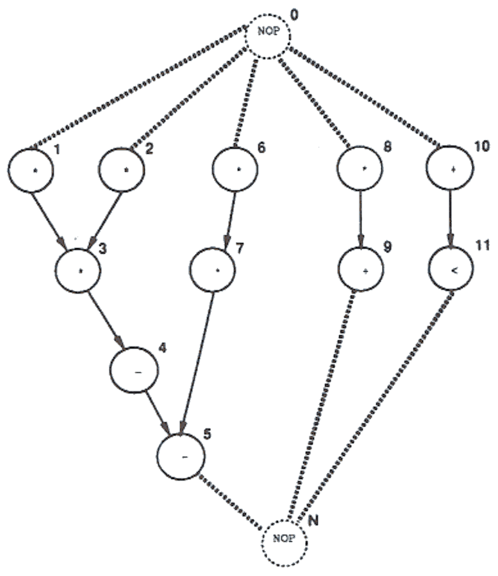

Figure 1: Example of sequencing graph.

tail v_j and head v_i. Note that paths in the graph represent *concurrent* (and not alternative) streams of operations.

Example 2.1. Consider the following program fragment, describing a set of computations.

$$\begin{aligned} xl &= x + dx \\ ul &= u - (3 \cdot x \cdot u \cdot dx) - (3 \cdot y \cdot dx) \\ yl &= y + u \cdot dx \\ c &= x < a \end{aligned}$$

The program describes a set of tasks, corresponding to simple operations like addition, subtraction, multiplication and comparison. The sequencing graph representation of these tasks is shown in Figure 1. The first statement corresponds to vertex v_{10} and the last to v_{11}. The third statement corresponds to vertices v_8 and v_9. The remaining vertices model the second statement. Note that alternative representations would be possible, by exploiting the commutativity and associativity of addition and multiplication. □

Let us consider now hierarchical sequencing graphs. A sequencing graph entity has two kinds of vertices. Some vertices model operations and are called *simple* vertices. Other vertices represent links to other sequencing graphs entities in the hierarchy and are called *complex* vertices. Obviously, sequencing graph entities that are leaves of the hierarchy have only simple vertices. Complex vertices represent *model call, branching* and *iteration* constructs.

A model call vertex is a pointer to another sequencing graph entity, at a lower level in the hierarchy. It models a set of dependencies from its immediate predecessors to the source vertex of the called entity and another set of dependencies from the corresponding sink to its immediate successors.

Branching constructs can be modeled by a *branching clause* and *branching bodies.* A branching body is an alternative partial order of tasks, that is selected according to the value of the branching clause. There are as many branching bodies as the possible values of the branching clause. Branching is modeled by associating a sequencing graph entity to each branch body and a complex vertex to the branching clause. The selection of a branch body is then modeled as a selective model call to the corresponding sequencing graph.

Iterative constructs are modeled by an *iteration clause* and an *iteration body*. An iteration (or loop) body is a partial order of tasks, that is repeated as long as the iterative clause is asserted. Iteration is modeled in sequencing graphs through the use of the hierarchy, thus preserving the acyclic nature of the graph. Iteration is represented as a repeated model call to the sequencing graph entity modeling the iteration body.

Example 2.2. We consider now an example of a sequencing graph, that has an iterative construct. The example has been adapted from one proposed by Paulin *et alii* [45].

```
diffeq
{
        read ($x, y, u, dx, a$
        repeat {
                $xl = x + dx$;
                $ul = u - (3 \cdot x \cdot u \cdot dx) - (3 \cdot y \cdot dx)$;
                $yl = y + u \cdot dx$;
                $c = x < a$;
                $x = xl; u = ul; y = yl$;

        until ( c )
    write ($y$);
}
```

The corresponding sequencing graph is shown in Figure 2. The loop body indicated in the figure is the sequencing graph entity shown in Figure 1. Note that the assignments $x = xl; u = ul; y = yl$; are not explicitly represented in the graph. Note also that the assignment $c = x < a$ could be moved to the top graph entity in the hierarchy. These particular choices in representing the loop of the HDL model by the sequencing graph of Figures 2 and 1 are motivated by the desire of keeping this example similar to what has been presented in the literature. □

The semantic interpretation of the sequencing graph model requires the notion of *marking* the vertices. A marking denotes the state of the corresponding operation, which can be: i) waiting for execution; ii) executing; iii) having completed execution. *Firing* an operation means starting its execution. Then, the semantics of the model is as follows: *an operation can be fired as soon as all its immediate predecessors have completed execution.*

We assume that a model can be reset, by marking all the operations as waiting for execution. Then, the model can be fired (i.e. executed) by firing the source vertex. The model has completed execution when the sink has completed execution. A model is called *re-entrant* when the source is always fired after the sink has completed execution. Note that the entity corresponding to an iteration body is a conditionally re-entrant model, where the condition is set by the iteration clause. A model is a *pipeline* when the source is fired before the sink has completed execution.

Several attributes can be given to the vertices and edges of a sequencing graph model. A *timed* sequencing graph model is one where each vertex is labeled by a *delay*. In the sequel, we refer to *propagation delay* as a

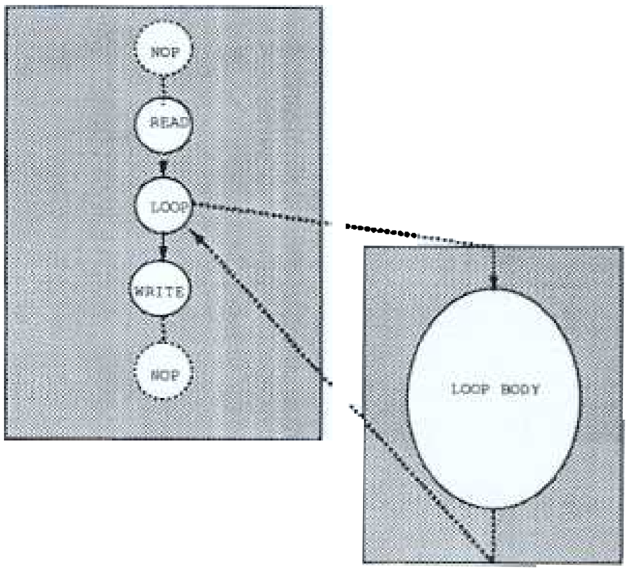

Figure 2: Example of hierarchical sequencing graph with an iteration construc

non-negative real number representing the delay through a combinational circuit implementing an operation. In the particular case of synchronous circuits, we refer to *execution delay* as the integer number of synchronous cycles to execute the operation.

In general the delay of a vertex can be *data-independent* or *data-dependent*. Only data-independent delays can be estimated before synthesis. Examples of operations with data-dependent delay are those that depend on external data, such as *data-dependent delay branching* and *iteration*. An example of the former case is a branch to two operations with different delays, where in the limit one branch body can be a No-Operation (e.g. a floating-point data normalization requiring conditional data alignment.) An example of the latter is an iteration whose exit condition is data-dependent. An arithmetic divisor, based on an iterative algorithm, can be modeled by an iterative construct. It is interesting to note that *external synchronization* can be modeled by an iteration of No-Operations, whose exit clause is the value of an external signal.

Data-dependent delays can be *bounded* or *unbounded*. The former case applies to data-dependent delay branching, where the maximum and minimum possible delay can be computed. It applies also to some iteration constructs, where the maximum and minimum number of iterations is known. The latter case is typical of some iteration constructs, such as those modeling external synchronization.

A sequencing graph model with data-independent delays can be characterized by its overall delay, called *latency*. When a sequencing graph entity has no complex vertices, then the latency is the length of the longest weighted path (from source to sink). Since the graph is acyclic, such a computation can be efficiently done in $O(|E|)$ time. Let us consider now sequencing graphs with complex vertices with data-independent delays. The latency computation can be performed by traversing the hierarchy bottom-up. The latency of a *model call* vertex is the latency of the corresponding graph entity. The latency of a *branching* vertex is the latency of one of the corresponding bodies. The latency of an *iteration* vertex is the latency of its body times the number of iterations. These considerations can be easily extended to the computation of latency bounds in presence of data-dependent bounded delays. Graphs with bounded delays (including data-independent) are called *bounded-latency* graphs. Otherwise they are called

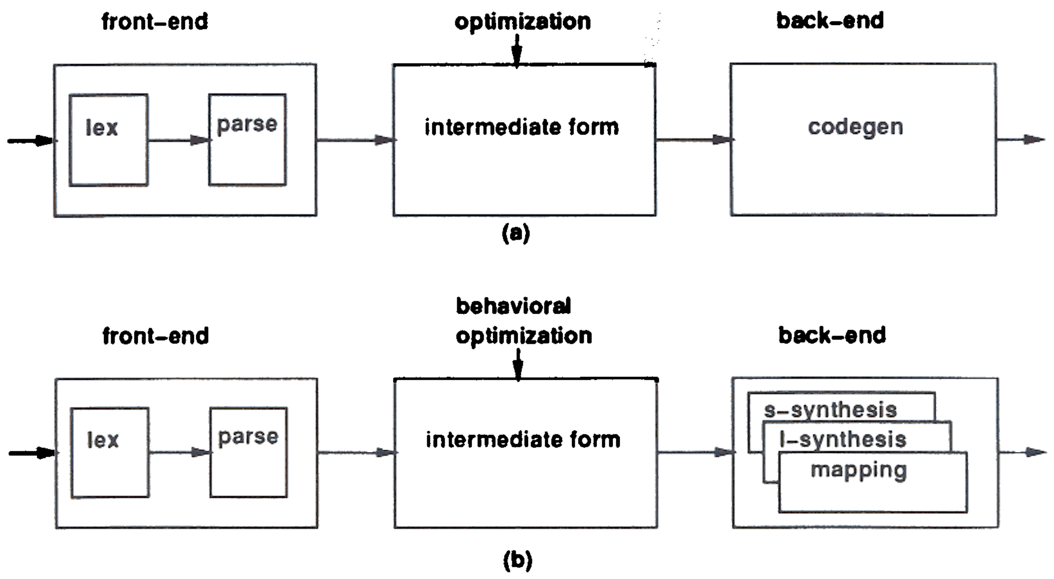

Figure 3: Anatomies of software and hardware compilers.

unbounded-latency graphs, because the latency cannot be computed.

3 Compilation and behavioral optimization.

We explain in this Section how circuit models, described by HDL programs, can be transformed into sequencing graphs, that will be used as starting point for synthesis in the following Sections. Most hardware compilation techniques have analogues in software compilation. Since hardware synthesis followed the development of software compilers, many techniques were borrowed and adapted from the rich field of compiler design [1]. Nevertheless, some behavioral optimization techniques are peculiar to hardware synthesis. We will briefly survey the general issues on compilation, where the interested reader can find a wealth of literature, and we will concentrate on the specific hardware issues.

A software compiler consists of a *front-end* that transforms a program into an *intermediate form* and a *back-end* that translates the intermediate form into the machine code for a given architecture. The front-end is language dependent, and the back-end varies according to the target machine. Most modern optimizing compilers improve the intermediate form, so that the optimization is neither language nor machine dependent.

Similarly, a hardware compiler can be seen as consisting of a front-end, an optimizer and a back-end. The back-end is much more complex than a software compiler, because of the requirements on timing and interface of the internal operations. The back-end exploits several techniques, that go under the generic names of structural synthesis, logic synthesis and mapping. We describe the front-end in Section 3 and the optimization techniques in Section 3.2.

3.1 Compilation techniques.

The front-end of a compiler is responsible for *lexical* and *syntax analysis*, *parsing* and creation of the intermediate form. A lexical analyzer is a component of a compiler that reads the source model and produces as an output a set of *tokens* that the parser then uses for syntax analysis. A lexical analyzer may also perform ancillary tasks, such as stripping comments and expanding macros.

A parser receives a set of tokens. Its task is to verify first that they satisfy the syntax rules of the language. The parser has knowledge of the grammar of the language and it generates a set of *parse trees*. A parse tree is a tree-like representation of the syntactic structure of a language. Syntactic errors, as well as some semantic errors (such as an operator applied to an incompatible operand), are detected at this stage. The error recovery policy depends on the compiler and on the gravity of the error. Software tools can be used to create lexical analyzers and parsers. The most commonly used ones are programs *lex* and *yacc* provided with the $UNIX^{TM}$ operating system.

While the front-end of a compiler for software and hardware are very similar, the subsequent steps may be fairly different. In particular, for hardware languages, diverse strategies are used according to their flavors.

The compilation of hardware models at the functional level involves a full *semantic analysis*, that comprises *data-flow* and *control-flow* analysis and *type checking*. Semantic analysis is performed on the parse trees in different ways. A common one is flattening the parse trees and creating the intermediate form. In doing this, the semantics of the model is checked. Type checking has some peculiarity when compiling HDLs. Operations on vectors of Boolean variables are checked for operand compatibility. Vectors may be padded with ones or zeroes to achieve compatibility in some cases.

The *overloading* of the arithmetic and relational operators has to be resolved at this stage. First, all metavariables need to be eliminated, by expanding the constructs that employ them, because metavariables have no corresponding hardware semantics. The operators on integer metavariables (both arithmetic and relational) have the usual meaning. By contrast, operations on Boolean vectors have to be mapped to hardware operators that do the corresponding function. For example, the sum of two Boolean vectors has to be resolved as a link to an adder circuit. Similarly, the comparison of two integers, to be implemented in hardware by Boolean vectors, has to be mapped to a link to a hardware comparator. Since the mapping to hardware resources is not always univalent, (because different hardware implementations have different area/performance parameters) abstract hardware operators are used at this stage and the binding to hardware resources is deferred to a later optimization stage (described in Section 6.)

The semantic analysis of the parse trees leads to the creation of the intermediate form, that represents the implementation of the original HDL program on an abstract machine. Such a machine is identified by a set of operations and dependencies, and it can be represented graphically by a sequencing graph. The hardware model in terms of an abstract machine is virtual in the sense that it does not distinguish the area and delay costs of the operations. Therefore, behavioral optimization can be performed on such a model while abstracting the underlying circuit technological parameters.

We assume here, for the sake of explanation and uniformity, that the sequencing graph model is used as intermediate form. Note that other intermediate models could be used, with similar meaning but different aspects. Similarly we assume here, for the sake of simplicity, that structured programming constructs are used (e.g. no unrestricted *goto* statements are employed), and that each model has a single entry and a single exit points. This allows us to interpret the hardware model as a sequencing graph that abides the definition given in Section 2.2.

Whereas the hierarchical structure of the sequencing graphs is derived from the *control-flow analysis* of the model, the graph topology is based on *data-flow analysis*. The parse trees for each assignment statement correspond then to the vertices of each graph entity. The edges are inferred by considering data-flow and serialization dependencies. Each sequencing graph entity corresponds to a *basic block* in compiler jargon.

Data-flow analysis comprises several tasks, and it is used as a basis for behavioral optimization. It entails the derivation of the variable *life-times*, i.e, the interval between their first definition (*birth*) and last reference (*death*). Note that sequencing graphs do not model explicitly the fact that variables need storage during their lifetimes, with a corresponding cost in terms of circuit implementation. When considering hardware models with imperative semantics, multiple assignments to variable may occur. Variables preserve their values until the next assignment. For hardware synthesis, it is often convenient to rename instances of variables, so that each instance has a single assignment and, of course, to resolve the references appropriately. A scheme for variable renaming is presented in reference [30].

3.2 Optimization techniques.

Behavioral optimization is a set of semantic-preserving transformations that minimize the amount of information needed to specify the partial order of tasks. No knowledge about the circuit implementation style is required at this stage. Behavioral optimization can implemented in different ways. It can be applied directly to the the parse trees, or during the generation of the intermediate form, or even on the intermediate form itself, according to the different cases. For the sake of explanation, we consider here these transformations as applied to sequences of statements.

Transformations for behavioral optimization of HDL models can be classified as data-flow and control-flow oriented. The former group resembles most to the transformations applied in software compilers. They rely on global data-flow analysis of the intermediate form.

3.2.1 Data-flow based transformations.

These transformations are dealt with in detail in most books on software compiler design [1] and [34]

Tree-height reduction. This transformation applies to the arithmetic expression trees, and strives to achieve the best-possible expression split into two-operand expressions, so that the parallelism available in hardware can be exploited at best. It can be seen as a local transformation, applied to each compound arithmetic statement, or as a global transformation, applied to all the compound arithmetic statements in a basic block. Enough hardware resources are postulated to exploit all the parallelism. If this is not the case, the gain of applying the transformation is obviously reduced.

Example 3.1. Consider the following arithmetic assignment: $x = a + b + c + d$; that can be trivially split as: $x = a + b$; $x = x + c$; $x = x + d$;. It requires three additions in series. Alternatively, the following split can be done: $p = a + b$; $q = c + d$; $x = p + q$;, where the first two additions can be done in parallel if enough resources are available (in this case, two adders). The second choice is better than the first one, because the corresponding implementation cannot be inferior, for any possible resource availability. □

Constant and variable propagation. Constant propagation, also called constant folding, consists of detecting constant operands and precomputing the value of the operation with that operand. Since the result may be again a constant, then the new constant can be propagated to those operations that use it as input.

Example 3.2. Consider the following fragment: $a = 0;\ \ b = a + 1;\ \ c = 2 * b;$. It can be replaced by $a = 0;\ \ b = 1;\ \ c = 2;$. □

Variable propagation, also called copy propagation, consists of detecting the *copies* of variables, i.e. the assignments like $x = y$, and using the right-hand side in the following references in place of the left-hand side. Data-flow analysis permits to identify the statements where the transformation can be done. In particular the propagation of y cannot be done after a different reassignment to x. Variable propagation gives the opportunity to remove then the copy assignment. Note that copy assignments may have been introduced by other transformations.

Example 3.3. Consider the following fragment: $a = x;\ \ b = a + 1;\ \ c = 2 * a;$. It can be replaced by $a = x;\ \ b = x + 1;\ \ c = 2 * x;$. Statement $a = x;$ may then be removed by dead code elimination, if there are no further references to a. □

Common subexpression elimination. The search for common logic subexpressions is best done by logic synthesis algorithms. The search for common arithmetic subexpressions relies in general on finding isomorphic patterns in the parse trees. This step is greatly simplified if the arithmetic expressions are reduced to two-input ones. Then, this transformation consists of selecting a target arithmetic operation, and searching for a preceding one of the same type and with the same operands. Operator commutativity can be exploited. Again, data-flow analysis is used in the search, to insure that in any matching expression the operands always take the same values. When a preceding matching expression is found, then the target expression is replaced by a copy of the variable that is the result of the preceding matching expression.

Example 3.4. Consider the following fragment: $a = x + y;\ \ b = a + 1;\ \ c = x + y;$. It can be replaced by: $a = x + y;\ \ b = a + 1;\ \ c = a;$. Note that a variable copy has been introduced for variable a, that can be propagated in the subsequent code. □

Dead code elimination. Dead code consists of all those operations that cannot be reached, or whose result is never referenced elsewhere. Such operations are detected by data-flow analysis and removed. Obvious cases are those statements that would follow a procedure *return* statement. Less obvious cases involve operations that just precede a *return* statement and whose results are not parameters of the procedure nor they affect any of its parameters.

Example 3.5. Consider the following fragment: $a = x;\ \ b = x + 1;\ \ c = 2 * x;$. If variable a is not referenced in the subsequent code, the first assignment can be removed. □

Operator strength reduction. Operator strength reduction means reducing the cost of implementing an operator by using a simpler one. Even though in principle some notion of the hardware implementation is required, very often general considerations apply. For example, a multiplication by two (or by a power of two) can be replaced by a shift. Shifters are always faster and smaller than multipliers in many implementations.

Example 3.6. Consider the following fragment: $a = x^2\ \ b = 3 * x$ It can be replaced by $x << 1;\ \ b = x + t;$. □

Code motion. Code motion often applies to loop invariants, i.e. quantities that are computed inside an iterative construct but whose values do not change from iteration to iteration. The goal is to avoid the repetitive evaluation of the same expression.

> **Example 3.7.** Consider the following iteration clause: **for** $(i = 1; i \leq a * b)\{$ It can be transformed to: $t = a * b;$ **for** $(i = 1; i \leq t)\{ \ \}$. □

3.2.2 Control-flow based transformations.

The following transformations are typical of hardware compilers. In some cases these transformations are automated, in others they are user-driven.

Model expansion. Writing structured models, by exploiting subroutines and functions, is useful for two main reasons: modularity and re-usability. Modularity helps in highlighting a particular task (or set of tasks). Often, calls to a model are done only once in a HDL model.

Model expansion consists in flattening locally the model call hierarchy. Therefore the called model disappears, being swallowed by the calling one. A possible benefit is that the scope of some optimization techniques (at different levels) is enlarged, yielding possibly a better final circuit. If the expanded model was called only once, there is no negative counterpart. Nevertheless, in the case of multiple calls, a full expansion leads to an increase in the size of the intermediate code and to the probable loss of the possibility of hardware sharing.

> **Example 3.8.** Consider the following fragment: $x = a + b;$ $y = a * b;$ $z = foo(x, y);$ where: $foo(p, q)\{t = q - p;$ $return(t);\}$. Then by expanding foo, we have: $x = a + b;$ $y = a * b;$ $z = y - x;$ □

Conditional expansion. A conditional construct can be always transformed into a parallel construct with a test in the end. Under some circumstances this transformation can increase the performance of the circuit. For example, this happens when the conditional clause depends on some late-arriving signals. Unfortunately this transformation precludes some possibilities for hardware sharing, because the operations in all bodies of the branching construct have to be performed.

A special case applies to conditionals whose clauses and bodies are evaluation of logic functions. Then, the conditional expansion is favorable because it allows us to expand the scope of logic optimization.

> **Example 3.9.** Consider the following fragment: $y = ab;$ **if** (a) $\{x = b + d;\}$ **else** $\{x = bd;\}$. The conditional statement can be flattened to: $x = a(b + d) + a'bd$ and by some logic manipulation, the fragment can be rewritten as: $y = ab;$ $x = y + d(a + b)$. □

Loop expansion. Loop expansion, or unrolling, applies to iterative constructs with data-independent exit conditions. The loop is replaced by as many instances of its body as the number of operations. The benefit is again in expanding the scope of other transformations. Needless to say, when the number of iterations is large, unrolling may yield a large amount of code.

> **Example 3.10.** Consider the following fragment: $x = 0;$ **for** $(i = 1; i \leq 3; i++$ $\{x = x + i;$ The loop can be flattened to: $x = 0;$ $x = x + 1;$ $x = x + 2;$ $x = x + 3$. □

Other transformations on loops are possible, such as moving the evaluation of the iterative clause from the top to the bottom of the loop [54].

Block-level transformations. Branching and iterative constructs segment the intermediate code into basic blocks. Such blocks correspond to the sequencing graph entities. Trickey studied the possibility of manipulating the size of the blocks, by means of block-level transformations, that include block merging and expansions of conditionals and loops. Even though he did not consider model expansion, the extension is straightforward. He assumed that operations in different blocks cannot overlap execution and that concurrency is limited only by the amount of hardware resources available for parallel execution in each block.

Therefore, collapsing blocks may provide more parallelism and enhance the average performance. To find the optimum number of expansions to be performed, he proposed five transformations, with rules to measure the expected performance improvement. The rules can then be applied bottom-up in the hierarchy induced by the control-flow hierarchy of the model. He proposed a linear-time dynamic programming algorithm, that returns an optimum block-level structure. Unfortunately, the optimality is weakened by the assumptions on the model and on the transformation rules. We refer the interested reader to reference [54] for further details.

4 Structural synthesis.

Structural synthesis is the creation of the macroscopic structure of a digital circuit. The starting point for structural synthesis is a circuit behavioral view at the functional level, that can be fully captured by a sequencing graph. The result of structural synthesis is a structural view, i.e. an interconnection of the major building blocks of a circuit. In other words, structural synthesis transforms a partial order of operations into an interconnection of operators and a corresponding control circuit.

Structural synthesis may be performed in many ways, according to the assumptions on the hardware being designed, the design style and the design goals. Therefore a large variety of problems, algorithms and tools have been proposed, that fall under the umbrella of structural synthesis. To be more specific, we address in this Section the synthesis problems for synchronous mono-phase digital hardware.

Even by focusing the structural synthesis task to one particular implementation style, the spectrum of solutions is still very wide. Indeed, the designer's goals in using a structural synthesis tool may be quite different. The major ones are to preserve the specified behavior, while optimizing the performance or the area of the implementation. Some further clarifications are needed at this point. First, the area and performance can only be estimated at this stage, because only the macroscopic structure of the circuit is dealt with. Second, worst case bounds on area and on performance may be required. No matter how fast a chip runs, its yield may drop above a certain size and/or manufacturing may be unfeasible. Similarly, compact implementations that fall below a certain level of performance may be irrelevant. Third, the circuit structure may be constrained to using some pre-specified units for some operations or to have auxiliary I/O or test circuits in appropriate positions.

In general, the designer may be interested in exploring a set of trade-off points corresponding to area/performance estimates. The *design space* consists of all feasible structures that correspond to a given behavior. Structural synthesis tools can be used to traverse the design space, by providing a designer with information about these estimates. Alternatively, structural synthesis can compute one point of the design space corresponding to an implementation that satisfies a particular optimality criterion.

We partition structural synthesis into two tasks. The former is the search for an optimal structure in the design space. Optimality can be defined according to different criteria and possibly subject to constraints. The underlying model is the sequencing graph model extended with annotations. The second task is to synthesize the data-path and the control circuits corresponding to the chosen structure as interconnection of logic blocks.

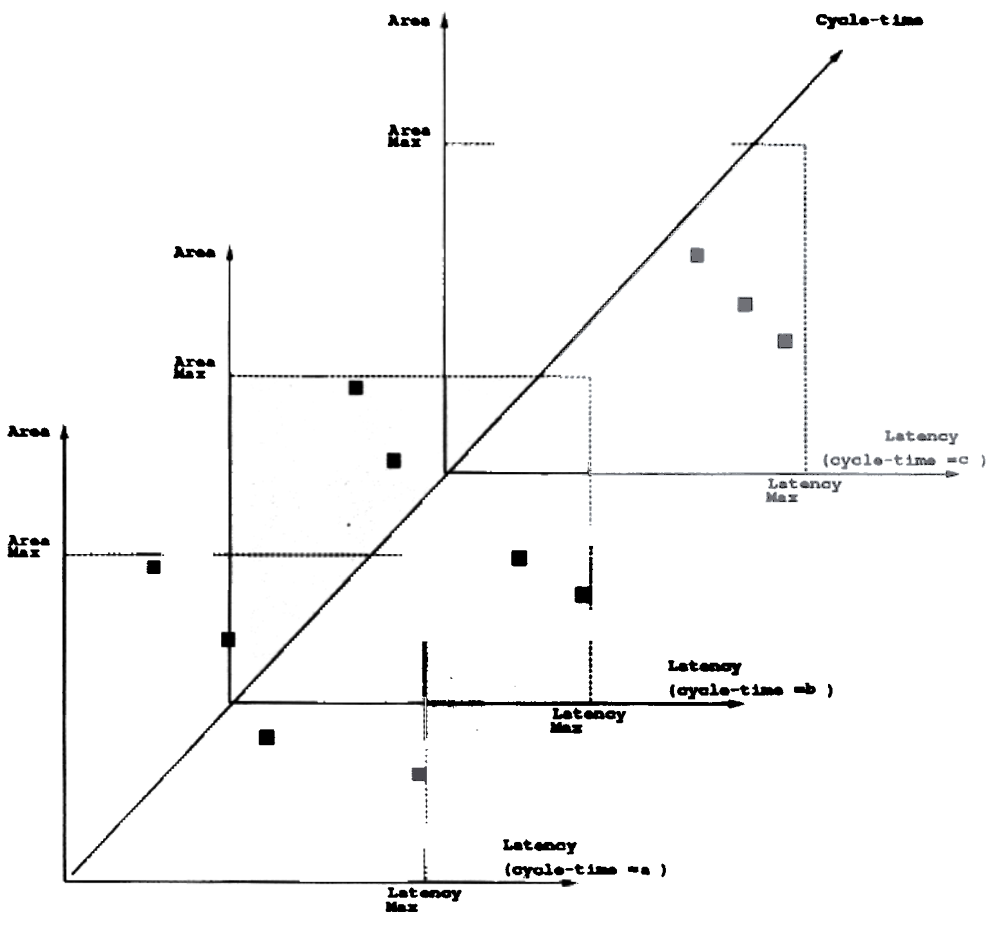

Figure 4: Sample points of the design evaluation space.

It is the goal of this Section to give an overview of the problems in structural synthesis and optimization. We defer the detailed descriptions of the algorithms for scheduling, data-path synthesis (including resource binding) and control synthesis to Sections 5 6 and 7 respectively.

4.1 The design space.

Circuits are often graded in terms of their area/performance trade-offs. The area consists of an estimate of the total size. In the case of non-pipelined synchronous systems, performance is expressed in terms of the *cycle time* (i.e. the clock period) and the *latency* (i.e. the number of cycles to perform all the operations.) It may be the case that one of these two parameters is fixed (e.g. the cycle time matches the inverse of system operation frequency) and the latter is derived as a function of the former. A third performance parameter is used in pipelined system design: the *throughput*. We defer the considerations on pipelined circuits until Section 8.

Informally speaking, the *design space* is the collection of all the feasible structures corresponding to a behavioral model. Each structure is associated to a triple (*area, latency, cycle-time*) that characterizes the structure. The *design evaluation space* is the ensemble of the corresponding triples, as shown in Figure 4.

Realistic design examples have shown that the design evaluation space is not a smooth surface. This is due to two reasons. First, the design space is a finite set of points, because the macroscopic structure of a circuit has a coarse grain. For example, a hypothetical circuit may have one or two multipliers, and its area would jump in correspondence of this choice. Second, there are several non-linear effects that are compounded in determining the area, latency and cycle-time as a function of the structure of the circuit.

The goal of structural optimization is to minimize one or more of the entries in the triple, under possibly some bounds on the remaining ones. Therefore, structural synthesis involves a *constrained multi-criteria optimization* problem. Due to the lack of compactness of the design space and of smoothness of the design evaluation space, the solution methods are fairly involved and rely on a combination of techniques that solve some related sub-problems.

4.2 Resources and constraints.

Hardware behavior is described by the set of the operations, their relations and by the inner models of the operations themselves. The operations and their relations can be represented by a sequencing graph, that captures the operations and their partial order.

The operations are preformed in hardware by operators, called *resources*. Resource models are also required by structural synthesis. It is important to remark that a circuit constructed by structural synthesis does not contain only resources. Indeed, it comprises also *steering logic circuits*, (e.g. multiplexers and busses), to send the data to the appropriate resources at the appropriate time, *registers* to hold data across cycle boundaries, and *control circuits* to sequence and synchronize the operations.

Constraints are also an integral part of the hardware model. Timing constraints, such as operation serialization and bounds on the separation between two operations can be seen as part of the required behavior. They can be described as additional relations added to the sequencing graph models. Other constraints, such as area or performance bounds, can be seen as frontiers that delimit the design space. They are not part of the hardware behavior, but they are part of the hardware specifications.

4.2.1 Resources.

Hardware resources are the circuits that implement the operations, corresponding to the vertices. A classification of the resources can be done according to the type of operations. Namely:

- *Functional resources* yield a result as a function of some input data. Examples are arithmetic operators (e.g adders, multipliers, ...), combinational and sequential logic functions.
- *Memory resources* store data. Examples are registers, read-only and read-write memory arrays.
- *Interface resources* support internal communication (e.g. busses) and external input/output functions, that allow the circuit to communicate with the external environment.

In addition, when considering hierarchical sequencing graph models, complex vertices may represent calls to other sequencing models. These models themselves, once synthesized in hardware, can be seen as combinational or sequential logic functions and treated as functional resources.

Functional resources represent the widest class of hardware operators. Indeed, there are as many kinds of these resources as there are many ways of implementing operations in hardware. Most algorithms for structural optimization exploit different choices and combinations of functional resources in the traversal of the design space.

Memory resources include registers and *memory arrays*, whose access can easily be modeled as transfer of data across circuit ports. Therefore an often used paradigm for memory resources is not to describe them explicitly in the sequencing graph model but assume that their usage is implied by the behavioral model.

Interface resources include busses, that may be used as a major means of communication inside a data-path. External I/O resources are in general standard circuits, and access to them can also be modeled as transfer of data across circuit ports.

We say that a circuit is *resource-dominated* if the total area and delay depend mainly upon the area and delay of the resources. This is the case of most DSP circuits, that employ several *standard* resources, such as arithmetic units. Conversely, the parameters of ASIC circuits depend often on control and on *application-specific* logic units. Structural synthesis of resource-dominated circuits benefits from simpler estimation methods.

4.2.2 Constraints.

Constraints in structural synthesis can be classified into two major groups: *interface constraints* and *implementation constraints*. Interface constraints are part of the hardware behavior. To understand the reason for this, we must view the circuit interface as a partition boundary between the behavior and the environment. This partition forces some implementation choices to be dictated by the environment, such as the size and the timing of the data transfer. The size of the data being transferred is related to the number of I/O pins of the chip. It is a hardware constraint that is generally related to the size of the *port variables* of the model. The timing of the data being transferred can be specified by means of detailed timing constraints, that specify the minimum and or maximum delay between any pair of operations, and in particular I/O operations. Detailed timing constraints are described in Section 5.1.2 as well as the algorithms to validate their consistency and to enforce them.

Implementation constraints reflect the desire of the designer to achieve a structure with some properties. Examples are area constraints and performance constraints, in terms of the *cycle time* (i.e. the clock period) and the *latency* (i.e. the number of cycles to perform all the operations.)

A different type of implementation constraints is the resource mapping constraint. In this case, a particular operation is constrained to be implemented by a given resource. These constraints are motivated by the designer's previous knowledge, or intuition, that one particular choice is the best and that other choices do not need investigation. Structural synthesis with resource mapping constraints is often referred to as *synthesis from partial structure* [26]. Design systems that support such a feature allow a designer to specify a system in a wide spectrum of ways, ranging from a full behavioral model to a structural one. This modeling capability may be useful to leverage previously designed components.

A common overall goal in structural synthesis is to maximize the circuit performance (e.g. minimize latency and/or cycle-times) under area constraints. Area estimation may be very complex, because it involves the computation of the resource usage, the steering and control logic usage, the register count and the wiring. Area and delay estimation are dealt with in Section 4.4.

Often the area is approximated by the functional resource usage. This approximation is justified in the case of *resource dominated* circuits, because of the reasons outlined above. It is far less justified for other kinds of circuits, including ASICs. Since structural synthesis techniques were investigated first on computational systems, the approximation is very common.

As a result, the problem of optimizing the performance *under resource constraints* has received a large attention. For general circuits, it is a heuristic approach to solve the corresponding area-constrained problem. Therefore, the maximum number of resource instances of a given type is often specified as part of the constraints. Synthesizing

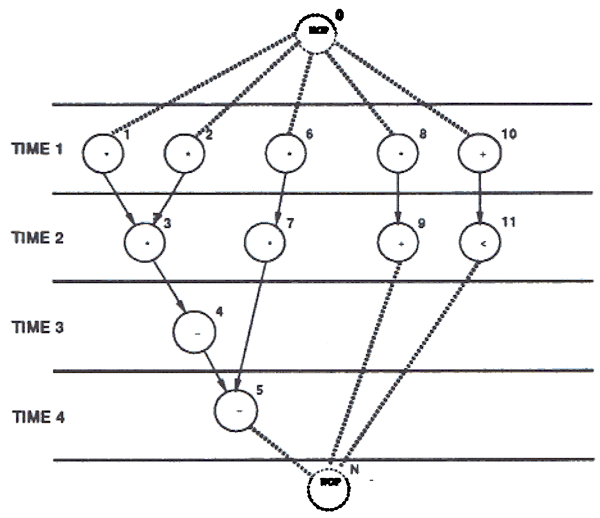

Figure 5: Scheduled sequencing graph.

a structure under varying resource bounds can be seen as a way of determining a set of points of interest of the design space.

4.3 Scheduling and binding.

We assume that the structural synthesis problems can be formulated by using a hardware model in terms of:

- A (possibly hierarchical) sequencing graph.
- A set of functional resources, fully characterized in terms of area and execution delays.
- A set of constraints.

Structural synthesis consists first of placing the operations in time and in space, i.e. determining the time interval for their execution and their mapping to the resources. Second, structural synthesis constructs the data-path and control circuits. We show now that the first task is equivalent to annotating the sequencing graph with additional information. For the sake of simplicity, we assume first non-hierarchical sequencing graph models with bounded delays. The extension to hierarchical models is straightforward.

Scheduling is the task of associating a *start time* to each operation of the model.

Let the *execution delays* of the operations, i.e. the number of cycles needed for execution, be denoted by the set $D^E = \{d_i \ ; \ i = 0, 1, \ldots, N\}$. A *schedule* of a sequencing graph is a function $\varphi : V \rightarrow Z^+$, where $\varphi(v_i) = t_i$ denotes the operation start time such that $t_i \geq t_j + d_j \ \forall \ i,j \ s.t. \ (v_j, v_i) \in E$. A *scheduled sequencing graph* is a vertex weighted sequencing graph, where each vertex is labeled by its start time. When timing constraints are specified, then the schedule must be consistent with them (See Section 5.1.2) .

Example 4.1. Consider the a sequencing graph of Figure 1. A scheduled sequencing graph is shown in Figure 5. All vertices have a start time corresponding to the index of the band that includes them. □

Let us consider now the relations among the operations and the resources. We define *type* of an operation the type of computation it performs. It may be an arithmetic operation, such as addition or multiplication, or an application-specific operation. More formally we define the type as a function $\mathcal{T} : V \rightarrow Y$, where Y is a set of enumerated types, such as $\{add, multiply, divide\}$.

We can extend the notion of type to the functional resources. We call *resource-type set* the set of resource types. An operation can be matched to a resource when their type is the same. Obviously, a feasible implementation requires that there are resources for all the types of operations in the specification. Therefore, without loss of generality, we can identify the resource-type set with the set Y. In the sequel, we identify this set with its enumeration, i.e. we set $Y = \{1, 2, \ldots, n_{res}\}$, where $n_{res} = |Y|$. It is obvious that No-Operations do not require any binding to any resource. Therefore, when referring to a binding, we consider the set of vertices excluding the source and sink, i.e. $V = \{v_i; i = 1, 2, \ldots, n_{ops}\}$.

It is interesting to note that there may be more than one operation with the same type. In this case, *resource sharing* may be applied. Similarly there may be more than one resource with the same type (e.g a ripple-carry and a carry-look-ahead adder). In this case, a *resource selection* (or *module selection*) problem arises.

A fundamental concept that relates operations to resources is *binding*. It specifies which resource implements an operation. A *resource binding* is a mapping $\beta : V \rightarrow Y \times Z^{+}$, where $\beta(v_i) = (t, j)$ denotes that the operation corresponding to $v_i \in V$, with type $\mathcal{T}(v_i) = t$, is implemented by the j^{th} instance of resource type $t \in Y$ for each $i = 1, 2, \ldots, n_{ops}$.

A simple case of a binding is that of using *dedicated resources*. In this case, each operation is bound to one resource, and the resource binding β is a one-to-one function.

Example 4.2. Consider the scheduled sequencing graph of Figure 5. There are 11 operation. Assume that 11 resources are available. In addition, assume that the resource types are $\{multiplier, ALU\}$, where the ALU can perform addition, subtraction and comparisons. We label the multiplier as type 1, the ALU as type 2. We need 6 instances of the multiplier type and 5 instances of the ALU type. Then $\beta(v_1) = (1,1)$, $\beta(v_2) = (1,2)$, $\beta(v_3) = (1,3)$ and so on. □

A resource binding may associate one instance of a resource-type to more than one operation. In this case, that particular resource is *shared*. A necessary condition for a resource binding to produce a valid circuit implementation is that the operations corresponding to shared resource do not execute concurrently.

Example 4.3. It is obvious that the resource usage of the previous example is not efficient. Indeed only four multipliers and two ALU are required by the scheduled sequencing graph of Figure 5. This is shown in Figure 6. Now $\beta(v_1) = (1,1)$, $\beta(v_2) = (1,2)$, $\beta(v_3) = (1,2)$ and so on. □

When binding constraints are specified, a resource binding must be compatible with them. In particular, a partial binding may be part of the original specification. This corresponds to specifying a binding for a subset of the operations $U \subseteq V$. A resource binding is *compatible* with a partial binding when its restriction to the operations U is identical to the partial binding itself.

Bounds on the maximum usage of a resource are often specified. We denote by $\{a_i \; ; \; i = 1, 2, \ldots, n_{res}\}$ the maximum usage of each resource type. These bounds represent the *allocation* [1] of instances for each resource type.

[1]The term allocation has often been misused in the literature. Some authors refer to binding as allocation. We prefer to use the term 'resource bounds' and 'binding' in this chapter, and we shall not use the term 'allocation' at all.

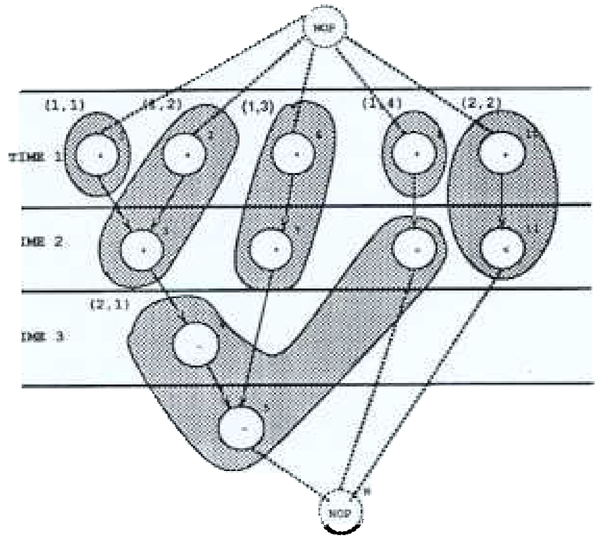

Figure 6: Scheduled sequencing graph with resource binding.

A resource binding is *compatible* with resource bounds when $\beta(v_i) = (t, j)$ satisfies $j \leq a_t$ for each operation v_i; $i = 1, 2, \ldots, n_{ops}$.

A scheduled and bound sequencing graph is a *feasible* structure when it satisfies all the constraints that are specified. We can now formalize the concept of the design space, by defining it as the collection of all *feasible* structures. Each feasible structure can be labeled with area and latency estimates using the criteria shown in Section 4.4. The former can be approximated by the sum of the areas of all the bound instances of each resource type. The latter can be computed as the start time of the sink vertex, i.e. t_N, which corresponds to the length of the schedule.

It is important to remark that the present formulation characterizes the design space in terms of area and latency, but it is also a function of the cycle-time parameter. Indeed the dependency on the cycle-time is due to the computation of the execution delays. By considering different values of the cycle-time, the corresponding execution delay vary and so do the corresponding feasible structures.

4.4 Estimation.

Accurate area and delay estimation is not a simple task. Much of the complexity of the structural synthesis problems is due to the difficulty of estimating the impact of a high-level decision on the structure of a circuit on the final area and performance.

Let us consider first resource-dominated circuits. A simple model is to assume that the area and the delay of the resource dominate, so that other factors can be neglected. This is a valid assumption in the case of DSP circuits.

The area estimate of a structure is the sum of the areas of the bound resource instances. Equivalently, the total area is a weighted sum of the resource usage. A binding specifies fully the total area, but it is not necessary to know the binding to determine the area. Indeed, it is just sufficient to know how many instances of each resource are used.

The latency of a circuit can be determined by its schedule. It is the start time t_N of the sink operation. In the case that no constraints on the resource usage are imposed, then the latency can be derived directly from the sequencing graph by computing the length of the longest weighted path.

Let us consider now general circuits, and let us consider in more detail the area and delay estimation problem.

Registers. All data that is transferred from a resource to another across a cycle boundary must be stored into some register. An upper bound on the register usage can then be derived by examining a scheduled sequencing graph. This bound is in general loose, because the number of registers can be minimized, as shown in Section 6.4. The binding information is needed for evaluating and/or performing the register optimization. Therefore, the accurate estimation of the number of registers requires both scheduling and binding.

The effect of registers on the evaluation of the cycle-time is easy to compute. In fact, their *set up* and *propagation delay* times must be added to the propagation delays of the combinational logic. It is more efficient to consider a reduced cycle-time in all the computations, that already discounts set up and propagation delays.

Steering logic. Steering logic affects the area and the propagation delay. While the area of multiplexers can be easily evaluated, their number requires the knowledge of the binding. Similarly, multiplexers add propagation delays to the resources, and the overall propagation delay must not exceed the cycle-time times the execution delay. Busses can also be used to steer data. In this case appropriate models should be used.

Wiring. Wiring contributes to the area and to the delays. The wiring area overhead can be estimated from the structure, once a binding is known, by using models that are appropriate of the physical design style of the implementation. The wiring delays are crucial. As in the case of steering logic, they add propagation delays and we must insure that the overall propagation delay is bounded. Unfortunately, estimating the wiring requires the knowledge of the structure (i.e. the binding) as well as the placement of the physical implementation of the resources. Fast floor-planners have been used. Alternatively, statistical wiring models have been used. In this case, it has been shown that the average interconnection length is proportional to the total number of blocks to the α power, where $0 \leq \alpha \leq 1$. The wiring delay and area track with the average interconnection length. We refer the interested reader to reference [47] for a detailed analysis of the wiring area and delay estimation.

Control logic. The control circuit contributes to the overall area and delay, because some control signals can be part of the critical path. Recently, the interest in synthesizing control-dominated circuits, such as some communication ASICs, has exposed the importance and difficulty of the problem. Simple models for estimating the size of the control circuit can be based on the latency. Consider bounded-latency non-hierarchical sequencing graphs. Read-only memory based implementations of the control units require an address space wide enough to accommodate all control steps and a word-length commensurate to the number of resources being controlled. Hard-wired implementations can be modeled by *finite-state machines* with as many states as the latency. Unfortunately, these models may provide loose bounds, because many optimization techniques can be applied to the controller, such as word-length reduction by encoding and state encoding. In addition, general models for sequencing graphs require more complex control units, as shown in Section 7, and estimating accurately the area and extracting the critical sub-path in the controller is a difficult task.

5 Scheduling.

Scheduling is a very important task in high-level synthesis. Whereas a sequencing graph denotes the partial order of the operations to be performed, the scheduling of a sequencing graph determines the detailed starting time for each operation. As a result, the degree of concurrency of the operations is determined by the scheduling task.

The start time of the operations must satisfy the original dependencies of the sequencing graph. These dependencies limit the amount of parallelism of the operations, because any pair of operations related by a sequential dependency (or a chain of dependencies) may not execute concurrently. As a limiting case, a scheduled sequencing graph may be such that the operations are fully serialized with respect to each other.

Scheduling a sequencing graph determines the concurrency of the resulting design, and therefore it affects its performance. By the same token, the maximum number of concurrent operations of a given type during the entire schedule is a lower bound on the number of required hardware resources for that operation. Therefore the choice of a schedule affects also the area of the implementation.

The number of resources of a given type may be constrained from above, to satisfy some requirements related to the physical design. For example, a circuit with a prescribed size may have at most one floating point multiplier/divider. When resource constraints are imposed, the number of operations of a given type that can overlap in time is limited by the number of resources of that type. Therefore tight bounds on the resources correlate to serialized implementations. A spectrum of solutions may be obtained by scheduling a sequencing graph with different constraints. This is indicative of the possible area-performance trade-off points in the design space.

We consider first sequencing graphs that are not hierarchical. We analyze the scheduling problem without resource constraints in Section 5.1 and with resource constraints in Section 5.2. We consider then extensions to the model in Section 5.3. We assume that execution delays are known, i.e. that all operations have data-independent delays. Exceptions are described in Section 5.1.3.

5.1 Scheduling without resource constraints.

We consider here scheduling with no resource constraints. Let us denote by $T = \{t_i \; ; \; i = 0, 1, \ldots, N\}$ the *start time* for the operations, i.e. a set of integer numbers denoting the cycle in which a particular operation starts. The *latency* of the schedule is the number of steps to execute the entire schedule, or equivalently the start time of the sink t_N. An unconstrained schedule is a set of values of the start times T, that satisfies the sequencing relations relations, i.e. $t_i \geq t_j + d_j \quad \forall \; i, j \quad s.t. \; (v_j, v_i) \in E$. The *minimum latency unconstrained scheduling problem* can be defined as follows.

Definition 5.1 *Given a set of operations V with integer delays D^E and a partial order on the operations , find an integer labeling of the operations $\varphi : V \rightarrow Z^+$, such that $t_i = \varphi(v_i)$, $t_i \geq t_j + d_j \; \forall \; i, j \; s.t. \; (v_j, v_i) \in$ and t_N is minimum.*

The unconstrained scheduling problem can be solved in polynomial time. Before considering the algorithms for its solution, we would like to comment on the relevance of the problem.

It is obvious that the problem is important when the number of resources of a given type affects only marginally the overall quality of the solution. An example is the case in which the area cost of the resources is small compared to the overhead of wiring and multiplexing the data to the resources being shared. In this case, the use of dedicated resources is preferred. A similar situation is when the operations require resources of different types, so that there is just a resource for each operation.

Unconstrained scheduling is also used when the decision on resource sharing and their binding to operations is done prior to scheduling. In this case, the area cost of an implementation is defined before and independently from the scheduling step. Eventually unconstrained scheduling can be used to derive bounds on latency for constrained problems. By relaxing the resource constraints, a lower bound can be computed, because the minimum latency of a schedule under some resource constraint is obviously at least as large as the latency computed with unlimited resources. Conversely, by assuming dedicated resources, an upper bound can be computed. These bounds are useful in simplifying the solution of the constrained problem.

5.1.1 The ASAP and ALAP scheduling algorithms.

We consider here the minimum latency scheduling problem. This problem can be solved in polynomial time by topologically sorting the vertices of the sequencing graph. This approach has been called *As Soon As Possible* (ASAP) scheduling algorithm, because the start time for each operation is the minimum allowed by the dependencies. The algorithm can be summarized by the following program:

```
ASAP ( $G(V,E)$ )
{
        Schedule $v_0$ by setting $t_0 = 0$;
        repeat {
                Select a vertex $v_i$ whose predecessors are all scheduled;
                Schedule $v_i$ by setting $t_i = \max_{j\ s.t.(v_j,v_i)\in E} t_j + d_j$;
        }
        until ($v_N$ is scheduled)
```

The computational complexity of the algorithm is $O(|E|)$.

We consider now the case in which a schedule must satisfy an upper bound on the latency, denoted by λ. The problem may be solved by executing the ASAP scheduling algorithm and verifying that $t_N \leq \lambda$.

If a schedule exists that satisfies the latency bound λ, it is possible then to explore the range of values of the start times of the operations that meet the bound. The ASAP scheduling algorithm yields the minimum values of the start times. A complementary algorithm, the *As Late As Possible* (ALAP) scheduling algorithm provides the corresponding maximum values. Here is a description of the algorithm.

```
ALAP( $G(V,E), \lambda$)
{
        Schedule $v_N$ by setting $t_N = \lambda$;
        repeat {
                Select vertex $v_i$ whose successors are all scheduled;
                Schedule $v_i$ by setting $t_i = \min_{j\ s.t.(v_i,v_j)\in E} t_j - d_i$ ;

        until ($v_N$ is scheduled) ;
```

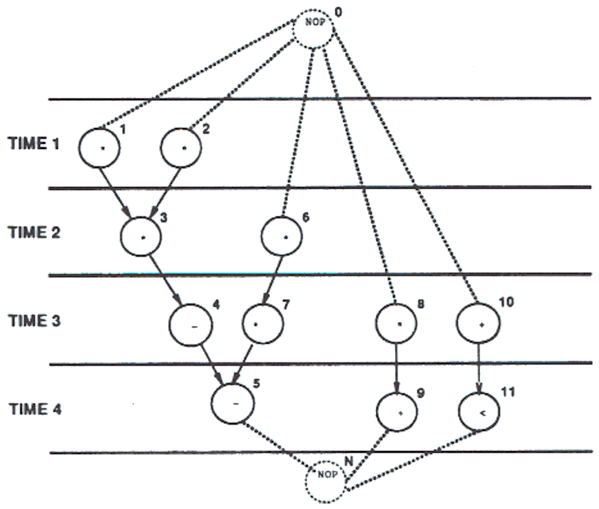

Figure 7: ALAP Schedule under a latency constraints of four steps.

}

The computational complexity of the algorithm is $O(|E|)$.

The ALAP scheduling algorithm is also used for unconstrained scheduling. In this case, the latency bound λ is chosen to be the length of the schedule computed by the ASAP algorithm, i.e. $\lambda = t_N$. An important quantity used by some scheduling algorithm is the *mobility* of an operation, denoted by μ_i, corresponding to the difference of the start times computed by the ALAP and ASAP algorithms. Zero mobility implies that an operation can be started only at one given time step in order to meet the overall latency constraint. When the mobility is larger than zero, then it measures the span of the time interval in which it may be started.

Example 5.1. An example of ASAP schedule is shown in Figure 5 for the sequencing graph of Figure 1. An example of ALAP schedule is given in Figure 7, under the assumption that the schedule should satisfy a latency bound of four steps. By comparing the two schedule, it is possible to deduce that the mobility of operations 1 through 5 is zero, i.e. they are on the critical path. The mobility of operations 6 and 7 is one, while the mobility of the remaining ones is two. □

The ASAP and ALAP algorithms are often used to derive bounds for resource constrained scheduling. In this case, the ASAP algorithm can be used to derive lower bounds on the start time of operations (by relaxing the resource constraints) and an upper bound t_N on the latency (by assuming dedicated resources). Upper bounds on the start times of the operations can be computed by the ALAP algorithm with $\lambda = t_N$.

5.1.2 Scheduling under relative timing constraints.

We consider in this Section *relative* timing constraints, that bind the time separation between operations pairs, regardless of their absolute value. Such constraints are very useful in hardware modeling, because the absolute schedule is not known *a priori*. Minimum timing constraints between any two operations can be used to insure that an operation follows another by at least a prescribed number of time steps, regardless of the existence of a dependency between them. It is often also important to limit the maximum distance in time between two operations by means of maximum timing constraints. The combination of maximum timing constraints with the minimum timing constraint permits us to specify the exact distance in time between two operations and, as a special case, their simultaneity. For example, consider a circuit with two independent streams of operations, that are constrained to communicate simultaneously to the external circuits by providing two pieces of data at two interfaces. The cycle in which the data are made available is irrelevant although the simultaneity of the operations is important. This requirement can be captured by setting a minimum and a maximum timing constraint of zero cycles between the two write operations.

We define more formally the timing constraints as follows:

- A *minimum* timing constraint $l_{ij} \geq 0$ implies that: $t_j \geq t_i + l_{ij}$

  A *maximum* timing constraint $u_{ij} \geq 0$ implies that: $t_j \leq t_i + u_{ij}$

A schedule under timing constraints is a set of start times for the operations satisfying the requirements stated in Definitions 5.1 and 5.3, and in addition:

$$\begin{aligned} &\geq t_j + l_{ij} \quad \forall l_{ij} \\ t_i &\leq t_j + u_{ij} \quad \forall u_{ij} \end{aligned} \qquad 2)$$

A consistent modeling of minimum and maximum timing constraints can be done by means of a constraint graph $G_c(V_c, E_c)$, that is an edge-weighted directed graph derived from the sequencing graph as follows. The constraint graph $G_c(V_c, E_c)$ has the same vertex set as $G(V, E)$ and its edge set includes the edge set E. Such edges are weighted by the the delay of the operation corresponding to their tail. The weight on the edge (v_i, v_j) is denoted by w_{ij}. Additional edges are related to the timing constraints. For every minimum timing constraint l_{ij}, we add a *forward* edge (v_i, v_j) in the constraint graph with weight equal to the minimum value $w_{ij} = l_{ij} \geq 0$. For every maximum timing constraint u_{ij}, we add a *backward* edge (v_j, v_i) in the constraint graph with weight equal to the negative of the maximum value $w_{ji} = -u_{ij} \leq 0$, because $t_j \leq t_i + u_{ij}$ implies $t_i \geq t_j - u_{ij}$. Note that the overall latency constraint can be modeled as a maximum timing constraint $u_{0,N}$ between the source and sink vertices.

Example 5.2. Consider the example in Figure 8. A minimum timing constraint requires operation v_4 to take place at least $l_{04} = 3$ cycles after operation v_0 has started. A maximum timing constraint requires operation v_3 to take place at most $u_{31} = 3$ cycles after operation v_1 has started. Note that the constraint graph has a backward edge with negative weight (e.g. -3). □

The presence of maximum timing constraints may prevent the existence of a consistent schedule, as in the case of the latency constraint. In particular, the requirement of an upper bound on the time distance between the start time of two operations may be inconsistent with the time required to execute the first operation, plus possibly the time required by any sequence of operations in between. Similarly, minimum timing constraints may also conflict with maximum timing constraint.

A criterion to determine the existence of a schedule is to consider in turn each maximum timing constraint u_{ij}. The longest weighted path in the constraint graph between v_i and v_j (that determines the minimum separation in time between operations v_i and v_j) must be less than or equal to the maximum timing constraint u_{ij}. As a

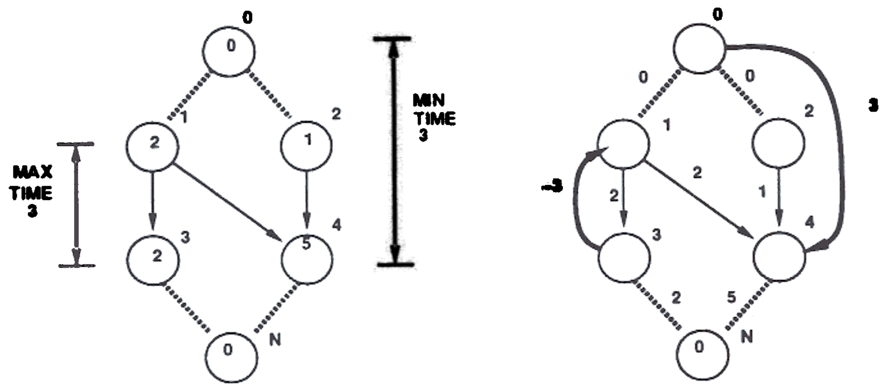

Figure 8: Example of a constraint graph, with a minimum and a maximum timing constraint. The number inside a vertex represents its execution delay.

consequence, any cycle in the constraint graph going through (v_j, v_i) must have negative or zero weight. Therefore, a necessary condition for the existence of the schedule is that the constraint graph does not have strictly positive cycles. We state without proof that the condition is also sufficient [30].

The existence of a schedule under timing constraints can be checked using the Bellman-Ford algorithm. It is often the case that the maximum timing constraints are fewer compared to the number of edges in the constraint graph. Then, relaxation-based algorithms like Liao-Wong's [36] can be more efficient. When a schedule exists, the length of the longest path from the source to a vertex is also the minimum start time. Thus the Bellman-Ford or the Liao-Wong algorithms provide also the schedule.

Example 5.3. A schedule for the constraint graph of the previous example, satisfying timing constraints, given by the following table.

Vertex	*Start time*
v_0	0
v_1	0
v_2	0
v_3	2
v_4	3
v_N	8

□

5.1.3 Relative scheduling.

We extend the notion of scheduling to the case of operations with *unbounded delays*. Such operations may model synchronization primitives, or data-dependent operations, such as the computation of the quotient of two numbers using iterative methods. While their execution delay is unknown, we assume that a completion signal is issued when the operation has finished its execution. The scheduling problem can be still modeled by a sequencing graph

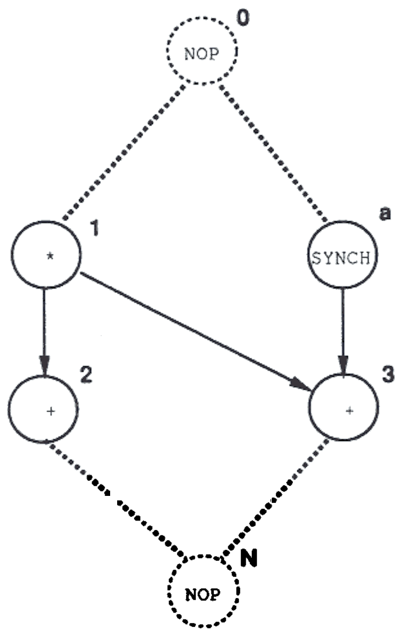

Figure 9: Example of a sequencing graph, with a synchronization operation with unknown delay.

$G(V,E)$, where a subset of the vertices has unspecified execution delay. Such vertices, as well as the source vertex, provide a frame of reference for determining the start time of the operations.

Definition 5.2 *The* **anchors** *of a constraint graph* $G(V,E)$ *consist of the source vertex* v_0 *and all vertices with unbounded delay, and are denoted by* $A \subseteq V$.

The start time of the operations cannot be determined on an absolute scale in this case. Nevertheless the start time of the operations can be computed as a function of the completion signals of the anchors and of the schedule of the operations *relative* to the anchors.

> **Example 5.4.** Consider the sequencing graph of Figure 9. There are three operations with known delay v_1, v_2, v_3 and one synchronization operation, denoted by a. Let us assume that the execution delay of a multiply operation is 2 and of an addition is 1. The start times in the graph depend on the start time of the source vertex t_0 and on the completion signal of the synchronization operation. Such a signal arrives at time $t_a + d_a$, where t_a is the time at which the synchronizer is activated and d_a is the unknown synchronization delay. The start times of v_1 and v_2 can be computed with reference to t_0. Namely, v_1 can start at t_0 and v_2 can start at $t_0 + 2$. The third operation can start no earlier than the synchronization signal at time $t_a + d_a$ and no earlier than $t_0 + 2$, i.e. its start time is $max\{t_0 + 2; t_a + d_a\}$. □

We summarize here the computation of the start times by means of the relative scheduling approach. We refer the interested reader to reference [30] for details. The anchors capture the *unknown* factors that affect the activation time of an operation. If we generalize the definition of the *start time* of a vertex in terms of partial schedules relative to the *completion* time of each anchor then it is possible to completely characterize the temporal relationships among the operations. In particular, let t_i^a be the schedule of operation v_i with respect to anchor a, computed by taking the subgraph induced by the successors of a, assuming that a is the source of the graph and that all anchors have zero execution delay. Let d_a be the unbounded unknown execution delay of anchor a. The start time of a vertex v_i is computed recursively as follows:

$$t_i = \max_{a \in A}\{t_a + d_a + t_i^a\} \tag{3}$$

In practice, a subset of the anchors A, called *relevant anchor set* is sufficient to determine the start time [30]. Note that if there are no operations with unbounded delays, then the start times of all operations will be specified in terms of time offsets from the source vertex, which reduces to the traditional scheduling formulation.

A *relative schedule* is a collection of schedules with respect to each anchor, or equivalently a set of offsets with respect to the relevant anchors for each vertex. From a practical point of view, the start times of Eq. 3 cannot be computed. However, the offset values and the anchor completion signals are sufficient to construct a control circuit that activates the operations at the appropriate times.

Relative scheduling can be performed under timing constraints. The constraint graph formulation still applies although the weights on the edges whose tails are anchors are unknown. Also in this case a schedule may or may not exist under timing constraint. It is important to be able to assess the existence of a schedule for any value of the unbounded delays, because these values are not known when the schedule is computed. We call a sequencing graph *well-posed* when it has this important property. Relative scheduling can be applied to well-posed graphs, to determine the start times of the operations. An important issue, is the verification of the well-posedness property. Related algorithms are reported in reference [30].

5.2 Scheduling with resource constraints.

Scheduling under resource constraints is an important and difficult problem to solve. Resource constraints stem from the fact that they give a rough measure of the area utilization, for some applications. Consider for example DSP filters, that use extensively bit-parallel addition and multiplication. The overall area of the implementation is affected mainly by the number of resources. Therefore, a maximum number of adders and of multipliers can be required to insure physical feasibility of the implementation or as a way of determining an area-performance trade-off point in the design space. Ideally, the entire design space can be characterized by solving the scheduling problem under different resource constraints. In practice two difficulties arise. First, the resource-constrained scheduling problem is intractable, and only near-optimal solutions can be found for problems of reasonable size. Second, the area-performance trade-off points are affected by other quantities, related to the area and delay of multiplexers, the length of the wires and the number of registers.

A *resource-constrained scheduling problem* is one such that the number of resources of any given type are bounded from above by a set of integers $\{a_k; k = 1, 2, \dots, n_{res}\}$. Therefore the operations are scheduled in such a way that the number of executing operations of a given type in every schedule steps does not exceed the bound. The *minimum latency resource-constrained scheduling problem* can be defined as follows.

Definition 5.3 *Given a set of operations V with integer delays D^E, a partial order on the operations E, and upper bounds $\{a_k; k = 1, 2, \dots, n_{res}\}$, find an integer labeling of the operations $\varphi : V \to Z^+$ such that $t_i = \varphi(v_i)$, $t_i \geq t_j + d_j \;\forall\; i, j$ s.t. $(v_j, v_i) \in E$, $|\{v_i | T(v_i) = k$ and $t_i \leq j < t_i + d_i\}| \leq a_k$ for each operation-type $k = 1, 2, \dots, n_{res}$ and schedule step j, and t_N is minimum.*

When all the resources are of a given type (e.g. ALUs), then the problem reduces to the classical multiprocessor scheduling problem. The minimum latency multiprocessor scheduling problem is intractable.

5.2.1 The Integer Linear Programming model.

A formal model of the scheduling problem under resource constraints can be achieved by using binary decision variables with two indices: $X = \{x_{il}; i = 1, 2, \ldots, N; l = 1, 2, \ldots, L\}$. The number L represents an upper bound on the latency, because the schedule latency is unknown. The bound can be computed by using a fast heuristic scheduling algorithm, such as a list scheduling algorithm (Section 5.2.2). In the sequel, we denote the summations over all operations as $\sum_i$ (instead of $\sum_{i=0}^{N}$) and those over all schedule steps as $\sum_l$ (instead of $\sum_{l=1}^{L}$) for the sake of simplicity.

The binary variable, x_{il}, is TRUE (i.e. 1) only when operation v_i starts in step l of the schedule, i.e. $l = t_i$. Equivalently, we can write $x_{il} = \delta(t_i, l)$, where $\delta(p, q)$ is the Kronecker delta notation. Therefore the start time of operation v_i can be stated in terms of x_{il} as: $t_i = \sum_l l \cdot x_{il}$.

The following model expresses the constraints on the schedule in terms of the binary variables. First, operations start only once:

$$\sum_l x_{il} = 1 \quad i = 1, 2, \ldots, N \qquad (4$$

Second, the sequencing relations represented by $G(V, E)$ must be satisfied. Therefore, $t_i \geq t_j + d_j \; \forall i, j \; s.t.(v_j$ E implies:

$$\sum_l l \cdot x_{il} \geq \sum_l l \cdot x_{jl} + d_j \quad i, j = 1, 2, \ldots, N \quad s.t.(v_j, v_i) \in E \qquad (5)$$

Third, the resource bounds must be met at every schedule time step. An operation v_i is executing at time step j when $\sum_{l=j-d_i+1}^{j} x_{il} = 1$. Therefore the number of all the operations executing at step j of type k must be lower than or equal to the bound a_k. Namely:

$$\sum_{i \; s.t. T(v_i)=k} \; \sum_{l=j-d_i+1}^{j} x_{il} \leq a_k \quad k = 1, 2, \ldots, n_{res}; \quad j = 0, 1, \ldots, t_N \qquad (6)$$

Let us denote by $\underline{t}$ the vector whose entries are the start times. Then, the minimum latency scheduling problem under resource constraints can be stated as follows:

$$\begin{aligned} & min \quad \|\underline{t}\| \quad such \; that \\ \sum_l x_{il} &= 1 \quad i = 1, 2, .. \; N & (7) \\ \sum_l l \cdot x_{il} - \sum_l l \cdot x_{jl} - d_j &\geq 0 \quad i, j = 1, 2. \quad N \; s.t.(v_j, v_i) \in E & (8) \\ \sum_{i s.t. T(v_i)=k} \; \sum_{l=j-d_i+1}^{j} x_{il} &\leq a_k \quad k = \quad 2, \quad n_{res}; \; j = 0, 1, .. \; t_N & (9) \end{aligned}$$

The choice of the norm of vector $\underline{t}$ to be minimized relates to slightly different goals. The infinity norm corresponds to minimize the maximum entry of $\underline{t}$, i.e. t_N. Therefore the objective function to minimize is:

$\sum_l l \cdot x_{Nl}$. The first order norm corresponds to minimizing the sum of the entries in $\underline{t}$, i.e. finding the earliest start times such that the constraint equations are satisfied. This is equivalent to minimize $\sum_i \sum_l l \cdot x_{il}$. Note that both cases correspond to minimizing a weighted sum of the variables X.

It is interesting to remark that detailed timing constraints can be incorporated in the model by adding the corresponding constraints equation in terms of $t_i = \sum_l l \cdot x_{il}$.

From a practical point of view, it is possible to focus on the interesting values of x_{il}. Indeed, the ASAP and ALAP algorithms give bounds for the start time of any operation, say v_i, as mentioned earlier. Let t_i^s be the start time for v_i computed by the ASAP algorithm and t_i^l the one computed by the ALAP algorithm. Then x_{il} is necessarily zero for $l < t_i^s$ and $l > t_i^l$. Therefore the summations with argument x_{il} can be restricted to $\sum_{l=t_i^s}^{t_i^l}$.

Example 5.5. Let us consider again the sequencing graph of Figure 1. We assume that there are two types of resources: a multiplier and an ALU that performs addition/subtraction and comparison. Both resources execute in one cycle. We also assume that the upper bounds on the number of both resources is 2; i.e. $a_1 = 2$ and $a_2 = 2$.

The full set of constraints for solving the minimum latency problem can be derived in a straightforward way. Instead, we would like to show how a simplified set of constraints can be derived, that fully characterizes the start times in a minimum latency solution.

By using a heuristic algorithm we find an upper bound on the latency of 4 steps. By executing the ASAP and ALAP algorithm on the unconstrained model, we can derive bounds on the start times. Note that the schedule of the ALAP algorithm, shown in Figure 5, already shows that the constrained schedule of 4 steps is an optimum one.

Let us consider the constraints in this situation. First, all operations must start only once.

$$\begin{aligned}
x_{11} & \\
x_{21} &= \\
x_{32} &= \\
x_{43} &= 1 \\
x_{54} &= 1 \\
x_{61} + x_{62} &= 1 \\
x_{72} + x_{73} &= \\
x_{81} + x_{82} + x_{83} &= \\
x_{92} + x_{93} + x_{94} &= \\
x_{10,1} + x_{10,2} + x_{10,3} &= \\
x_{11,2} + x_{11,3} + x_{11,4} &= \\
x_{N4} &=
\end{aligned}$$

Note that the last constraint implies that the start time of the sink vertex is 4, i.e. the overall latency is 4

We consider then the constraints based on sequencing. Namely

$$\begin{aligned}
x_{61} + 2x_{62} - 2x_{72} - 3x_{73} + 1 &\leq 0 \\
x_{81} + 2x_{82} + 3x_{83} - 2x_{92} - 3x_{93} - 4x_{94} + 1 &\leq 0 \\
x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} + 1 &\leq 0 \\
4x_{5,4} - 4x_{N4} &\leq 0
\end{aligned}$$

$$\begin{aligned} 2x_{92} + 3x_{93} + 4x_{94} - 4x_{N4} &\leq 0 \\ 2x_{11,2} + 3x_{11,3} - 4x_{11,4} - 4x_{N4} &\leq 0 \end{aligned}$$

Finally we consider the resource constraints:

$$\begin{aligned} x_{11} + x_{21} + x_{61} + x_{81} &\leq 2 \\ x_{32} + x_{62} + x_{72} + x_{82} &\leq 2 \\ x_{73} + x_{83} &\leq 2 \\ x_{10,1} &\leq 2 \\ x_{92} + x_{10,2} + x_{11,2} &\leq 2 \\ x_{43} + x_{93,2} + x_{10,3,2} + x_{11,3} &\leq 2 \\ x_{54} + x_{94,2} + x_{11,4} &\leq 2 \end{aligned}$$

Any set of start times satisfying these constraints is valid. For the sake of illustration, let us assume now that the heuristic algorithm would have given us a bound on the latency of 5 steps. We want to find out if the bound is tight. In this case, the larger mobility of the operations would lead to a larger set of constraints. We leave as an exercise to the reader to derive the constraints equation. We just remark that the uniqueness constraint on the start time of the sink vertex would be $x_{N4} + x_{N5} = 1$ and the objective function would be $min\ (4 \cdot x_{N4} + 5 \cdot x_{N5})$. Therefore, assuming that all the new constraint equations are satisfied, the optimum solution would imply $x_{N4} = 1$ and $x_{N5} = 0$, i.e. a schedule of four steps. □

So far we have considered the *minimum-latency scheduling problem under resource constraints*. The dual problem is the *minimum-resource scheduling under latency constraints*, that can be formalized as follows:

$$\begin{aligned} min \quad & \|\underline{a}\| \quad such\ that \\ \sum_l x_{il} &= 1 \quad i = 1, 2, \ldots, N \\ \sum_l x_{il} - \sum_l l \cdot x_{jl} - d_j &\geq 0 \quad i, j = 1, 2, \ldots, N\ s.t.(v_j, v_i) \in E \\ \sum_i j x_{ij} &\leq \lambda \quad j = 0, \quad \ldots, t_N \end{aligned}$$

where the last constraint equation bounds the latency instead of the resource usage.

Note the choice of the norm of vector $\underline{a}$ to be minimized relates again to slightly different goals, for example the sum of the resources. A slightly different formulation may incorporate weights for the resources.

Example 5.6. Let us consider again the sequencing graph of Figure 1, with the assumptions used in the previous example. Let us assume that the multiplier costs five times as much as an ALU in terms of area. We assume that an upper bound on the latency is $\lambda = 4$.

The uniqueness constraints on the start time of the operations and the sequency dependency constraints are the same of the previous example. The resource constraints are in terms of the unknown variables a_1 and a_2.

$$x_{11} + x_{21} + x_{61} + x_{81} \leq$$

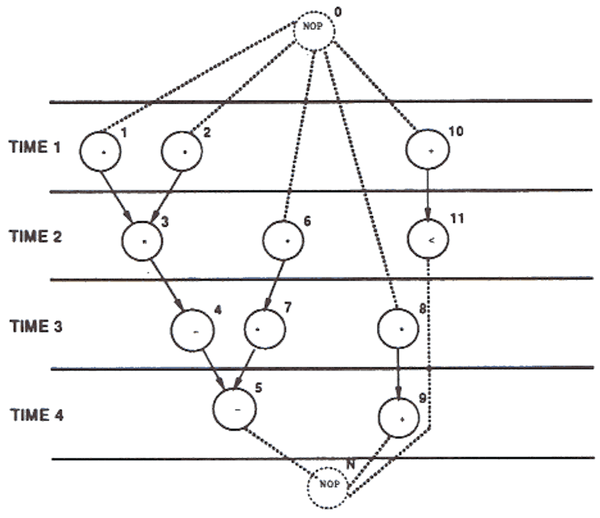

Figure 10: Scheduled sequencing graph under latency constraints that minimizes a weighted resource cost.

$$\begin{aligned}
x_{32} + x_{62} + x_{72} + x_{82} &\leq a_1 \\
x_{73} + x_{83} &\leq \\
x_{10,1} &\leq a_2 \\
x_{92} + x_{10,2} + x_{11,2} &\leq a_2 \\
x_{43} + x_{93,2} + x_{10,3,2} + x_{11,3} &\leq a_2 \\
x_{54} + x_{94,2} + x_{11,4} &\leq a_2
\end{aligned}$$

The objective function to minimize is $5 \cdot a_1 + 1 \cdot a_2$. A solution is shown in Figure 10. □

The ILP formulation of the scheduling problem is attractive for two reasons. First its solution is an optimum, i.e. it provides an exact solution to the scheduling problem. Second, it makes possible to use general purpose packages for solving the optimization problems. In some cases, a linear programming solution is sought first, and then transformed into an optimal integer solution by using a branch and bound algorithm. The number of variables X is $(n_{ops} + 1) \cdot L$. Gebotys *et alii* developed a set of tighter bounds that help reducing the number of relevant variables and constraints, and therefore enhanced the applicability of the approach [17]. Practical implementations of ILP schedulers have been shown to be efficient for medium scale examples, but to fail to solve problem with hundreds of binary variables or more.

5.2.2 List scheduling.

The minimum latency resource-constrained scheduling problem and the minimum resource latency-constrained one are known to be intractable. Therefore, heuristic algorithms have been developed. We consider in this Section a family of algorithms called *list scheduling* algorithms.

We consider first the minimum latency scheduling problem, under resource constraints.

The following algorithm is a framework for the minimum latency problem that is an extension of Hu's algorithm [21] to handle multiple operation types. It can be described as follows:

```
LIST_L( $G(V,E)$, $a$ )

    Set $i = 1$;
    repeat {
        for each resource type $k = 1, 2, \ldots n_{res}$ {
            Determine candidate operations U;
            Select $s_k \leq a_k$ vertices among the candidates $U$;
            Schedule the $s_k$ selected vertices in step $i$;

        $i = i + 1$

    until ($v_N$ is scheduled) ;
```

The candidate operations are those whose predecessors have already been scheduled early enough, so that the operation is completed. Namely: $\{U \subseteq V; v_j \in U$ when $t_k + d_k \leq t_j;\ v_k \in pred(v_j)\}$. The algorithm complexity is $O(n_{ops})$. It computes a schedule that satisfies the resource constraints by construction. However, the computed schedule may not have minimal length.

The list scheduling algorithms are classified according to the selection step. A priority list of the operations is used in choosing among the operations, based on some heuristic urgency measure. The algorithm is called greedy scheduling when the selection is random.

A common priority list is to label the vertices with length of their longest path to the sink vertices and to rank them in decreasing order. Therefore, most urgent operations are scheduled first. Note that when the operations have unit delay and when there is only one resource type (i.e. $n_{res} = 1$), then the algorithm is the same as Hu's and it yields an optimum solution for tree-like sequencing graphs.

Detailed timing constraints can be handled by list scheduling, by modifying the priority list to reflect the proximity of an unscheduled operation to a deadline.

Example 5.7.

Let us consider the sequencing graph of Figure 1. Let us assume that we have $a_1 = 3$ multipliers and $a_2 = 1$ ALU, performing addition, subtraction and comparisons. Let us assume that the multipliers take two cycles to execute and the ALU 1.

We consider first a list schedule, where the priority function is based on the length of the longest path to the sink vertex. Then the operations are scheduled as follows:

Time step	*Multiply operation*	*ALU operation*
1	$v_1 v_2 v_6$	v_{10}
2	$v_1 v_2 v_6$	v_{11}
3	$v_3 v_7 v_8$	-
4	$v_3 v_7 v_8$	-
5	-	v_4
6	-	v_9
7	-	v_5

It is possible to verify that the schedule has minimum latency.

Consider now a greedy scheduling approach, where operations v_2, v_6, v_8 are scheduled at the first step. Then, the schedule would require at least 8 steps, and it would not be optimal. □

List scheduling can also be applied to the latency-constrained minimum resource problem. In this case, the *slack* of an operation can be used to rank the operations, where the slack is the difference between the latest possible start time computed by an ALAP schedule and the current schedule step under consideration. The lower the slack, the higher the urgency in the list is. Operations with zero slack are always scheduled. The remaining ones are scheduled only if the number of required resources does not increase. Ties are broken using the urgency list.

```
LIST_R( $G(V,E), \lambda$ )
{
        Compute the earliest possible schedule by ASAP ( $G(V,E)$);
        Compute the latest possible schedule by ALAP ( $G(V,E), \lambda$);
        Set $i = 1$;
        repeat {
                Compute the slacks of the operations;
                for each resource type $k = 1,2,\ldots n_{res}$ {
                        Schedule at step $i$ the candidate operations with zero slack;
                        Schedule at step $i$ the candidate operations that do not require additional resources;

                $i = i + 1$;

        until ($v_N$ is scheduled) ;
}
```

Overall list scheduling algorithms have been widely used in synthesis systems. The experimental results have shown that the algorithm can be applied to large graphs. Solutions have been shown not to be much different in latency from the optimum ones, for those (small) examples whose optimum solutions are known.

5.2.3 Other heuristic scheduling algorithms.

Several other heuristic scheduling algorithms have been proposed. Some are derived from software design techniques. Example of these algorithms are *trace scheduling* and *percolation scheduling*. The latter has been used in different forms in various synthesis systems, even though some of its implementations were more restrictive in scope and power than the original algorithm. Percolation scheduling uses a transformational approach, that moves operations from one control step to another. This has to be contrasted to the list and force-directed scheduling approaches, that are constructive.

A commonly used heuristic scheduling algorithm is *force-directed* scheduling, that was proposed by Paulin and Knight [45]. It can be considered as an extension to the list scheduling approach, where operations are still scheduled for increasing time steps. The major contribution of the force-directed scheduling approach is that it considers a global function in the candidate selection process, namely the probability distribution of operation-type across the schedule. This function models the likelyhood that a resource is used at any given step. A uniform distribution means that an operation-type is evenly scattered in the schedule and it relates to a good measure of utilization of that resource. The objective in force-directed schedule is then to schedule the operations so that the distributions are as uniform as possible. To this aim, the priority of an operation is based on a measure, called *force*, of the operation concurrency. We refer the interested reader to reference [45] for details.

5.3 Scheduling algorithms for extended sequencing models.

5.3.1 Scheduling and operation chaining.

Combinational hardware resources have an intrinsic propagation delay that can be used to compute the execution delay, by dividing it by the clock cycle and rounding-up the result. By having computed the execution delays of the operations in terms of cycle times, the previous scheduling model and algorithms can be used. Unfortunately some inefficiencies may surface by using this approach.

Example 5.8. Assume that two operations in a sequence require 20ns to execute and that the target cycle time is 50ns. Then, the execution delay of both operations is 1 time unit, and the sequence of the two operations takes 2 time steps in the schedule. The operations could be instead be assigned to the same control step. □

Chaining is the task of combining more than one operation in a control step. Chaining can be performed before scheduling, and then the combined operations can be assigned to the same vertex in $G(V, E)$. Else, chaining can be performed concurrently with scheduling. The latter approach is followed when resource constraints have to be taken into account.

Let us consider the ILP model, and let us assume in this Section that the delays $D^P = \{d_i; i = 1.2. \ldots . n_{ops}\}$ are the propagation delays. Let us denote by F the set of vertex pairs (v_i, v_j) such that v_j is a successor of v_i, the sum of the delays of the vertices on some path between v_i and v_j exceeds the cycle time, and the sum of the delays from v_i to v_k does not, when v_k is any successor of v_i and a predecessor of v_j.

Then, to model the chaining problem within the ILP scheduling framework, we must adapt the sequencing dependency relations (5) to the following two:

$$\sum_l l \cdot x_{il} \geq \sum_l l \cdot x_{jl} + d_j \quad i,j = 1,2, \quad N \quad s.t.(v_j, v_i) \in E \qquad 3)$$

and

$$\sum_l l \cdot x_{il} \geq 1 + \sum_l l \cdot x_{jl} + d_j \quad i,j = 1,2,\ldots,N \quad s.t.(v_j, v_i) \in F \qquad (14)$$

With this modifications, the ILP model can be used to solve chaining in conjunction with either resource-constrained or latency-constrained scheduling. The ASAP, ALAP, list and force-directed scheduling algorithms can be extended to incorporate chaining in a straightforward way.

Example 5.9. Consider the sequencing graph of Figure 11 (a). Let us assume a cycle-time of 60ns. Let us also assume unconstrained resources. Then, an ASAP algorithm would assign to the first time step all those vertices whose predecessor are scheduled and that are heads of paths from the source, with weight less than 60. These vertices are represented in Figure 11 (b) above the top dark line. Then, the ASAP algorithm would do the assignment of the operations to step 2, by repeating the same computation after having eliminated the vertices

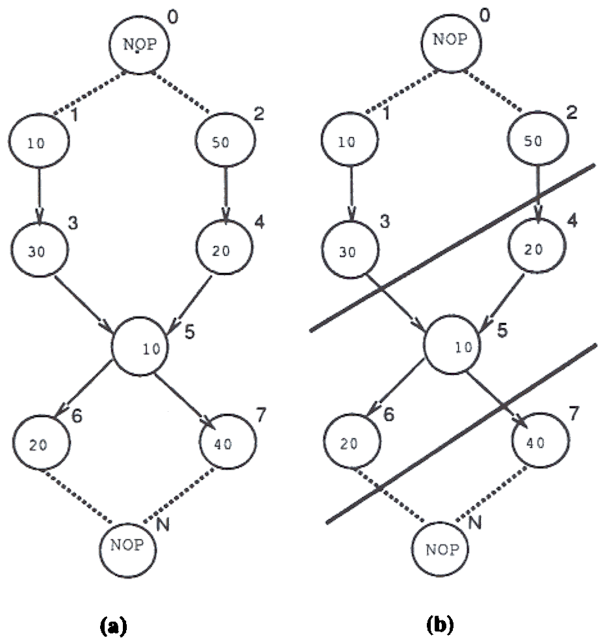

Figure 11: (a) Sequencing graph. The numbers inside the circles denote the propagation delay. (b) Scheduled sequencing graph. Required cycle time is 60.

scheduled at step 1. The operations assigned to step 2 are those in between the dark lines. The remaining one is assigned to step 3. □

It is worth-while to note that in the standard chaining problem the cycle-time is given. A related problem is to find the optimum operation chaining that minimizes the cycle-time, given a specified latency. Such a problem is equivalent to *retiming* [35], when resource and timing constraints are neglected.

5.3.2 Hierarchical scheduling.

We consider now sequencing graphs that are hierarchical and that support branching and iterative constructs. We assume that the delays are data-independent. We show how the algorithms can be extended to this general model.

Hierarchical scheduling can be solved by traversing the hierarchy bottom-up, and by solving the corresponding scheduling problem at each level of the hierarchy. A simplifying assumption that can be used to handle the hierarchy is that no resource sharing is done across different graph entities in the hierarchy, and that constraints apply within each graph entity. With this model, each graph can be scheduled independently. The latency and the resource usage of a schedule can be passed to the next level up in the hierarchy: the latency corresponds to the delay of the *calling vertex* and the resource usage to its types. (Obviously a calling vertex may have more than one type, because it relates to the resources that are needed in the execution of the called graph.) This model can be extended to handle the hierarchy induced by the branching and iterative constructs. In the latter case, the delay of an iteration vertex is the latency of the scheduled loop body times the number of iterations.

5.3.3 Scheduling graphs with alternative paths.

We assume in this Section that the sequencing graph contains alternative paths, related to branching structures. This extended sequencing graph can be obtained from our former model, by expanding the branching vertices, i.e. by replacing them by the called graphs. The mutual exclusive graphs representing the body of a branch give rise to alternative paths in the graph.

An exact formulation for the scheduling problem can be achieved by a slight modification of the ILP model. In particular, the resource constraints expressed by inequality (6), must reflect that operations in alternative branches can be scheduled in the same steps without affecting the resource constraints. More precisely, inequality (6) can be restated as follows:

$$\sum_{i,s.t.T(v_i)=k} \max_{e \in E(i)} \sum_{l=j-d_e+1}^{j} x_{el} \leq a_k \quad k = 1,2,\ldots,n_{res}; \quad j = 0,1,\ldots \quad N \qquad 15)$$

where the $E(i) \subseteq V$ is the subset of operations that includes v_i and all operations that are mutually exclusive to v_i. The above equation can be transformed into a linear constraint as follows:

$$\sum_{e \in E(i) s.t. T(v_e)=k} \sum_{l=j-d_e+1}^{j} x_{el} \leq a_k \quad k = 1,2, \quad ,n_{res}; \quad j = 0,1,\ldots,t_N; \quad i = 1,2 \qquad 16)$$

Camposano proposed a specialized scheduling algorithm that exploits the alternative paths in a sequencing graph, called *As Fast As Possible*, or AFAP [6]. In this approach, the sequencing graph represents alternative flows of operations, instead of parallel ones. Namely, fork vertices (i.e. vertices with more than one successor) represent branching conditions and not parallel streams. In addition, chaining is considered in conjunction with scheduling.

The AFAP algorithm schedules first each path independently. Since paths are alternative, then resource constraints apply only within each individual path. Resource constraints may limit the amount of operation chaining in any single step. Note that operations in a path are already ordered. Timing constraints can also be applied to each path. Camposano proposed a scheduling algorithm based on the search for the intersection of the constraints that delimit the time-step boundaries. Scheduling each path corresponds to determining the *cuts* in each path, where a cut is a subset of adjacent operations in a path such that any can define a schedule step boundary.

Once the paths are scheduled, they are merged together. Another graph is derived, where the cuts are represented by vertices and their intersection by edges. A clique in the graph corresponds to a subset of operations that can be started at some time step. Therefore a minimum clique covering of the graph provides a minimum latency solution. Since the graph has no particular property, the computational complexity of the approach is limited by solving the clique-covering problem, which is intractable, and by the fact that the number of paths may grow exponentially with the number of vertices in the sequencing graph. However, an implementation of this algorithm with an exact solution of the covering problem has given good results [6]. The formulation fits processor synthesis problems, where a large amount of alternative paths is related to executing different (alternative) instructions.

Example 5.10. Consider the example of Figure 11(a). Assume that the paths are alternative and that the only constraint is to meet a cycle-time of 60ns. There are four alternative paths, corresponding to operations with indices 1,3,5,6 ; 1,3,5,7 ; 2,4,5,6 and 2,4,5,7. By analyzing the first path, it is clear that a cut is required. It can be done after operation v_1 or after v_3 or after v_5. We indicate this cut by $c_1 = \{1,3,5\}$. A similar analysis of the second paths suggests that a cut is required after v_3 or v_5, i.e. $c_2 = \{3,5\}$. The other cuts are $c_3 = \{2\}$, $c_4 = \{2\}$

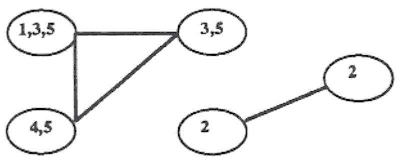

Figure 12: Intersection graph of the cuts with a clique cover. The clique cover corresponds to the intersection of the operations in the cuts and denote the last operation of a schedule step.

and $c_5 = \{4, 5\}$. Note that the last two cuts are related to the last path. Let us consider now the intersection graph corresponding to the cuts and shown in Figure 12. The clique cover indicates that operations v_2 and v_5 are the last before starting a new time step. This implies that the schedule has to conform to that of Figure 11(b). □

6 Data-path synthesis and optimization.

Data-path synthesis is a generic term that involves several tasks. At the high-level, data-path synthesis and optimization comprise *resource* and *register binding*. These task involve a functional model of a data-path in terms of a sequencing graph, a set of resources and registers, and a set of relations among them. At the circuit level, data-path connectivity synthesis involves the selection and the binding of *steering logic circuits* and *bus interfaces*. The interface signal of the data-path to the control circuit and external ports are also identified in this step. At the physical level, data-path synthesis consists of generating the corresponding layout.

Physical synthesis algorithms are not described here. We would like to mention that different approaches have been used, according to different design styles, namely *bus-oriented*, *macro-cell based*, or *array-based* data-paths. In the first case, a data-path generator constructs the data-path as a stack of bit-slices according to a predefined pattern. An example is the bus-oriented data-path synthesized by the SYCO compiler [12], that has an architecture similar to the M68000 processor. Macro-cell based data-paths are typical of DSP. Module generators are used to synthesize the resources, that need be placed and wired. This method has been used by the Cathedral-II compiler [15]. This approach is more flexible than using bus-oriented data-path synthesis with respect to the choice of a resource set, especially when application-specific resources are needed (e.g. arithmetic operators with non-standard word lengths). Unfortunately, this style often leads to a less efficient wiring distribution. Eventually, in the case of array-based data-paths, logic and physical synthesis techniques are applied to the data-path. Thus, the data-path is treated no differently than other portions of the design. In general, bit-sliced data-paths consume less area and perform better than data-paths designed in a macro-cell based or array-based style. The difference in performance may be small though, as compared to manual design, when data-path optimization is used.

We concentrate in the sequel on binding of resources and registers on scheduled sequencing graphs. A *resource sharing* is the assignment of a resource to more than one operation. The primary goal of resource sharing is to reduce the size of a circuit, by allowing multiple non-concurrent operations to share the same hardware operator. Resource sharing is often mandatory to meet specified upper bounds on the circuit area (or resource usage).

Resource binding is the explicit definition of a mapping between the operations and the resources. A binding may imply that some resources are shared. Resource binding (or partial binding) may be an original circuit specification and thus some sharing may be defined explicitly in the hardware description. Resource usage constraints may infer implicitly some resource sharing, even though they may not imply a particular binding.

In general, the overall area and performance depend on the total number of resource instances and registers, the steering logic circuits (e.g. multiplexers) and the wiring. Precise models take all these factors into account. In some cases, as in the case of *resource-dominated* circuits, the models can be simplified and made dependent only upon the total number of resources and registers. As a consequence, circuit performance is not affected by resource binding. Even though this assumption may seem crude, it is often viable for some classes of circuits, such as DSPs, that rely on several instances of few, well-characterized resources and storage elements. In this Section, ve consider sharing and binding for resource-dominated circuits only. We refer the reader to reference [30] for the general case.

6.1 Sharing and binding.

We call *resources* those hardware operators that are explicitly modeled in a sequencing graph. They include the functional resources and those interface resources, such as I/Os, that are defined explicitly in the sequencing graph model. We refer to *registers* as to those used to store the intermediate values of the variables. Note that registers are implied, but not represented, by the sequencing graphs.

Two (or more) operations may be bound to the same resource if they are not concurrent and they have the same type. A necessary and sufficient condition for non-concurrency is that the operations are scheduled in different time-steps or if they are alternative, i.e. they are part of different bodies of a branching construct. Two operations are said to be *compatible* when this condition is met and when they have the same type. Therefore, an analysis of the sequencing graph is sufficient to determine the compatibility of two or more operations for sharing. We postpone this analysis to the following two Sections and we concentrate now on the the compatibility issue.

Definition 6.1 *The resource compatibility graph $G_+(V, E)$ is a graph whose vertex set $V = \{v_i, i = 1, 2, \ldots, n_{ops}\}$ is in one to one correspondence with the operations and whose edge set $E = \{\{v_i, v_j\}\ i, j = 1, 2, \ldots\ n_{ops}\}$ denotes the compatible operations pairs.*

A group of mutually compatible operations corresponds to a subset of vertices that are all mutually connected by edges, i.e. to a *clique*. Therefore a *maximal* set of mutually compatible operations is represented by a *maximal clique* in the compatibility graph.

An optimal resource sharing is one that minimizes the number of required resource instances. Since we can associate a resource instance to each clique, than the problem is equivalent to finding the minimum number of cliques that cover the graph, i.e. that implement all the operations. Note that the unweighted clique covering and clique partitioning problems are equivalent, because a partition is a cover and a cover identifies partitions with the same cardinality.

Example 6.1. Let us consider as an example the scheduled sequencing graph of Figure 10. We assume again that there are two resource types: a multiplier and an ALU, that performs addition, subtraction and comparison. The compatibility graph is shown in Figure 13. Examples of compatible operations are $\{v_1, v_3\}$ and $\{v_4, v_5\}$ among others. Examples of maximal cliques are the subgraphs induced by $\{v_1, v_3, v_7\}$, $\{v_2, v_6, v_8\}$ and $\{v_4, v_5, v_{10}, v_{11}\}$. These cliques, in addition to $\{v_9\}$ cover the graph. Four resources are needed, corresponding to two multipliers and two ALUs. □

An alternative way of looking at the problem is to consider the *conflict* between operation pairs. Two operations have a conflict when they are not compatible. Conflicts can be represented by *conflict graphs*.

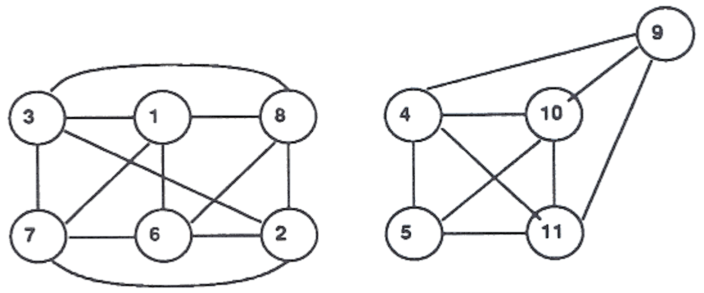

Figure 13: Compatibility graph.

Definition 6.2 *The resource conflict graph $G_-(V,E)$ is a graph whose vertex set $V = \{v_i, i = 1,2,\ldots,n_{ops}\}$ is in one to one correspondence with the operations and whose edge set $E = \{\{v_i, v_j\}\ i,j = 1,2,\ldots,n_{ops}\}$ denotes the conflicting operations pairs.*

It is obvious that the conflict graph is the *complement* of the compatibility graph. A set of mutually compatible operations corresponds to a subset of vertices that are not connected by edges, also called *independent set* of $G_-(V,E)$. A proper vertex coloring of the conflict graph provides a solution to the sharing problem: each color corresponds to a resource instance. An optimum resource sharing corresponds to a vertex coloring with a minimum number of colors.

The *clique partitioning* and *vertex coloring* problems have been studied extensively. Both problems are intractable for general graphs, and exact and heuristic solution methods have been proposed. According to specific circuit type under consideration, the compatibility graph can be sparser than the conflict graph (or *vice versa*). In this case, clique partitioning (or vertex coloring) may be easier to solve.

In some particular cases, it is possible to exploit the structure of the sequencing graph to derive compatibility and conflict graphs with special properties, that make the partitioning and coloring tractable. This will be considered in the following Section.

6.2 Resource sharing in non-hierarchical sequencing graphs.

A flat sequencing graph is acyclic and polar. Each source to sink path represents a parallel stream of operations. We denote by $T = \{t_i\ ;\ i = 1,2,\ldots,n_{ops}\}$ the *start time* for the operations and by $D^E = \{d_i\ ;\ i = 1,2,\ldots,n_{ops}\}$ the set of *execution delays*. Data-dependent delays are not considered here because the sequencing graph is assumed to be scheduled. We refer the interested reader to reference [30] for the general case. The type of an operation is represented by $\mathcal{T}(v_i)$; $i = 1,2,\ldots,n_{ops}$.

Two operations are then compatible if they have the same type and if they are not concurrent. Therefore, the compatibility graph $G_+(V,E)$ is described by the following set of edges: $E = \{\{v_i, v_j\} |\ \mathcal{T}(v_i) = \mathcal{T}(v_j)\ and\ ((t_i + d_i \geq t_j)\ or\ (t_j + d_j \geq t_i)),\ i,j = 1,2,\ldots,n_{ops}\}$. Such a graph can be constructed by traversing the sequencing graph in $O(|V|^2)$ time. This graph is a *comparability* graph because it has a transitive orientation property. Indeed, a corresponding directed graph could be derived by assigning an orientation to the edges compatible with the relations $((t_i + d_i \geq t_j)\ or\ (t_j + d_j \geq t_i)),\ i,j = 1,2,\ldots,n_{ops}\}$ that are *transitive*.

The search for a minimum clique cover of a comparability graph can be achieved in polynomial time, by transforming it into a minimum-flow problem [18].

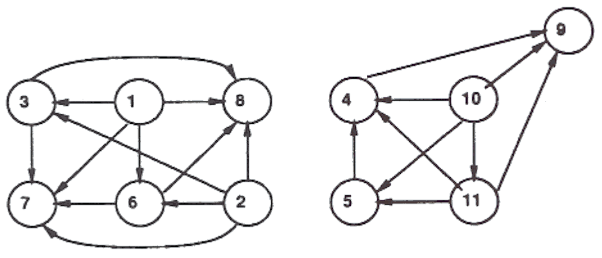

Figure 14: Transitive orientation of the compatibility graph.

Example 6.2. All operations have unit execution delay. Let us consider operation v_1, with $t_1 = 1$. Now $T(v_1) = \textit{multiplier}$. Then, all the operations whose type is a multiplier and whose start time is larger than or equal to 2 are compatible with v_1. (Obviously, no operation can be compatible by having a start time less than or equal to zero). Such operations are $\{v_3, v_6, v_7, v_8\}$. The corresponding vertices are incident to edges that stem from v_1. The compatibility graph can be constructed by visiting each operation and checking for others with the same type with non-overlapping execution intervals.

Note that a directed graph could be constructed, having the compatibility graph as underlying graph. The orientation is determined by comparing the start times. In this case, it would have the edges $\{(v_1, v_3), (v_1, v_6), (v_1, v_7), (v_1, v_8)\}$ among others. Note also that the relations $\{(v_1, v_3), (v_3, v_7)\}$ imply $\{(v_1, v_7)\}$, because ordering is a transitive relation. Hence the compatibility graph is a comparability graph. The transitive orientation of the compatibility graph is shown in Figure 14. □

Let us consider now the conflict graph. Two operations conflict if their type is different or if their execution overlaps. Let us assume first that all operations have the same type and consider the execution intervals for each operation $\{[t_i, t_i + d_i - 1]\ i = 1, 2, \ldots, n_{ops}\}$. The conflict graph is a graph whose edge set denotes an intersection among intervals, hence it is an *interval graph*.

The search for a minimum coloring of an interval graph can be achieved in polynomial time. A few algorithms can be used, including the *left-edge* algorithm [47]. When operations have different types, it is more convenient to color the interval subgraphs induced by the operations of each type.

6.3 Resource sharing in hierarchical sequencing graphs.

Let us now consider hierarchical sequencing graphs. A simplistic approach to resource sharing is to perform it independently within each sequencing graph entity. Such an approach is overly restrictive, because it would not allow sharing resources in different entities. Therefore we consider here resource sharing across the hierarchy levels.

Let us first restrict our attention to sequencing graphs where the hierarchy is induced by *model calls*. We need to distinguish here between *single* and *multiple* model calls. In both cases, model calls make the sequencing graph representation modular. Moreover, in the latter case, model calls express also the sharing of the application specific resource corresponding to the model.

We consider single model calls first. The concept of compatibility can be extended to hierarchical compatibility. Two non-concurrent complex vertices imply the compatibility of the vertices with the same type in the graph entities corresponding to the called models. Unfortunately, concurrency of complex operations does not necessarily imply conflicts of the operations in the called models. Therefore this model does not fully capture the compatibility property of the operations.

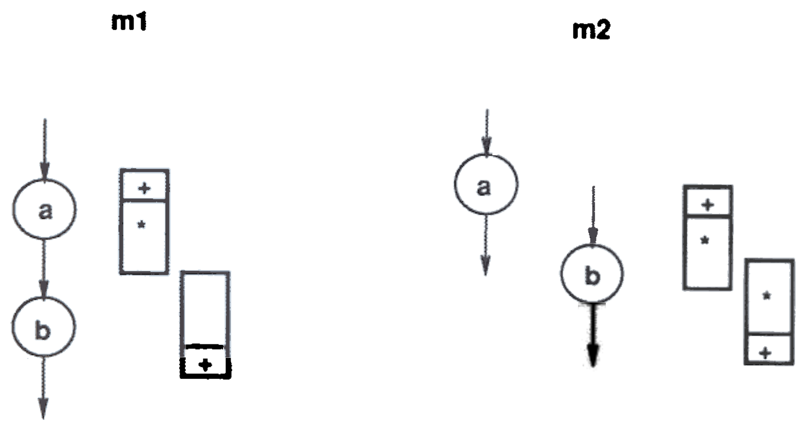

Figure 15: Hierarchical conflicts and compatibility.

Example 6.3. Consider a model a that has two operations: an addition followed by a multiplication. Consider also a model b that has two operations: a multiplication followed by an addition. Assume that the addition has a unit delay and the multiplication two unit delays. When a model $m1$ has a call to model a followed by a call to model b, then a and b are not concurrent and the corresponding additions and multiplications are compatible.

When another model $m2$ has two calls to a and b that partially overlap, say with start times $t_a = 1$ and $t_b = 2$, then by the above argument the additions and the multiplications are not compatible. Indeed the multiplications are not compatible while the additions are! Both situations are shown in Figure 15. □

Therefore the appropriate way of computing the compatibility of operations across different levels of the hierarchy is to expand the hierarchy itself, by replacing the complex vertices by the graphs of the corresponding models. Such an expansion can be done explicitly, or implicitly by computing the execution intervals of each operation with respect to the source operation of the top model in the hierarchy. As a result, a complete compatibility graph can be generated, with the property of being a comparability graph. Similar considerations apply to the conflict graph computation.

Let us consider now multiple model calls, that already represent a resource sharing. Such multiple model calls can be part of the circuit specification, that embeds the notion of application-specific resource sharing. Alternatively, the multiple model call can model a binding derived by applying resource sharing while traversing the hierarchy top-down. We question the possibility of sharing those resources, that are part of the shared model, with other compatible resources in the overall sequencing graph model.

Example 6.4. Consider a model a that has two operations: an addition followed by a multiplication. Assume that model $m3$ has two calls to model a, that are not concurrent, scheduled at times 1 and 5 respectively. Assume also that model a has three other multiplication operations. We question the sharing of the multipliers across the hierarchy. A sequencing graph fragment (related to $m3$), the execution intervals and the conflict graph for the multiplier are shown in Figure 16. Note that the double call to a results in two non-contiguous intervals for the multiplier in a. As a result, the conflict graph is not an intersection among intervals, and therefore not an interval graph. It is not even a chordal one, as shown in the picture. □

Also in this case, to model completely the compatibility of the operations inside the called models, the hierarchy must be expanded. Note though that multiple model calls represent now shared models, and therefore their internal

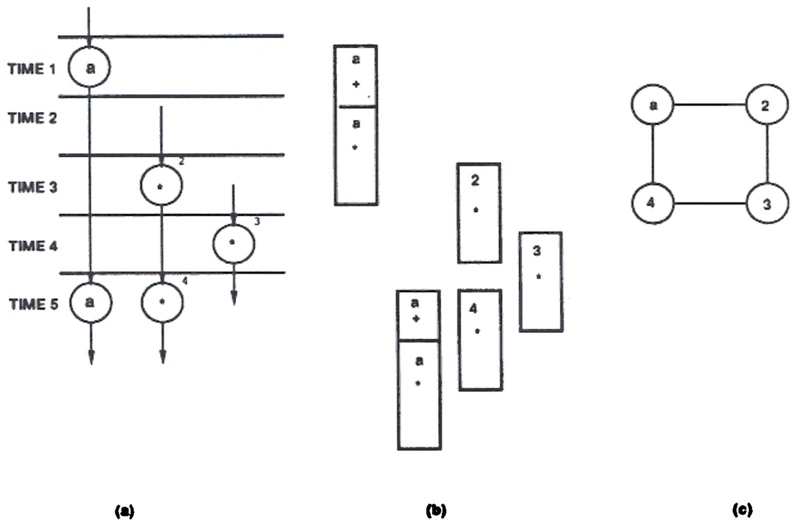

Figure 16: Hierarchical conflicts. (a) sequencing graph segments. (b) execution intervals. (c) non-chordal conflict graph.

resources are implicitly shared. While the computation of the compatibility and conflict graphs is still straightforward, such graphs are no longer necessarily comparability and interval graphs. Therefore their clique partitioning and vertex coloring are now intractable problem, and heuristic algorithms must be used.

Iterative constructs, that can be unrolled, can also be expanded. Similar considerations apply. Note that each resource in a loop corresponds to many resource instances when the loop is unrolled. These instances are shared among each other and can be possibly shared with other compatible ones in the overall sequencing graph model. Note that model call inside a loop body becomes a multiple call when the loop body is unrolled.

Let us consider now the *branching* constructs. When considering operation pairs in two alternative branching bodies, their compatibility corresponds to having the same type. A complication arises in modeling the compatibility across the hierarchy, i.e. checking for compatibility of the operations in a sequencing graph entity and in those modeling the branching bodies. Expanding the branching hierarchy yields graphs with alternative paths, that have different properties than the extended data-flow graphs. Such graphs can still be traversed in $O(|V|^2)$ time to compute the compatibility (or conflict) graphs. In this case, two operations are compatible if they have the same type and they are either non-concurrent or alternative. Now the compatibility graph is not necessarily a comparability graph and the conflict graph may not be an interval graph and not even a chordal one.

Example 6.5. Consider the sequencing graph of Figure 17 (a). We assume that all the operations take 2 time units to execute and that the start times are the following: $t_a = 1; t_b = 3; t_c = t_d = 2$. The intervals are shown in Figure 17 (b) and the conflict graph in figure Figure 17 (c). Note that the alternative nature of operations c and d makes them compatible and prevent a chord $\{v_c, v_d\}$ to be present in the conflict graph. □

6.4 Register sharing.

We consider in this Section those registers that hold the values of temporary variables. Each variable has a *lifetime* that is the interval from its *birth* to its *death*, where the former is the time in which the value is generated as an

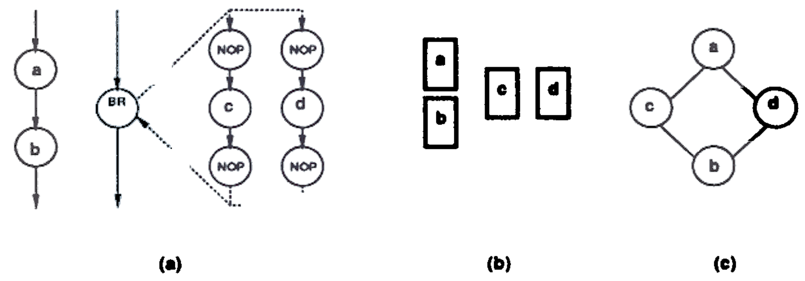

Figure 17: Conditional execution. (a) sequencing graph segments. (b) execution intervals. (c) non-chordal conflict graph.

output of an operation and the latter is the latest time in which the variable is referenced as an input to an operation. We assume that those variables with multiple assignments within one model are aliased, so that each variable has a single lifetime interval in the frame of reference corresponding to sequencing graph entity where it is referenced. Note that the lifetimes can be data-dependent, for example due to branching and iterative constructs.

Whereas an implementation that associates a register to each variable suffices, it is obviously inefficient. Indeed variables that are alive in different intervals, or under alternative conditions, can share the same register. Such variables are called *compatible.*

The *register compatibility* and *conflict* graphs are defined analogously to the resource compatibility and conflict graphs. The problem of minimizing the number of registers can be cast in a minimum clique partitioning problem of the compatibility graph or into a minimum coloring problem of the conflict graph. We consider now how these graphs are generated and their properties.

Let us consider first non-hierarchical sequencing graphs. In this model, a conflict between two variables corresponds to a life-time overlap. Since in this model the variable lifetimes are intervals, then the conflict graph is an interval graph and its complement is a comparability graph. Therefore, optimum register sharing can be computed in polynomial time, for example by optimum coloring using the left-edge algorithm.

Let us now consider sequencing models of iterative bodies. In this case, some variables are alive across the iteration boundary. For example, the loop counter variable. The cyclicity of the lifetimes is modeled accurately by circular graphs, that represent the intersection of arcs on a circle. The register sharing problem can then be cast as a minimum coloring of a circular graph, that unfortunately is intractable. Branch-and-bound or heuristic algorithms can be used. Stok [50] has shown that this problem can be transformed into a multi-commodity flow problem, and then solved by a primal algorithm.

The register sharing problem can be extended to cope with hierarchical models. The compatibility and conflict graphs can be derived by applying similar considerations to hierarchical resource sharing. In particular, interval conflict graphs can be derived from hierarchical models with only single *model calls*, by considering the variable lifetimes with reference to the start time of the sequencing graph entity in the top model in the hierarchy. For general graphs, compatibility and conflict graphs can still be derived by traversing the hierarchy and comparing the variable lifetimes. In the general case the compatibility and conflict graphs are not comparability and interval graphs respectively, and therefore the corresponding optimal register sharing problem is intractable. Springer and Thomas [49] have shown that polynomial-time colorable conflict graphs can be achieved by enforcing some restrictions on the model calls and on the branch types.

6.5 Other binding and sharing problems.

Other binding and sharing problems stem from the use of particular circuits, such as memory arrays, busses, and interfaces.

Some design styles use multi-port memories to store the values of the variables. Such memories are also referred to as general-purpose registers (GPRs), common to RISC architectures. Let us assume the memory has a ports for either *read* and *write* requiring one cycle per access. A binding problem consists of computing the minimum number of ports a required to access as many variables as needed. Balakrishnan *et alii* [2] considered the dual problem. They assumed a fixed number of ports and they maximized the number of variables to be stored in the multi-port memory, subject to the port limitation. Both problems can be formulated as ILPs.

Busses act as transfer resources that feed data to functional resources. The operation of writing a specific bus can be modeled explicitly as a vertex in the sequencing graph model. In this case, the compatible (or conflicting) data transfers may be modeled by compatibility (or conflict) graphs, as in the case of functional resources. Alternatively, busses may not be explicitly described in the sequencing graph model. Their (optimal) usage can be then derived by exploiting the data transfers. Since busses have no memory, we consider only the transfers of data within each schedule step (or across two adjacent schedule steps, when we assume that the bus transfer is interleaved with the computation). Two problems then arise. First, to find the minimum number of busses to accommodate all (or part of) the data transfers. Second, find the maximum number of data transfers that can be done through a given number of busses. Both problems can be modeled again by ILPs.

7 Control synthesis.

We consider in this Section the synthesis of the control units. From a circuit implementation point of view, we can classify the control-unit model as microcode-based or hard-wired. The former implementation style stores the control information into a read-only memory (ROM) array, while the latter uses a hard-wired sequential circuit consisting of an interconnection of a combinational circuit and registers. From a logic stand-point, synchronous implementation of control can be modeled as a *finite-state machine* . Both implementation styles can be modeled as such, because a read-only memory and a synchronous counter behave as a finite automaton as well as an interconnection of combinational logic gates and synchronous registers.

The interface between the data-path and the control circuit is provided by the signals that enable the registers, and that control the steering circuits (i.e. multiplexers and busses). Sequential resources require an *activation* (and sometimes a *reset*) signal. Data-dependent operations must provide a *completion* signals. The ensemble of these control points are identified during the synthesis of the data-path. In addition, the control unit requires some *condition* signals from the data-path, that are needed to evaluate the clauses of some branching and iterative constructs.

Example 7.1. Figure 18 shows an example of the interconnection between the data-path and control. The data-path provides signals to the control unit related to the execution of alternative control flows, such as

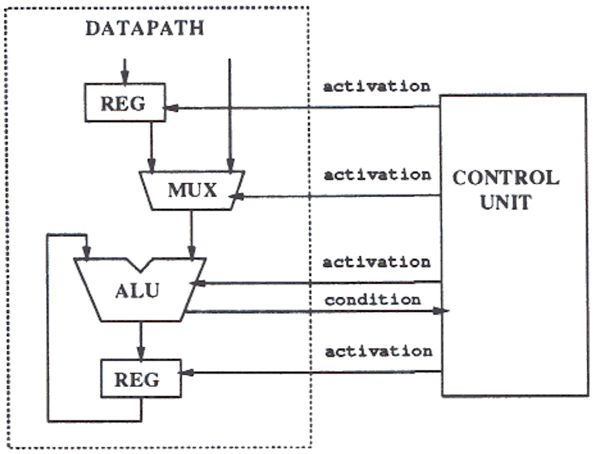

Figure 18: Example of interface signals between data-path and control.

the overflow signal from the ALU. The control unit provides the *activation* signals to the ALU that select the appropriate operation, as well as the *activation* signals that select the multiplexers and enables the registers. □

Control synthesis for non-hierarchical graphs with data-independent delays requires the specification of the *activation* signals only. Hierarchical graphs, modeling branching and iteration, must also take into account the *condition* signals. Control units for unbounded-delay operations require handling the *completion* signals, as well as the others. Therefore, we shall analyze increasingly complex models for control.

7.1 Control synthesis for non-hierarchical sequencing graphs.

We consider in this Section the synthesis of the control unit for a scheduled sequencing graph that is bound to the resources. The knowledge of a schedule allows us to determine the time frame of the operations. The binding determines the control points of the data-path. We assume that the sequencing graph is not hierarchical (i.e. all vertices are *simple*) and that all the operations have data-independent delays. We assume that each operation can be started by an *activation* signal, that triggers the start of of the functional resource and/or steers data into (and/or out from) a functional, memory, or interface resource. We assume that there are n_{act} *activation* signals to control.

Let us consider first the microcode-based implementation style. A microcoded implementation can be achieved by using a memory that has as many words as the latency t_N. Each word is in one-to-one correspondence with a schedule step. Therefore the ROM must have as many address bits as $n_{bit} = \lceil log_2 \ t_N \rceil$. A synchronous counter with n_{bit} bits is used to address the ROM. The counter has a reset signal, that clears the counter, so that it can address the first word in memory, corresponding to the first operations to be executed. When the sequencing graph models a set of operations that must be iterated, then the last word of the schedule clears the counter. The counter runs on the system clock. The only external control signal provided by the environment is the counter *reset*. By raising that signal, the overall circuit halts and resets. By lowering it, it starts execution from the first operation.

Let us consider now hard-wired control implementations. The synthesis of a Moore-type *finite-state machine* from a scheduled sequencing graph is straightforward. Indeed, such a machine has as many states as the latency t_N (i.e. schedule length), and the state set $S = \{s_i \ ; \ i = 1, 2, \dots, t_N\}$ is in one to one correspondence with the schedule steps. State transitions are unconditional and only among state pairs $(s_i, s_{i+1}) \ ; \ i = 1, 2, \dots, (t_N - 1)$.

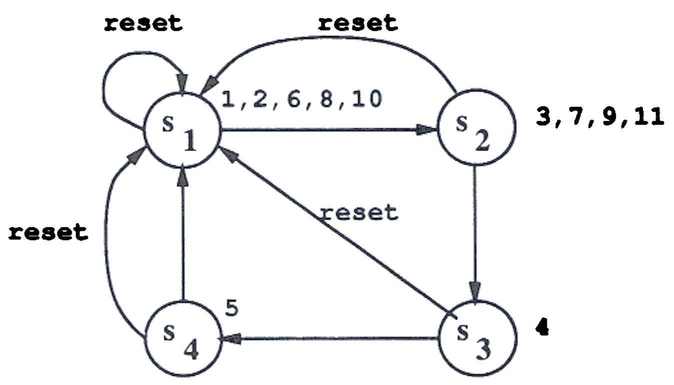

Figure 19: Example of state diagram for hard-wired control.

An unconditional transition (s_N, s_1) provides for repetitive execution of the schedule. Conditional transitions into s_1 from all the other states, controlled by a *reset* signal, provide the start and reset capability. The output function of the *finite-state machine* in each state s_i ; $i = 1,2,\ldots,t_N$ activates those operations whose start time is t_i. More specifically, the *activation* signal for the control point $k, k = 1,2,\ldots,n_{act}$ in state s_i ; $i = 1,2,\ldots,t_N$ is δ_{i,t_k}, where $\delta_{i,j}$ denotes a Kronecker delta function. A hard-wired control unit can be obtained by synthesizing the *finite-state machine* model using standard techniques [12] and in particular by encoding the states and by implementing the combinational logic in the appropriate style (e.g. sparse logic, PLA, *et cetera*). It is straightforward that a binary encoding of the *finite-state machine* states and a completely-specified two-level combinational logic representation correspond to the microcode-based implementation specification. Conversely, a microcode-based implementation can be transformed into hard-wired control by re-encoding the states and by casting the combinational logic function stored in the ROM into the desired circuit style.

Example 7.2. Consider again the scheduled sequencing graph of Figure 5. The state transition diagram of the *finite-state machine* implementing a hard-wired control unit is shown in Figure 19. The numbers by the vertices of the diagram are the reference to the *activation* signals. □

7.2 Control synthesis for hierarchical sequencing graphs.

Hierarchical sequencing graphs represent model calls, branching and iteration through the hierarchy. In this Section, we assume that the graphs have bounded latency, and therefore each vertex has a known, fixed execution delay.

Let us consider first model calls and their control implementations. We can assume that every sequencing graph entity in the hierarchy has a corresponding local control unit. Since a sequencing graph entity represents a model that may be shared, we need to make the following assumption. Each control unit has its own *activation* signal, that controls the stepping of the counter or the *finite-state machine* transitions and that gates the *activation* signals to the resources. Therefore, asserting the *activation* signal for a control unit block corresponds to executing the related operations. Lowering the *activation* signal corresponds to halting all operations. We recall that in our previous models, the controller resets itself after having executed the last operation. An additional *reset* signal may be provided to each control unit in the hierarchy.

The hierarchical control implementation can be achieved as follows. The execution of a complex vertex, corresponding to a model call, is translated to sending an *activation* signal to the corresponding controller. That

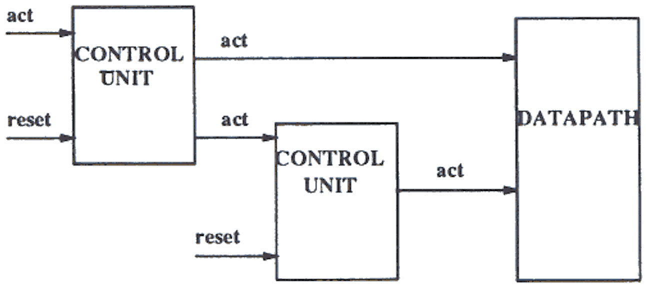

Figure 20: Example of interconnecting a hierarchical control structure.

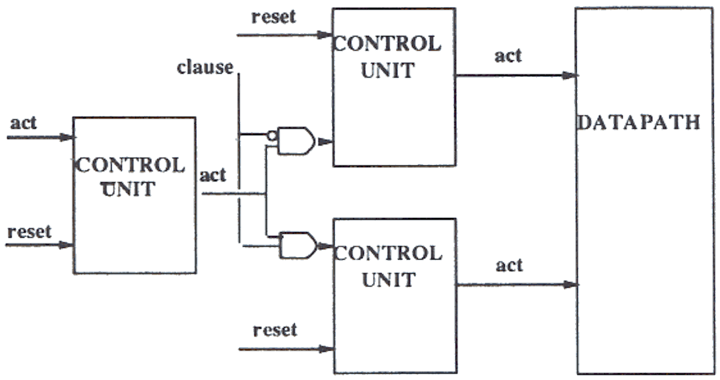

Figure 21: Example of interconnecting a hierarchical control structure.

signal is asserted for the duration of execution of the called model, i.e. as long as its local latency. Note that the controller of the calling model continues its execution, because the model call is in general concurrent with other operations. An example is shown in Figure 20.

The interconnection of the local control-unit blocks corresponding to the different sequencing graph entities in the hierarchy can be done regardless of the implementation style, as long as the *activation* signal is provided. The *activation* signal of the root model can be used to start the hardware. Alternatively, it can always be asserted and the circuit can be started by pulsing the reset line. Note that call to (and return from) a model does not require an additional control step with this scheme.

Let us consider now branching operations. A branch is represented in the hierarchical sequencing graph model by a selective model call, controlled by the branching clause. Therefore, a straightforward implementation can be achieved by activating the control-unit blocks corresponding to a body of a branch by the conjunction of the *activation* signal with the branch clause value, as shown in Figure 21. For this control scheme to be correct, we must assume that the branching clause does not change during the execution of the branch itself. Therefore, the value of the clause may have to be temporarily stored.

The control for an iteration complex vertex can be done in a similar way. The loop body can be seen as a model call that is repeated a finite and known number of times. Since we already assume that each control-unit block resets itself when all operations have finished execution, it suffices to assert the *activation* signal for the loop body controller as long as the iteration has to last. Recall that the latency of an iteration complex vertex is the product of the loop body latency times the number of execution. This number, which in this case is known at synthesis time, is the duration of the *activation* signal.

7.3 Control synthesis for unbounded-latency sequencing graphs.

Unbounded-latency sequencing graph contain unbounded-delay operations, that provide *completion* signals to notify the end of execution. We will assume that the *completion* signal is raised during the last cycle of execution of an operation, so that no control step is wasted in detecting a completion and starting the successor operations. Similarly, the control-unit of an unbounded-latency graph is assumed to provide its own *completion* signal, to denote the end of execution of all the operations. This *completion* signal is used when composing control-unit blocks to form a more complex controller, as in the case of hierarchical graphs.

There are three approaches to synthesize a control unit for unbounded-latency graphs. The first one is the *clustering* method, that clusters the graph into bounded-latency subgraphs. The number of clusters depends on the number of unbounded-delay operations. The method is efficient (in terms of control unit area) when this number is small. Control implementations can be microcode-based or hard-wired. The second approach, called *adaptive* control synthesis, is reminiscent of some control synthesis techniques for self-timed circuits. It leads to a hard-wired implementation and it is efficient when the number of unbounded-delay operations is high. The third method is based on *relative scheduling*. We describe here the first method only. We refer the interested reader to reference [30] for the others.

The clustering method consists of extracting bounded-latency subgraphs, whose control can be synthesized as shown in the previous Sections. Consider the unbounded-delay vertices in the graph one at a time, in a sequence compatible with the partial order represented by the graph itself. Let $S \subset V$ be the subset of vertices that are not unbounded delay vertices nor are their successors. Then the subgraph induced by S can be made polar, by adding a sink vertex representing a No-Operation and edges from the vertices in S with no successors to the sink. Then this subgraph can be scheduled and its control unit can be generated with a microcoded or hard-wired style. The vertices S can be then deleted from the graph and the unbounded-delay vertex under consideration replaced by a No-Operation, that is now the source vertex of the subgraph induced by the remaining vertices.

A synchronizer is added to the control unit in correspondence to the unbounded-delay vertex previously under consideration. The synchronizer is a control primitive, that can be implemented by a simple *finite-state machine* . The synchronizer takes as input the *completion* signal of the controller of the subgraph just extracted and the *completion* signal of the unbounded-delay operation itself. The synchronizer issues an *activation* signal to the controller of the subsequent operations. The synchronizer memorizes the arrival of both completion signals into two independent states. The *activation* signal is asserted either in coincidence of both completion signals or when one completion signal is received and the *finite-state machine* is in the state that memorizes the arrival of the other one at some previous time step.

Example 7.3. Consider the graph of Figure 9. The set S is equal to $\{v_1, v_2\}$. The subgraph induced by S consists of vertices $\{v_1, v_2\}$. The subgraph can then be scheduled and its control unit built. After deleting $\{v_1, v_2, v_a\}$ the remaining cluster has only vertex v_3, and its schedule and control-block can be easily synthesized. The overall hard-wired control implementation is described by a state transition diagram in Figure 22. The shaded area on the left controls operations of the cluster $\{v_1, v_2\}$. The shaded area on the right is a synchronizer circuit, that operates as follows. Its reset state is s_a. A transition $s_a \rightarrow s_b$ is caused by the completion of the controller of the first cluster, while a transition $s_a \rightarrow s_c$ is caused by the completion of the unbounded-delay operation.

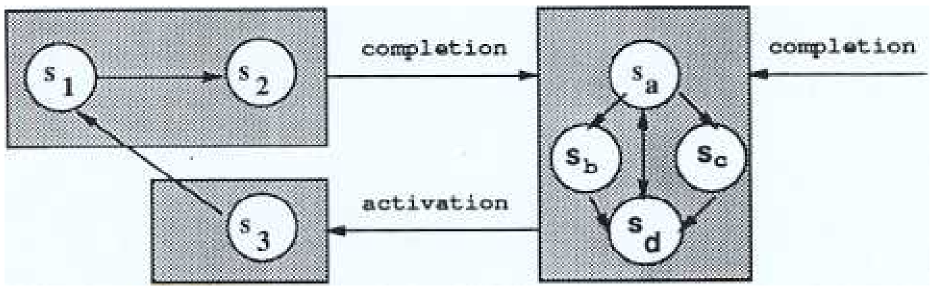

Figure 22: State transition diagram for a sequencing graph with two clusters and a synchronizer.

When the synchronizer is in s_a (or in s_c) and the the completion of the unbounded-delay operation (or of the controller of the first cluster) is detected, the synchronizer issues the *activation* signal and goes to state s_d. If the two *completion* signals are simultaneous, there is a state transition to s_d and the *activation* is asserted. □

8 Synthesis of pipelined circuits.

Pipelining is a common technique to enhance the circuit performance. In a pipeline implementation, the circuit is partitioned into a linear array of *stages*, each concurrently executing a task on a different set of data and feeding its results to the following stage. Pipelining has been applied to general purpose as well as signal/image processors. In the former case, pipeline design is complicated because it must be efficient while running on different instruction streams. Pipelined DSP design may be simpler, because often the processor executes a fixed algorithm.

At present, synthesis techniques for pipelined circuits are still in their infancy. In particular, synthesis techniques for data-paths have been proposed, under some limiting assumptions, such as neglecting pipeline stalling, stage bypasses and variable data rates. As a result, present synthesis techniques are of interest to the DSP designer community and are still immature for processor design. Therefore we consider in this Section only simple pipelined circuits, that can be modeled by pipelined sequencing graphs, as described in Section 2.2.

We recall that in a *pipelined sequencing graph* the source vertex is fired at a constant rate, called *throughput*. The inverse of the rate, i.e. the time separation between two successive firing of the source vertex, normalized to the *cycle-time*, is called *data introduction interval* (or DII). The data introduction interval is smaller than, or equal to, the *latency*.

Let us assume that the data introduction interval is a proper fraction of the latency. Then, at any given time, there are multiple instances of the circuit behavior (i.e. partial order of tasks) executing concurrently. They are equal to the quotient latency/DII, that corresponds also to the number of *stages* in the pipeline.

Example 8.1. Figure 23 shows two instances of the sequencing graph, representing a functionally pipelined circuit with $DII = 2$. If we assume that the operations have unit execution delays and that the number of resources is not constrained, then the sequencing graph can still be scheduled in 4 steps, i.e. the latency is 4. However, input and output data will be requested and made available at every other cycle. Therefore the throughput has doubled. □

The design evaluation space for pipelined circuits can be characterized by four parameters: the *throughput*, the *latency*, the *cycle-time* and the *area*. Structural synthesis of a pipelined circuit involves a multi-criteria optimization

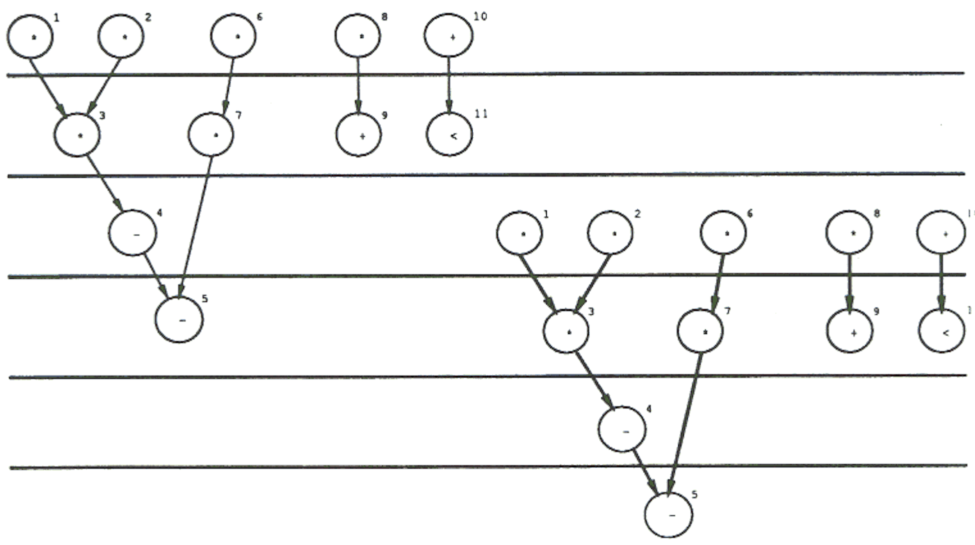
Figure 23: Two instances of a sequencing graph representing a functionally pipelined implementation. (The source and sink vertices have been omitted.)

problem, with four objective functions.

The number of required resources in pipelined implementations depend on the *data introduction interval*. Indeed, since several operations are executing concurrently in different pipe-stages, less hardware sharing is possible. Conversely, an upper bound on the resource usage implies a lower bound on the data introduction interval. These bounds are useful in determining the frontier of the design space, and the values of DII of interest. By exploring the resource usage and latency for different values of the DII (usually a few), the design space can be characterized and an efficient solution chosen. The limiting cases are those in which the DII matches the latency (unpipelined circuit) and when the DII is unity (maximum rate pipeline).

The scheduling and binding problems are more complex in the case of pipelined circuits, because several operations may be executing concurrently in different stages of the pipeline. Scheduling pipelined circuits under a required DII constraint will be described in Section 8.1 and binding in Section 8.2.

Control synthesis for pipelined circuits is more complex. Present synthesis research efforts have dealt with static pipelines and data-independent delay operations. Control synthesis of sequencing graphs with data-independent delay operations can be achieved by extending the techniques shown in Sections 7.1 and 7.2. The operations with start time $t_{i+kDII}; k = 1, 2, \ldots, \lceil t_N/DII \rceil$ are activated by word i (of a microcoded implementation) or at state s_i (of a hard-wired implementation). Unresolved and difficult control synthesis issues are related to the global control of the pipeline, that would handle stalling, flushing and bypasses. This is the subject of ongoing research.

8.1 Scheduling pipelined circuits.

We consider in this Section the extensions of the scheduling algorithms to the case of pipelined models. This problem is referred to in the literature as *functional pipelining*.

A formal model for counting the sharable resources can be derived again in terms of the ILP model. Constraint (6) of Section 5.2.1 needs to be modified because the operations at steps $j + pDII; p \in Z^+; \forall j$ are executed

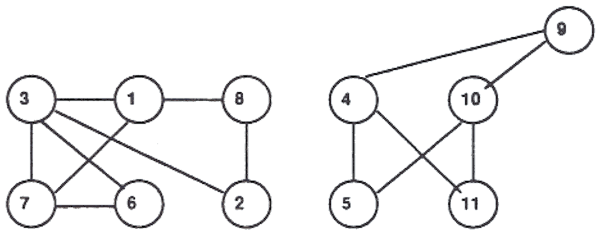

Figure 24: Compatibility graph for DII = 2.

simultaneously and cannot be shared. If we denote by L the latency, or an upper bound on the latency when this is unknown, the constraint on the number of resources used at each step j is:

$$\sum_{p=0}^{\lfloor (L-j)/DII \rfloor} \; \sum_{i \; s.t. \; \mathcal{T}(v_i)=k} \; \sum_{l=j-d_i+1+pDII}^{j+pDII} x_{il} \le a_k \quad k = 1,2,\ldots,n_{res}; \;\; j = 0,1,\ldots,t_N \tag{17}$$

Heuristic scheduling algorithms can support functional pipelining. For example, list scheduling algorithm can be used to solve the resource constrained scheduling problem with a given DII. The equation above can be used to check whether the resource bound is violated at any given step, and therefore to determine the schedulable candidates. An example of such a list scheduling algorithm was implemented in program Sehwa [44], where operation chaining is done concurrently with scheduling and where the priority function is based on the sum of the operation propagation delays from the candidate to the sink vertex.

8.2 Resource sharing and binding for pipelined circuits.

Resource sharing in pipelined implementations is limited by the pipeline throughput. Indeed, by increasing the throughput we increase the concurrency of the operations and therefore their conflicts. We comment in this Section on resource-dominated circuits only.

To be more specific, let us consider a scheduled sequencing graph. For the sake of simplicity, let us assume that all the operations have unit execution delay. Then, any operation with start time $t_j, j = 0, 1, \ldots, t_N$ is concurrent with any other operation with start time $t_j + pDII$, $p = 0, 1, \ldots \lceil t_N / DII \rceil$. This allows us to construct compatibility and conflict graphs, and to achieve a binding with minimum (or near minimum) area cost, for a given schedule and data-introduction interval. In addition, given a schedule and a latency t_N, an array of binding and area evaluations can be achieved for a set of DIIs.

Example 8.2. Consider the pipelined scheduled sequencing graph of Figure 23, with DII=2. The corresponding compatibility graph is shown in Figure 24, that can be contrasted to the compatibility graph for the non-pipelined implementation (DII = 4) shown in Figure 25. The compatibility graph for DII=1 has no edges. (Note the the compatibility graph shown in Figure 25 differs from the graph of Figure 13, because they relate to different schedules.) □

When considering hierarchical sequencing graphs, special attention has to be paid for branching constructs. Indeed, when the branch bodies are expanded in the sequencing graph, operations in different alternative branches are compatible when they have the same type. However, sharing pairs of compatible exclusive operations in different

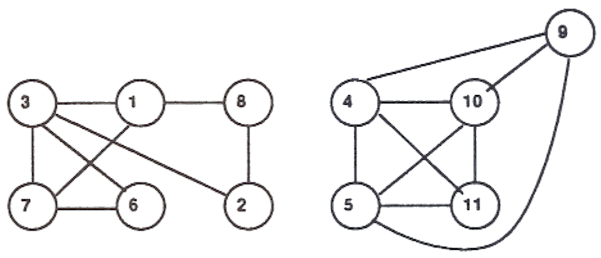

Figure 25: Compatibility graph for DII = 4.

time steps, may create deadlocks in the pipeline when they form *twisted pairs*. Therefore special attention has to be paid for these cases. We refer the interested to [22] for further details.

9 High-level synthesis systems.

Several contributions have been done to the field, and it is impossible to comment on all of them here. Some specialized books [9, 10, 15] describe in detail the most relevant results. We would like to present here a brief history of this area, and to describe the most salient features of some systems.

Early work in the field was related to compiling register-transfer level representations into logic circuits. The *Expl* system, developed in the seventies at Carnegie-Mellon University, was the first that considered series/parallel trade-offs. Successive efforts at CMU concentrated on converting behavioral models in the ISPS language into logic circuits, while addressing many of the fundamental problems related to scheduling and binding. At the same time, the *Mimola* system was developed at the University of Kiel, Germany, and could synthesize a CPU and microcode from an input specification. Later the system was ported to Honeywell, where it is now used.

The field matured in the eighties, when a few synthesis systems were developed at several locations. The most notable examples are the *ADAM* system at University of Southern California, the *CADDY/CALLAS* system at the University of Karlsruehe, *McPitts* at MIT and the *VSS* system at University of California at Irvine. Researchers at Carleton University developed several algorithms for high-level synthesis. Similarly, researchers at AT&T Bell Laboratories, General Electric, and at IBM T.J.Watson contributed algorithms and programs, including the *Yorktown Silicon Compiler* [15] and *HIS*.

At present, several systems are in development and in use at some major corporations. CAD vendor companies market systems that synthesize circuits from VHDL and Verilog descriptions, performing resource sharing and control synthesis, but not yet scheduling. Hence, such systems cannot be classified as high-level synthesis systems to a full extent.

We describe now three systems that are archetypes of different high-level synthesis styles and that address different classes of target circuits. In particular we review the CMU System Architect's Workbench, Stanford Olympus Synthesis System and the family of Cathedral Systems developed at IMEC, Belgium.

9.1 The System Architect's Workbench

Research at Carnegie Mellon University on high level system specifications opened the way to a set of tools for high-level synthesis, developed over more than one decade. These tools, are now collected under the name of *System*

Architect's Workbench [51]. Their purpose is to explore architectural choices. Hardware systems are described in ISPS or Verilog, that can be simulated and compiled into an intermediate data-flow format called Value Trace (VT). The Value Trace can be edited graphically, to perform operations such as partitioning and expansion of selected blocks. It can be annotated, to provide a link between the behavioral specification and the corresponding structural domain by program *Coral.*

Synthesis in the System Architect's Workbench is in terms of hardware resources, i.e. predefined library macrocells, such as ALUs, adders and multipliers. Recently, the system has been extended to cope with target implementations in terms of Field-Programmable Gate Arrays.

The workbench consists of a set of tools. *Aparty* is an automatic partitioner, based on a cluster search. *Cstep* is responsible for deriving the hardware control portion: it is based on a list scheduling algorithm, under resource constraints. *Emucs* is a global data allocator, that binds resources based on the interconnection cost. *Busser* synthesizes the bus interconnection, by optimizing the hardware using a clique covering algorithm. *Sugar* is a dedicated tool for microprocessor synthesis. It recognizes some specific components of a processor (e.g. an instruction decode unit) and takes advantage of these structures in synthesis. All the tools are interfaced to each other, and they have been used successfully for a few years.

9.2 The Olympus Synthesis System

The *Olympus Synthesis System*, developed at Stanford University, is a vertically integrated set of tools for the synthesis of digital circuit designs. The system is specifically designed to support synthesis of Application-Specific Integrated Circuits from behavioral-level descriptions, written in a hardware description language called *HardwareC*. HardwareC is a language with both procedural and declarative semantics and a C-like syntax [30].

The *Olympus* system supports synthesis with timing constraints at the behavioral, structural and logic levels. A front-end tool, called *Hercules*, performs parsing and behavioral-level optimization. The circuit behavior can be simulated at the functional level by program *Ariadne*, that interprets the sequencing graph models. Program *Theseus* provides a waveform display facility. Program *Hebe* performs structural synthesis. It strives to compute a minimal-area implementation subject to performance requirements, modeled as relative timing constraints. Hebe applies the relative scheduling algorithm after having bound resources to operations. If a valid schedule cannot be found that satisfies the timing constraints, a new resource binding is tried. Binding and scheduling are iterated until a valid solution is found, unless Hebe determines that the constraints cannot be met and need to be relaxed. Details are reported in reference [30].

A logic synthesis and simulation program, called *Mercury*, and a library binding tool, *Ceres*, complete the system, as shown in Figure 26. The system has been used to design three ASIC chips at Stanford University and it has been tested against benchmark circuits for high-level synthesis.

9.3 The Cathedral Synthesis Systems

The *Cathedral* project was developed at IMEC, in connection with the Catholic University of Leuven in Belgium and other partners under the auspices of project Esprit of the European Community. Cathedral rejects the idea of the existence of a general purpose silicon compiler, in analogy with the present lack of software compilers for multiple source languages and back-ends. Therefore, Cathedral is designed to map behavioral descriptions of a particular class of designs, namely Digital Signal Processors (DSP), into a particular hardware model. *Cathedral-I* is a hardware compiler for bit-serial digital filters. *Cathedral-II* is a synthesis system for single-chip concurrent bit-parallel

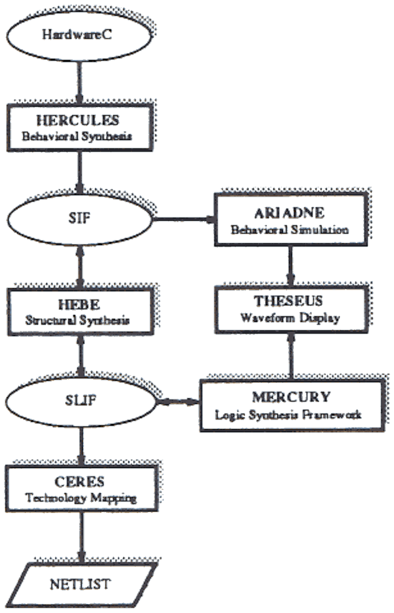

Figure 26: The Olympus Synthesis System.

processors. Typical applications are speech synthesis and analysis, digital audio, modems, etc ... *Cathedral-III* targets hard-wired bit-sliced architectures, intended for the implementation of algorithms in real-time video, image and communication domain. The data-paths consist of *application-specific units*, that are compositions of functional resources tailored to a specific application [42]. *Cathedral-IV* is used for implementing very repetitive algorithms for video processing. Cathedral-I and II have been described extensively in the literature [15].

The general design methodology in Cathedral-II is called "meet in the middle" strategy. There are two sets of tasks in the system. The former is compiling behavioral descriptions into an interconnection of instances of primitive modules, such as arithmetic components. The latter is a set of parametrizable module generators for these modules, that construct the physical layout and that can be viewed as a set of procedures called by the high-level compiler. The basic components of the architecture are six execution units, which are prototypes of data-path, memories, I/O units and controllers.

Hardware description is done in the Silage language. Hardware compilation includes the following tasks: system partitioning into processes and protocols; data-path synthesis, i.e. mapping partitioned behavior into execution-units while minimizing the interconnection busses; control synthesis based on a microcode style. The data-path synthesis step is done with the aid of an architecture knowledge data-base. Control synthesis is based on a heuristic scheduling algorithm. The physical layout is achieved by invoking the module generators. These modules can be seen as a library of high-level cells. They are designed to be portable across different technologies.

10 Conclusions.

High-level synthesis and optimization techniques provide a means of raising the abstraction level of the input description of a circuit and performing coarse-grain area/performance trade-offs. As a result, higher productivity is expected as well as higher quality circuits, because the design space can be thoroughly explored.

High-level synthesis involves computationally intractable problems. Hence heuristic algorithms are usually applied, in particular to the scheduling and binding problems. Design systems have been constructed based on the algorithms described here. They have been successfully used for circuit design in both research and product development.

Several issues are still open in this field and require further investigation. First, the definition of a commonly accepted language for synthesis, with precise hardware semantics as well as support for interface description. Indeed, timing waveforms at interfaces are an integral part of design specifications and graphics can be more expressive than text in this case. Second, the improvement of the algorithms for performance-oriented design, with particular reference to synthesis and optimization of pipelined circuits. Last, but not least, the integration of high-level synthesis with logic and physical synthesis, that would permit accurate estimation of area and delay parameters and their use in earlier stages of high-level optimization.

The trend toward larger circuit integration and system-level design mandates increasingly higher modeling abstractions and corresponding synthesis systems. High-level synthesis techniques will be crucial components of those CAD systems used to design competitive circuits and systems. Much fundamental and applied research is needed to solve the open problems and to insure the availability and efficiency of such systems to a large community of electronic designers.

11 Acknowledgements

This survey has been sponsored by NSF, jointly with DEC, under a Presidential Young Investigator Award.

References

[1] A.Aho, R.Sethi and J.Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1988

[2] Balakrishnan, A.Majumdar, D. Banerji, J.Linders and J.Majithia, "Allocation of Multiport Memories in Data Path Synthesis", *IEEE Transactions on CAD*, vol. CAD-7, no. 4, pp. 536-540, April 1988.

[3] T.Blackman, J.Fox and C.Rosebrugh, "The Silc Silicon Compiler: Language and Features" *Proc. ACM/IEEE Design Automation Conference*, June 1985, pp.232-237.

[4] F. Brewer, D. Gajski, *Knowledge Based Control in Micro Architecture Design*, Proceeding 24th DAC p. 203-209, June 1987.

[5] R. Camposano, R. A. Bergamaschi, *Synthesis Using Path-based Scheduling Algorithms and Exercises*, Proceedings of 27th Design Automation Conference, Orlando, FL, June 1990, pp. 450-455.

[6] R.Camposano, "Path-Based Scheduling for Synthesis", *IEEE Transaction on CAD*, Vol CAD-10, No. pp. 85-93, January 1990.

[7] R. Camposano, W. Rosenstiel, *Synthesizing Circuits from Behavioral Descriptions*, IEEE Trans. on CAD, Vol 8, No. 2, Feb 1989, pp. 171-180.

[8] R. Camposano, W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions", *IEEE Transactions on CAD*, Vol 8, No. 2, Feb 1989, p. 171-180.

[9] R. Camposano and W.Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publisher, 1991

[10] R. Camposano and R.Walker, Editors, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publisher, 1991

[11] R. Camposano, L.F. Saunders, R.M. Tabet, *High-Level Synthesis from VHDL*, IEEE Design&Test of Computers, March, 1991

[12] G.De Micheli, P.Antognetti and A.Sangiovanni-Vincentelli, Editors, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, M.Nijhoff, 1987.

[13] G. De Micheli, D. Ku, F. Mailhot, T. Truong, *The Olympus System for Digital Design*, IEEE Design & Test October 1990, pp. 37-53.

[14] D.Gajski N.Dutt, A.Wu and S.Lin, *High-Level Synthesis*, Kluwer, 1992.

[15] D.Gajski, *Silicon Compilation*, Addison Wesley, 1988.

[16] B.Pangrle and D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Transactions on CAD*, vol. CAD-6, no. 6, pp. 1098-1112, November 1987.

[17] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, 1992.

[18] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[19] G.Goossens, J.Vandewalle and H.De Man, "Loop Optimization in Register-Transfer level scheduling for DSP systems", *Proceedings of the ACM/IEEE Design Automation Conference*, 1989, pp. 826-831.

[20] L. Hafer, A. Parker, "Automated Synthesis of Digital Hardware", *IEEE Transaction on Computers*, Vol C-31 No 2, February 1982.

[21] T.C.Hu, "Parallel Sequencing and Assembly Line Problems", *Operations Research*, No. 9, pp. 841-848.

[22] K.Hwang, A.Casavant, M.Dragomirecky and M. d'Abreu, "Constrained Conditional Resource Sharing in Pipeline Synthesis", *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 52-55, 1988.

[23] J.Huisken, H.Janssen, P.Lippens, O.McArdle, R.Segers, P.Zegers, A. Delaruelle and J. van Meerbergen, *Efficient Design of Systems on Silicon with PYRAMID*, in *Logic and Architecture Synthesis for Silicon Compilers*, North Holland, Amsterdam, 1989.

[24] C.-T.Hwang, J.-H. .Lee and Y-C Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis", *IEEE Transaction on CAD*, Vol CAD-10, No. 4, pp. 464-475, April 1991.

[25] O.Karatsu, "VLSI Design Standardization Effort in Japan, *Proceedings of the Design Automation Conference* 1989, pp.50-55.

[26] D.Knapp "Synthesis from Partial Structure", in D. Edwards, Editor, *Design Methodologies for VLSI and Computer Architecture*, pp. 35-51, Elsevier Science Publications, 1989.

[27] D. W. Knapp, *Manual Rescheduling and Incremental Repair of Register-Level Datapaths*, Proc ICCAD-89. Santa Clara, CA, Nov 1989, pp 58-61.

[28] D.W. Knapp *Feedback Driven Datapath Optimization in Fasolt* ICCAD-90, Santa Clara, California, November 1990, pp.300-303

[29] T. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Boston, MA; Kluwer Academic Publishers. 1985.

[30] D.Ku, G.De Micheli, *High-Level Synthesis of ASICS under Timing and Synchronization Constraints*, Kluwer 1992.

[31] D. Ku, G. De Micheli, *Relative Scheduling Under Timing Constraints, Proceedings of* 27th *Design Automation Conference*, Orlando, Florida, June, 1990.

[32] D.Ku and G. De Micheli, " Relative Scheduling under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits" *IEEE Transactions on CAD/ICAS*, Vol. 11, No. 6, April 1992, pp. 696-718.

[33] D. Ku, G. De Micheli, "Constrained Resource Sharing and Conflict Resolution in Hebe", *Integration. VLSI Journal*, Vol. 12, No. 2, December 1991, pp. 131-166.

[34] D. Kuck, *The Structure of Computers and Computation*, Wiley, 1978.

[35] C. Leiserson, F. Rose, and J. Saxe. "Optimizing Synchronous Circuitry by Retiming." *Proceedings of the 3rd CalTech Conference on Very Large Scale Integration*, 1983.

[36] Y.Liao and C. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints", *IEEE Transactions on CAD/ICAS*, Vol CAD-2, No. 2, April 1983, pp.62-69.

[37] R.Lipsett, C. Schaefer and C.Ussery, *VHDL: Hardware Description and Design*, Kluwer, 1991

[38] M. McFarland, A.Parker and R. Camposano, *The High-level Synthesis of Digital Systems*, Proceedings of the IEEE, Vol. 78, No. 2, February 1990, pp. 301-318.

[39] M. J. McFarland, "Reevaluating the Design Space for register Transfer Hardware Synthesis", *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp. 184-187, 1987.

[40] M. J. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", *Proceedings* 23th *Design Automation Conference*, June 1986, p. 474-480.

[41] P.Michel, U.Lauther and P.Duzy, *The Synthesis Approach to Digital System Design*, Kluwer, 1992.

[42] S.Note, W.Geurts, F.Chattor and H.DeMan, "Cathedral-III: Architecture-driven High-level Synthesis for High throughput DSP applications", *Proc. Des Autom. Conf*, 1991, pp. 597-602.

[43] A. Parker, J. Pizarro, M. Mlinar, *MAHA: A Program for Data Path Synthesis*, Proceedings 23th Design Automation Conference, June 1986, p. 461-466.

[44] N.Park and A.Parker "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications' *IEEE Transaction on CAD*, Vol CAD-7, No. 3, pp. 356-370, March 1988.

[45] P.Paulin and J.Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transaction on CAD/ICAS*, Vol CAD-8, No. 6, pp. 661-679, July 1989.

[46] P. G. Paulin, J. P. Knight, E. F. Girczyc, *HAL: A Multi-Paradigm Approach to Automatic Data-path Synthesis*, Proceedings 23th Design Automation Conference, June 1986, pp. 263-270.

[47] B.Preas and M.Lorenzetti, *Physical Design Automation of VLSI Systems*, Benjamin Cummings Wesley, 1988.

[48] J.Southard, "MacPitts, An Approach to Silicon Compilation", *IEEE Computer*, Vol 16, No. 12, December 1983, pp. 59-70.

[49] D. Springer and D.Thomas, "Exploiting the Special Structure Of Conflict and Compatibility Graphs in High-Level Synthesis", *ICCAD, Proceedings of the International Conference on Computer-Aided Design*, pp.254-259, 1990.

[50] L. Stok, *Architecture Synthesis and Optimization of Digital Systems*, Ph.D. Dissertation, Eindhoven University The Netherlands, 1991.

[51] D.Thomas, E.Lagnese, R.Walker, J.Nestor, J.Rajan and R.Blackburn, *Algorithmic and Register Transfer Leve Synthesis: The System Architect's Workbench*, Kluwer Academic Publisher, 1990

[52] D. Thomas, C. Hitchcock III, T. Kowalski, J. Rajan, R. Walker, *Automatic Data Path Synthesis*, IEEE Computer magazine, December 1983.

[53] D.Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991

[54] H.Trickey, "Flamel: A High-Level Hardware Compiler", *IEEE Transaction on CAD/ICAS*, vol. CAD-6, No. 2, pp.259-269, March 1987.

[55] C. Tseng, D. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", *IEEE Transaction on CAD*, Vol CAD-5, pp. 379-395, July 1986.

[56] G.Zimmermann, "The MIMOLA Design System: Detailed Description of the Software System. *Proc 16th Des Autom. Conf*, 1979, pp 56-63.