# 8

# Synthesis of ASICs with Hercules and Hebe

David C. Ku · Giovanni De Micheli

*Center for Integrated Systems*
Stanford University
Stanford, CA 94305

## 1 Introduction

Computer-aided synthesis of digital circuits from behavioral specifications offers an effective means of dealing with the increasing complexity of digital hardware design. The benefits of such a methodology include shortened design time to reduce design cost, ease of modification of the hardware specifications to enhance design reusability, and the ability to more effectively and completely explore the different design tradeoffs between area of the resulting hardware and its processing time.

Most of the previous work in high-level synthesis addressed processor and digital signal processing designs, as documented by the other chapters of this book. They are effective in using domain-specific knowledge in synthesizing designs with certain architectures. One area that we believe to be particularly suited for high-level synthesis is Application Specific Integrated Circuits (ASICs). ASICs are typified by control-dominated interface and communication circuits, such as for bus arbitration or communication line interfaces. For ASICs, reducing the design time and cost is often more important than minimizing area or improving performance.

While logic synthesis techniques have been established as standard steps in the design methodology for digital circuits, high-level synthesis techniques have been lagging behind for several reasons. One of the most difficult issue is that as designs increase in size and complexity, system integration issues, such as coordinating and interfacing between the components, often dominate a design. In particular, hardware interfacing and design constraints on timing and area need to be addressed at both the *design specification* level, by providing more powerful hardware models that supports external synchronization

and timing constraints, and at the *design synthesis* level, by providing powerful synthesis algorithms that can either guarantee the resulting implementation satisfy the given constraints, or indicate when no such implementation exists. In addition, although effective logic synthesis techniques are available, they have not been adequately incorporated by many systems to complement and enhance the high-level optimizations. This can result in inflexible design styles, lack of integration between different synthesis domains, and an inability to use lower level synthesis information to guide the high-level design tradeoffs. Finally, given the diversity of the approaches to digital circuit design, it is difficult to encode all implementation decisions in terms of algorithms or rules that can be universally applied. Practical high-level synthesis techniques therefore need to support both *automatic* and *user-driven* synthesis modes to leverage off the designer's knowledge and experience.

Existing synthesis approaches and algorithms are limited in their ability to synthesize interface and communication designs, with few exceptions [1, 13, 4]. In particular, most input languages of synthesis systems do not support interfacing and synchronization with external signals and events. Furthermore, the synthesis paradigm of most systems is to design hardware that performs a set of computations within a given amount of time. Being able to specify a global timing constraint to limit the overall latency of a design is clearly inadequate for interface and communication circuits that require complex handshaking protocols with other hardware modules. For example, a specification for a bus interface may require that a ready signal be detected before putting some data on the data lines, with the stipulation that there be at least 5 cycles separating the detection of the ready signal and the outputting of data. Therefore, two of the most important issues in the synthesis of ASIC designs are *external interfacing* and *synchronization* with input signals and events, and the support for *local timing constraints* that specify bounds on the timing of input and output events.

**System Overview.** With the motivation described in the previous section, we have developed a system for the high-level synthesis of general-purpose synchronous digital circuits, with specific attention to the requirements of ASIC designs. The system is divided into two parts: *Hercules* that performs the front-end parsing and behavioral optimizations, and *Hebe* that synthesizes one or more structural implementations that realize the given behavior. In addition, we have developed as input to the system a synthesis-oriented hardware description language (HDL) called *HardwareC*. HardwareC serves as a platform for experimenting with different constructs in specifying hardware for synthesis.



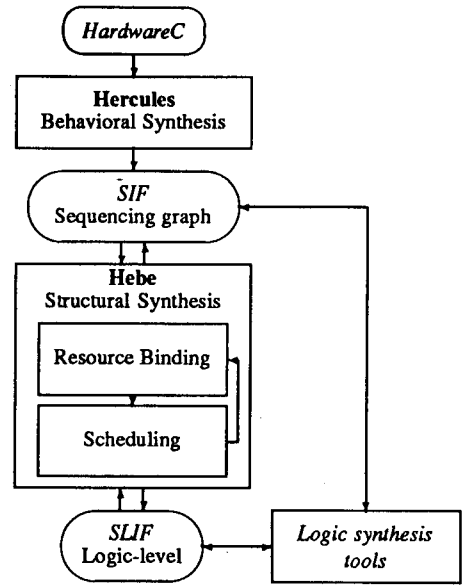



Figure 1: Block diagram of the *Hercules* and *Hebe* system.

Hercules and Hebe transform a behavioral description of hardware in HardwareC, through a series of translations and optimizations, to a synchronous logic implementation that satisfies the timing and resource constraints that are imposed on the design. Hercules performs the front-end parsing and behavioral optimizations, with the objective of identifying the parallelism in the input specification. It generates an implementation-independent description of the hardware behavior in a graph-based representation, called the *Sequencing Intermediate Form* (SIF). Hebe binds operations to resources and control steps, and generates a logic-level implementation consisting of data-path and control, described in the *Structural/Logic Intermediate Form* (SLIF). A block diagram of the system is shown in Figure 1. Note that logic synthesis tools are used to optimize the combinational logic portions of the design, and they provide feedback on area and delay that is used to drive Hebe.

We would like to emphasize the support of the system for the following features.

- *External interfacing and synchronization.* The ability to *wait* for the occurrence of a particular input event, i.e. assertion of a ready signal, is necessary to coordinate the actions between a set of concurrently execution modules. This interfacing is specified as either synchronization mechanisms or data-dependent loops in the input description, and it is modeled as *unbounded delay operations* in the synthesis formulation.

- *Detailed timing constraints.* Detailed timing constraints specify upper and lower bounds on the activation of pairs of operations. The bounds can either be specified directly in terms of number of cycles, or they can be derived given a cycle time. They permit the specification and synthesis of designs with complex protocols and strict timing requirements. We have developed a technique called *relative scheduling* that permits the analysis of timing constraints in the presence of unbounded delay operations.

- *Partial binding of operations to resources.* Often the designer may wish to share resources by manually binding certain operations to resources in order to meet some high level objectives. It is important to capture this partial structure in the specification to guide the synthesis algorithms; for example, the partial structure is used to limit the number of different design implementations.

- *Synthesis algorithms with provable properties.* Timing and resource constraints are used to drive the synthesis optimizations, to ensure that either the resulting implementation satisfy the required constraints, or that no such implementation exists.

- *Logic synthesis techniques.* To meet the area requirements, resource sharing is a necessary part of the synthesis system. Since resources correspond to models that are described and invoked in the high level description, the characterization of resources to evaluate sharing feasibility is carried out using logic synthesis techniques to provide estimates on timing and area. This methodology is particularly suited for ASIC designs that tend to rely on application-specific logic functions. The use of logic synthesis for estimates improves the quality of the synthesized designs, and avoids erroneous high-level decisions due to insufficient data or inappropriate assumptions.



The synthesis flow can be fully automated, transforming an input HardwareC description directly to a logic-level implementation. The system also supports user-driven synthesis, where a designer can intervene and drive high-level decisions based on an evaluation of the possible design tradeoffs.

## 2 Hardware Modeling

The input to the synthesis system is a description of hardware behavior in a high-level hardware description language called *HardwareC* [6]. The motivation for choosing HardwareC over other hardware description languages is because we would like, in addition to developing synthesis algorithms and techniques, to experiment with different language constructs for synthesis. The interaction between specification and synthesis provides an effective framework for testing new synthesis approaches and algorithms.

As its name suggests, HardwareC has a C-like syntax. However, the language has its own hardware semantics, and it differs from the C programming language in many respects. HardwareC supports both declarative semantic (e.g. interconnection of modules) and procedural semantic (e.g. set of operations ordered in time) in the modeling of hardware. There are four fundamental design abstractions, corresponding to *block*, *process*, *procedure*, and *function models*. At the topmost level, a design is described in terms of a block, which contains an interconnection of logic and instances of other blocks and processes. A process consists of a hierarchy of procedures and functions, and represents a functionality that executes repeatedly, restarting itself upon completion. Since a process executes concurrently and independently with respect to the other processes in the system, it allows the modeling of *coarse-grain* parallelism at the functional level. A procedure or function is an encapsulation of operations, and may contain calls to other procedures and functions.

HardwareC supports the usual iterative and branching constructs, including both fixed-iteration and data-dependent looping constructs. Data-dependent loops can be used to detect signal transitions, which are important in describing external interfaces. For example, the construct `while (data==0);` will wait until the rising transition of the signal `data`. In addition, there are several features of HardwareC that support hardware specification and synthesis:

- *Interprocess communication* – To support communication and synchronization among the concurrent processes, HardwareC supports both *parameter passing* and *message passing*. The former assumes the existence

of a shared medium (e.g. shared bus or memory) that interconnects the hardware modules implementing processes. The handshaking protocols are described in the HardwareC description. The latter uses a synchronous *send/receive* mechanism that can be used for synchronization or data transfer. The corresponding hardware for communication, as well as its protocol, are automatically synthesized.

- *Explicit instantiation of models* – Hierarchical designs are supported through the use of model calls. A call to a model can be either *generic* or *instantiated*: a generic call invokes a model without specifying the particular instance that is used to implement the call, whereas an instantiated call identifies also a specific instance of the model which will implement the call. Through explicit instantiation of model calls, HardwareC supports resource constraints and partial bindings of operations to resources. The designer can constrain the synthesis system to explore a subset of the possible structures corresponding to a behavioral model to satisfy a particular architectural requirement.

- *Template models* – A template model is a single description that describes a class of behaviors. As an example, a single template can be used to describe a family of adders of different size. Templates are similar to high-level module generation, and are therefore very useful in describing libraries of hardware operators at a high level.

- *Degree of parallelism* – For procedural semantic models, HardwareC offers the designer the ability to adjust the degree of parallelism in a given design through the use of *sequential* ([ ]), *data-parallel* ({ }), or *parallel* (< >) groupings of operations. In the first case, operations are executed sequentially. In the second one, all operations are executed in parallel, unless data dependency requires serialization. In the last case, all operations execute in parallel unconditionally. Parallel grouping is used, for example, to describe the swapping of two variables without the use of a temporary variable, i.e. $< a = b; b = a >$.

- *Constraint specification* – Timing constraints are supported through tagging of operations, where lower and upper bounds are imposed on the time separation between the tags. Timing constraints are useful in interface specification by constraining the time separation between I/O operations. Resource constraints limit the number of resources and the binding of operations to resources in the final implementation.



```
process gcd ( xin, yin, restart, result )
    in port xin[8], yin[8], restart;
    out port result[8];
[
    boolean x[8], y[8];
    tag a, b;

    /* set output to zero during computation */
    write result = 0;

    /* wait for restart to go low */
    while ( restart )
        ;

    /* sample inputs */
    <
        constraint mintime from a to b = 1 cycles;
        constraint maxtime from a to b = 1 cycles;

        a: x = read(xin);
        b: y = read(yin);
    >

    /* Euclid's algorithm */
    if ( (x != 0) & (y != 0) ) {
        repeat {
            while (x >= y)
                x = x - y;
            /* swap values */
            < y = x; x = y; >
        } until (y == 0);
    } else
        x = 0;

    /* write result to output */
    write result = x;
]
```

Figure 2: Example of a *HardwareC* description to find the greatest common divisor of two values.

An example of a HardwareC description that computes the greatest common divisor of two numbers is given in Figure 2. The model `gcd` waits until the `restart` signal is low, samples the inputs, then performs Euclid's algorithm iteratively. The read operations are tagged, and timing constraints are applied on the tags to ensure that the reading of `yi` occurs exactly 1 cycles after the reading of `xi`. Note that any statement in the description can be tagged.

# 3 Hercules – Behavioral Synthesis

The objective of *behavioral synthesis* is to identify as much parallelism as possible in the input description. This gives an indication of the *fastest* design that the system can produce, assuming that in the design implementation each operation is implemented by a dedicated hardware component. While this assumption may not be realistic in some cases due to area and interconnection costs, it is important to compute the related performance as a limiting bound for a given behavior.

The input HardwareC description is parsed and translated first into an abstract syntax tree representation, which provides the underlying model for semantic analysis and behavioral transformations. The transformations are categorized into *user-driven* and *automatic* transformations. User-driven transformations are optional, and allow the designer the capability of modifying the model calls and hierarchy of the input description. They include the following:

- *Selective in-line expansion of model calls*, where a call to a model is replaced by the functionality of the called model. Once expanded, the optimization algorithms can be applied across the call hierarchy.
- *Selective operator to library mapping*, where operators, such as "+" or "−", in the input description are mapped into calls to specific library template models. Although an operator can be synthesized in a variety of different implementation styles, the designer is often constrained to elements of a particular library. With such mapping, the designer has the flexibility to select the specific implementation for the operators. If no mapping is given, then by default the operators are implemented as combinational logic expressions.

Automatic transformations optimize the behavior by performing transformations similar to those found in optimizing compilers [17, 16, 15]. The automatic



transformations are carried out without human intervention, and include the following:

- `For`-*loop unrolling*, where fixed-iteration loops are unrolled to increase the scope of the optimizations.
- *Constant and variable propagation*, where the reference to a variable is replaced by its last assigned value.
- *Reference stack resolution*, where multiple and conditional assignments to variables are resolved and eliminated by creating *multiplexed values* that can subsequently be referenced.
- *Common sub-expression elimination*, where redundant operations that produce the same results are removed.
- *Dead-code elimination*, where operations whose effects are not visible outside the model are removed.
- *Conditional elimination*, where conditionals with branches containing only combinational logic are collapsed to increase the scope in which logic synthesis can be applied.

Upon completion of the automatic transformations, the behavior is optimized with respect to the data-dependencies that exist among the operations. At this point, *combinational coalescing* is performed to group together combinational logic operations into *expression blocks*. The expression blocks define the largest scope (without crossing control step boundaries) in which logic synthesis can be applied, where a control step is a fundamental unit of sequencing in synchronous systems and corresponds to a clock cycle. The expression blocks are used to identify the critical combinational logic delays through the design. They are directly passed to logic synthesis for minimization and technology mapping, the results of which are fed-back as estimates on area and timing that are used to refine the design. *Operation chaining*, where multiple operations are packed within a single control step, is supported through coalescing. Combinational coalescing is important particularly for ASIC designs because of their extensive use of logic expressions in the hardware specification.

**Sequencing Graph Model.** The optimized behavior resulting from behavioral synthesis is translated into a **sequencing graph** abstraction called the *sequencing intermediate form* (SIF). The sequencing graph is a concise way of capturing

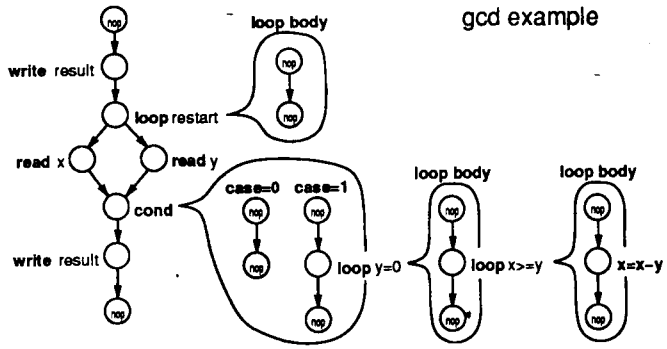



Figure 3: The SIF representation for the *gcd* example. Note the hierarchical nature of the model.

the partial order among a set of operations, and it is modeled as a polar (single source-vertex and single sink-vertex), directed acyclic graph. The source vertex represents the start of computation, and the sink vertex represents the completion of all computations. The vertices represent the operations to be performed, and the edges represent the dependencies that are either explicit in the hardware specification, or represent dependencies due to *data-flow* restrictions (i.e. a value must be written before it can be referenced) or hardware *resource-sharing* considerations (i.e. two operations sharing the same hardware resource must be serialized to avoid simultaneously activating the resource). A vertex is enabled when all its predecessors have completed execution. Since a vertex may have multiple predecessors and successors, the model supports *multiple threads of concurrent execution flow.*

The vertices are categorized as either *simple* or *complex* vertices. Simple vertices include primitive computations in the language, such as arithmetic or logic expressions and message passing commands. Complex vertices allow groups of operations to be performed, and include model calls, conditionals, and loops. The complex vertices induce a hierarchical relationship among the graphs. A call vertex invokes the sequencing graph corresponding to the called model. A conditional vertex selects among a number of branches, each of which is modeled by a sequencing graph. A loop vertex iterates its body until the exit condition is satisfied; the body of the loop is also a sequencing graph. The sequencing graph is *acyclic* because only structured control-flow constructs are assumed (no `goto`), and loops are broken through the use of hierarchy. An



example of the sequencing graph for the *gcd* example of Figure 2 is shown in Figure 3.

**Hardware Resources** In contrast to micro-architectural synthesis systems that use a predefined set of library elements as building blocks, Hercules and Hebe treat each *model* in the input description as a *resource* that can be allocated and shared among the calls to the models (either procedures or functions). Each different implementation of the called model represents a specific *resource type* with its own area and performance characteristics. For example, two calls to a model `A` can be implemented either by a single resource corresponding to the hardware implementation of `A`, where both calls share the use of the resource; or by two resources, where each call is implemented by a different resource. Operators such as + or − can either be converted into calls to the appropriate library models, or by default be implemented in terms of logic expressions.

There are several motivations for adopting this view of models and resources. First, many complex ASIC designs use application specific logic functions in describing hardware behavior; the delay and area attributes of these modules are not known *a priori* since they depend on the particular details of the logic functionality. Having the ability to synthesize in a bottom-up manner each model according to its distinct needs allows the calling models to more accurately estimate their resource requirements. Second, the granularity of resource sharing can be controlled by the designer in the high level specification, which increases the flexibility of the system. Finally, instead of relying on parameterized and predefined modules, logic synthesis techniques applied hierarchically to each model can significantly improve the quality of the resulting design.

# 4 Hebe – Structural Synthesis

The input to the structural synthesis phase consists of a sequencing graph model of the hardware behavior to synthesize, along with the following constraints, which can either be specified in the input hardware description, or entered interactively by the designer.

- *Timing constraints* – that specify upper and lower bounds on the time separation between pairs of operations.
- *Resource constraints* – that both limit the number of instances allocated for each resource type, and partially bind operations to specific instances of the resource pool.

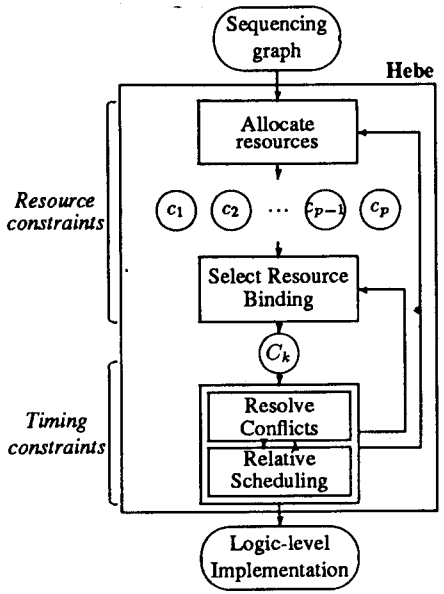



Figure 4: Block diagram of the *Hebe* structural synthesis system.

- *Cycle time* – for the final synchronous logic implementation.

The constraints are not mandatory; they serve to guide the synthesis system in obtaining an acceptable solution. For example, if the cycle time is not given, then the cycle time is by default equal to the critical combinational logic delay in the final implementation. An important characteristic of Hebe is its support for detailed timing and resource constraints at both the design specification and synthesis levels.

The objective of Hebe is to explore the design tradeoffs by sharing hardware resources to obtain a suitable implementation that satisfies the user constraints on resource and timing. Although we consider an implementation to be acceptable as long as both the resource and timing constraints are satisfied, Hebe provides a framework in which the designer can experiment with different design goals that indicate the emphasis of the final implementation with respect to area and/or performance. Hebe performs a number of distinct but interdependent subtasks. The subtasks include **data-path** optimization and generation, such as *resource allocation* and *binding* to bind operations to specific resources,



as well as *scheduling* to bind operations to control steps. In addition, **control** optimization and generation is performed to synthesize and minimize the corresponding control logic. The interaction among these various tasks is critical in determining how effectively or completely the space of design alternatives can be explored.

An effective strategy is to perform resource binding before scheduling, as in *Caddy* [2] and *BUD* [11]. This strategy has the advantage of being able to provide the scheduling phase with detailed interconnection delays, because the interconnect structure is known once a binding of operations to resources has been made. This basic approach is extended in Hebe to provide closer interaction and guidance to the designer, and is shown in Figure 4. The flow of structural synthesis in Hebe is described as follows.

- *Perform resource allocation and binding.* For a resource allocation that satisfies the *resource constraints*, operations are bound to specific resources. The allocation and binding are guided by the desired design goals, i.e. minimum area or maximal performance.

- *Resolve resource conflicts.* A binding implies a certain degree of resource sharing, and in general *resource conflicts* may arise when more than one operation simultaneously attempt to activate the same resource. The resource conflicts can be resolved by *serializing* operations bound to the same resource that can otherwise execute in parallel. Different bindings may have different I/O behavior because of this serialization, and *timing constraints* are used to determine whether a given I/O behavior meets the imposed timing requirements.

- *Perform scheduling.* After the conflicts have been resolved, scheduling is performed to bind operations to control steps, subject again to the required timing constraints. Scheduling is necessary for control generation.

The synthesis algorithm explores the different possible resource binding alternatives by iterating these three tasks. We describe now Hebe's formulation of the design space, and how it is explored in obtaining a desired implementation.

**Formulating the Design Space.** More specifically, a **resource pool** is a set of hardware resources (e.g. implementations of models) with an upper-bound on the number of instances of each type of hardware resources that the user allows in the final implementation. A **resource binding** is a matching of the operations (i.e. the vertices of the sequencing graph) to specific resources in the

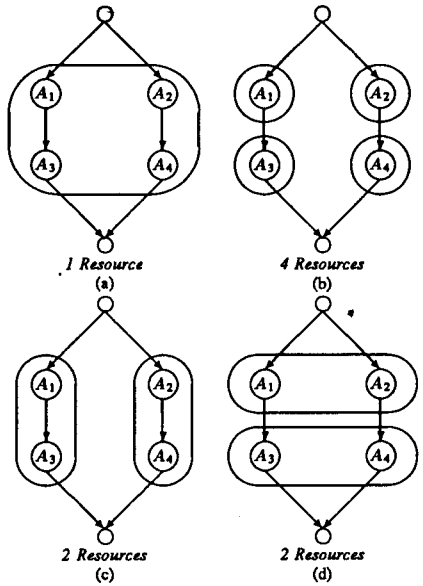



Figure 5: Examples of resource binding, where operations within a group are bound to the same resource instance.

resource pool. The **design space** is the entire set of resource bindings that are compatible with the partial binding of operations to resources that is specified as a form of resource constraint in the input description. A resource binding is considered *valid* if its resource conflicts can be resolved and a scheduling exists that satisfies the timing constraints. Therefore, Hebe's goal is to find the "best" valid resource binding, subject to a particular design goal.

Examples of resource bindings for a sequencing graph containing four calls to model A are shown in Figure 5. All operations that are grouped together share the same resource instance in the final implementation, e.g. the resource binding of Figure 5(a) utilizes one resource instance, the resource binding of Figure 5(b) utilizes four resource instances, etc. In the case of allocating two resource instances, (c) is favored over (d) if the design goal is to minimize the latency of the graph. The reason is because it is necessary to resolve two resource conflicts in (d), i.e. between $A_1$ and $A_2$, and between $A_3$ and $A_4$. The conflict resolution may increase the latency of the graph, as described later.

An important aspect of the design space formulation is that it is a *complete* characterization of the entire set of possible design tradeoffs for a given allocation of resources, and offers two important advantages:



- *Uniformly incorporates partial binding information.* In some circuits the designer may wish to bind certain operations to resources in order to achieve high-level design goals. This information can be used to limit the design space such that the synthesis system focus on the remaining unbound operations. At the extreme, if all operations are bound, then the design space trivially reduces to a single point.

- *Supports exact and heuristic algorithms.* With exact algorithms, Hebe guarantees that given a binding configuration, it is possible to find a resolution of the resource conflicts that satisfies the timing constraints, if one exists. Otherwise, the system can detect the inconsistency and inform the designer accordingly. Since exact algorithms may not be practical for some designs, they are complemented by heuristic algorithms that try to find a resolution of the resource conflicts, but they are not guaranteed to find a solution when one exists. As a result, the exact algorithms may be necessary if the heuristic fails, when an optimum implementation is sought.

The exploration of the binding alternatives is guided by the principle of finding a conflict resolution satisfying the given constraints for points of the design space of increasing cost. The search stops when a valid design point is found. Therefore, when exact algorithms are used, an optimum solution is found, while near-optimal implementations are otherwise achieved.

**Exploring the Design Space.** The size of the design space may be large, because it grows exponentially with the number of shareable resources. However, it is often the case in ASIC designs that the number of shareable resources is sufficiently small to make systematic exploration of all resource bindings practical and meaningful. For these cases, *exact* pruning techniques are used to limit the search for a valid binding. In exact pruning, a *partial order* is imposed on both the resource allocations and on the bindings such that if a resource allocation fails to satisfy the timing constraints, then the allocations that follow it in the partial order are guaranteed to also not satisfy the timing constraints. For example, if an allocation of 3 adders and 2 multipliers fails to produce a valid binding, then allocating 2 adders and 1 multiplier will also fail to produce a valid binding.

Hebe can compute the *exact* cost for a given design point. Specifically, the area cost is obtained through logic synthesis techniques on the corresponding logic-level implementation for both the control and the data-path; the performance cost is obtained after conflict resolution and scheduling. For designs where the design space is too large, the system supports also *heuristic* search for the resource bindings. The search is based on alternative evaluation and ranking of the binding cost using a set of *cost criteria*. The cost criteria represent estimates of the effect of a particular binding on the area and delay of the final implementation, and the bindings with more favorable costs are synthesized first. The cost criteria include the following.

- *Interconnection cost*: The interconnection structure is the steering logic that guides the appropriate values to their proper destinations in the final implementation. Since a binding configuration is a complete assignment of operations to resources, the interconnection structure is completely specified. The interconnection cost is a function of the interconnect's area and delay, computed using a multiplexer-based scheme. Logic synthesis can be used to optimize the interconnect structure.

- *Area cost*: A resource binding implies a certain degree of resource utilization and sharing. The area cost estimates the total area cost of the final implementation, and includes the area costs due to the resources in the resource pool, the interconnect structure, the registers, and the control structure. More sophisticated area estimates that consider also the cost due to layout and wiring can also be incorporated as this stage, although they have not been implemented yet in the current version of the system.

- *Serialization cost*: Resource conflicts may arise due to a binding. Determining whether a conflict resolution exists under timing constraints is computationally expensive. We use the notion of *widths* of bindings to estimate the number of threads of parallelism that *need to be serialized* in order to resolve the resource conflicts. In particular, all operations bound to the same resource instance should not execute simultaneously, e.g. either there exists sequencing dependencies among the operations, or the operations occur in mutually exclusive branches of a conditional. The serialization cost is a heuristic measure of the effect of the resource binding on the performance of the design.

The decision of whether one alternative is favorable with respect to another depends on the relative importance of these criteria, which is determined by the value of a *weight* associated with each criterion. Through Hebe, the designer can experiment with different design goals by adjusting the values of the weights, where the bindings are ranked according to their costs. The designer can focus the synthesis efforts on the resource bindings with acceptable costs.



For example, if the goal is to minimize the area, then the area and interconnection costs can be used to identify the resource bindings with minimal area. Likewise, if the goal is to maximize performance under area constraints, then the area and interconnection costs can bound the search to those bindings that meet the area constraints, while the serialization cost can provide further pruning of the design space. We emphasize that a resource binding may still be invalid even if it has favorable costs. The reason is because conflict resolution and scheduling have not yet been performed at this heuristic ranking stage.

# 5 Synthesis Algorithms

Given a binding in the design space, it is necessary to resolve the resource conflicts to determine whether the binding is a valid binding. In most existing approaches, resolving conflicts is formulated as a *scheduling* problem to assign operations to fixed time slots, where two operations in different time slots can have their resources shared. However, the sequencing graph model supports operations whose execution delays are unbounded and unknown *a priori*. Unbounded delay operations are useful in modeling interfacing with external signals and events. For example, waiting for the rising edge of a request signal can be modeled as an operation whose completion indicates the detection of the rising edge. Since the rising edge can occur at any time, the execution delay of this synchronizing operation is data-dependent, and can be represented as having unbounded execution delay. The support for unbounded delay operations invalidates the traditional scheduling formulation because it is no longer possible in general to statically assign operations to fixed time slots.

To address this difficulty, we have proposed a *relative scheduling* formulation in which the activation of operations is specified as time offsets from the set of unbounded delay operations [7]. An important characteristic of this formulation is the support for detailed timing constraints. We formulate the *conflict resolution* problem as the task of *serializing* the graph model so that operations bound to the same resource cannot execute in parallel. The serialization cannot in general be arbitrarily applied due to the presence of timing constraints. The conflict resolution approach takes advantage of the relative scheduling formulation to ensure that the resulting serialized graph satisfies the required timing constraints,

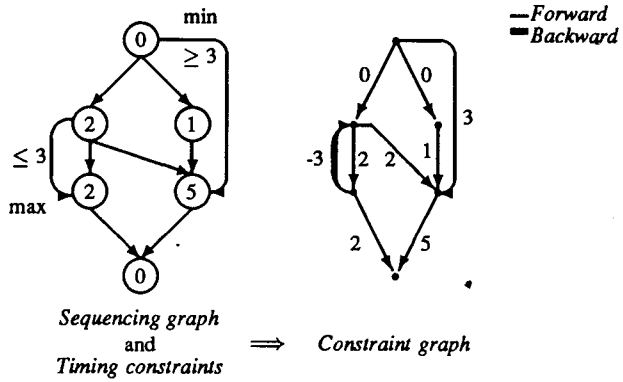



Figure 6: Example of a constraint graph, with a minimum and a maximum timing constraint. The number inside a vertex represents its execution delay.

if a solution exists. We consider an implementation of a binding to be acceptable as long as the timing constraints are satisfied. Once the graph is appropriately serialized, relative scheduling is carried out to determine the time offsets, which are used to generate the control circuit. It is important to remark that a solution to the conflict resolution implies that a valid schedule satisfying the timing constraints exists.

**Constraint Graph Model.** Before presenting the details of the conflict resolution and relative scheduling approaches, we describe first our model of hardware timing behavior in terms of a polar directed edge-weighted *constraint graph* $G(V, E)$. The vertices $V$ represent the operations, and the edges $E$ capture the precedence and timing relationships (sequencing and min/max constraints) among the operations. Each operation $v \in V$ is synchronous and therefore takes an integral number of cycles to execute, called it execution delay and is denoted by $\delta(v)$. The execution delay may not be known a priori, as in the case of external synchronization and data-dependent loops. In this case, we say the execution delay is *unbounded.*

A weight $w_{ij}$ associated with each edge $e_{ij} = (v_i, v_j) \in E$ represents the requirement that the start time of $v_j$ (denoted by $T(v_j)$) must occur later than $w_{ij}$ after the start time of $v_i$, i.e. $T(v_j) \geq T(v_i) + w_{ij}$. For example, a sequencing dependency from $v_i$ to $v_j$ is represented by a forward edge from $v_i$ to



$v_j$ with weight $\delta(v_i)$. The edges are categorized into *forward* ($E_f$) and *backward edges* ($E_b$). The forward (backward) edges have positive (negative) weights and represent minimum (maximum) timing requirements among the operations. Both forward and backward edges may have unbounded weights. Without loss of generality, we assume the graph induced by the forward edges is acyclic, and that all cycles in the graph have bounded length. Figure 6 illustrates how a constraint graph is derived from a sequencing graph with timing constraints. We refer the interested reader to [7] for details of the constraint graph model.

**Resource Conflict Resolution.** We call an *operation set* as consisting of the subset of operations that are bound to a particular resource. Obviously, if the elements of an operation set execute in parallel, then resource conflicts will arise. We formulate the problem of conflict resolution as finding an *ordering* of the elements of an operation set, such that the serialized graph satisfies the imposed timing constraints. For example, in Figure 5(d), the two calls $A_1$ and $A_2$ executing in parallel but bound to the same resource must be serialized to ensure that they cannot execute simultaneously. A straightforward approach, but computationally prohibitive, is to simply enumerate the possible orderings. We can however take advantage of the *topology* of the input sequencing graph and the set of timing constraints to reduce significantly the complexity of the ordering search.

We use the constraint graph model as the basis for the formulation. Since the objective is to find an ordering that satisfies the timing constraints, an important observation is that constraint violations will occur only if overconstraint in the form of inconsistent cyclic timing relationships is introduced. The conflict resolution approach is as follows, for a given operation set.

- *Identify operation clusters* – an operation cluster represents a subset of vertices in the operation set that are connected by a cycle in the constraint graph, i.e. a cyclic timing requirement is imposed on them.

- *Find an ordering among the operation clusters* – By definition, a partial order is induced among the operation clusters with respect to timing requirements. Therefore, we see that the problem of finding an ordering for an operation set can reduced to the problem of finding an *ordering* for the elements of an operation cluster, since any ordering of the clusters that is compatible with the original partial order will satisfy the timing constraints. By taking advantage of the *topology* of the graph, the computational complexity of the conflict resolution strategy now depends on the

size of the operation clusters instead of depending on the size of the operation sets. For designs with few cyclic timing constraints, this reduction in complexity is significant.

- *Order operations within the operation clusters* – The problem of finding an ordering of operations within an operation cluster satisfying timing constraints is NP-complete in the strong sense, since it can be cast as an instance of "sequencing with release times and deadlines" [3]. Hebe supports both *heuristic* and *exact* branch-and-bound search. The heuristic search is based on sorting the elements to be ordered by the length of the longest path from the source. Since we are interested in finding quickly one valid ordering, the heuristic search is always performed first; the branch-and-bound search is used only when the heuristic fails to find a valid solution.

- *Serialize graph according to ordering* – Once a valid ordering satisfying timing constraints is found, the sequencing graph is serialized accordingly. Scheduling is then performed, as described in the next section.

Given a valid ordering, the sequencing graph is free from resource conflicts. Furthermore, it is guaranteed that the resulting serialized graph satisfies the required timing constraints. If a valid ordering for a given binding is not found, then the binding is discarded and another one is selected.

**Relative Schedule.** With the resource conflicts resolved, scheduling is still necessary to assign the operations to control states in order to generate the control circuit for the final hardware. We use a novel technique called *relative scheduling* that uniformly supports operations with fixed and unbounded delays. We describe briefly the main results in relative scheduling. The interested reader is referred to [7] for further details.

Given a constraint graph $G(V, E)$, we define a subset of the vertices, called **anchors**, that serve as reference points for specifying the start times of the operations. The anchors consist of the source vertex and the set of unbounded delay vertices. *Offsets* are then defined with respect to each anchor of the graph. In particular, the **anchor set** of a vertex is the set of anchors that are predecessors to the vertex, and represents the *unknown* factors that affect the activation time of the vertex. The start time of a vertex is then generalized in terms of fixed time offsets from the completion of each anchor in its anchor set. Specifically, let $A(v_i)$ denote the anchor set of $v_i$, and $\sigma_a(v_i)$ as the offset from the completion of anchor $a \in A(v_i)$. The start time $T(v_i)$ of $v_i$ is given as:



$$T(v_i) = \max_{a \in A(v_i)} \{T(a) + \delta(a) + \sigma_a(v_i)\}$$

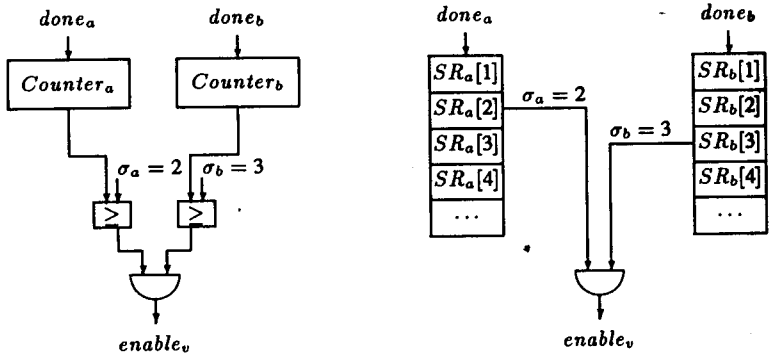
Note that if there are no unbounded delay vertices in the graph, then the start times of all operations will be specified in terms of offsets from the source vertex, which reduces to the traditional scheduling formulation.

An important consideration during scheduling is whether the timing constraints can be satisfied for any value of the unbounded delay operations. A constraint graph is **feasible** if its constraints can be satisfied when the unbounded delays are equal to zero. If there are no unbounded delay operations, then the concept of feasibility is sufficient to guarantee that a schedule exists. With the presence of unbounded delays, we extend the analysis by introducing the concept of **well-posed** constraints. Specifically, a timing constraint is well-posed if it is satisfied for all values of the unbounded delays. We are interested in well-posed constraints because the final implementation must be able to satisfy the timing constraints for any values of unbounded delays. Note that if a graph is well-posed, then it is also feasible; the contra-positive also holds, where an unfeasible graph is also ill-posed. Since feasibility can easily be checked by detecting positive cycles in the constraint graph, we can assume the constraints to be feasible in the subsequent analysis. The relative scheduling approach consists of the following steps.

1. *Checking well-posedness* – The constraint graph is first checked for well-posedness. If the constraint graph is ill-posed, it is sometimes possible to made it well-posed by additionally serializing the graph. An algorithm is applied that is guaranteed to yield a well-posed constraint graph with *minimum serialization*, if one exists. If the graph cannot be made well-posed, then no schedule exists and scheduling is aborted.

2. *Removing redundant anchors* – It is often the case that not all anchors in the anchor set are needed to compute the start time of an operation. This is due to the cascading effect of anchors that make some *redundant* in computing the start time. For a well-posed graph, we identify and remove the *redundant* anchors. Through redundancy removal, it is possible to obtain a smaller and faster control implementation because the start time depends on fewer offsets, and hence fewer synchronizations.

3. *Finding the minimum schedule* – Finally, the relative schedule can be computed by using an efficient algorithm called *iterative incremental schedul-*




Counter-based control (a)

Shift-register based control (b)

Figure 7: Alternate implementation styles for control generation: (a) *counter-based* and (b) *shift-register based.*

*ing*. It is guaranteed to find the *minimum* relative schedule, or detect the presence of inconsistent constraints.

The polynomial-time complexity of the above steps allows relative scheduling to be effectively integrated within the design space exploration.

**Control Generation.** Once we have computed the relative schedule corresponding to a constraint graph, it is necessary to generate the control logic that will activate each operation according to its schedule. In the simple case where the hardware model does not contain any unbounded delay operations, the task of control generation reduces to the traditional control synthesis approaches. For the general case, we use an approach that is based on an extension of the *adaptive control synthesis* scheme [9].

Given a schedule, we abstract the task of control generation as generating *enable/done* signals for the operation associated to each vertex $v$ such that its execution is indicated by the assertion of $enable_v$. We model the control in terms of a modular interconnection of synchronous FSMs; the FSM abstraction decouples the control generation from a particular style of logic-level implementation. For the sake of simplicity and without loss of generality, our control abstraction considers only the synchronization of an operation with respect to



the completion of its anchors. We assume that the completion of the operation corresponding to each anchor $a \in A$ is indicated by the assertion of a signal $done_a$. Details of generating $done_a$ and the support for conditional branching and looping are described in [9]. Two different approaches to generating the control for a vertex $v$ are shown in Figure 7. The vertex $v$ has two anchors, $a$ and $b$, with offsets equal to 2 and 3, respectively. The two approaches are described below.

- *Counter based control* – A counter is used to indicate the time offset from the completion of an anchor. The enable signal is described as comparisons between the values of the counters with the corresponding offsets.
- *Shift-register based control* – The comparator cost in the previous approach can be reduced by using shift registers instead of counters. In this case, the enable signal is described as the logical-and of the corresponding stages of the shift register.

Alternative strategies to describe the control logic exists, Figure 7 serves to illustrate two possible control styles. For example, a finite state machine can be generated where the control signals correspond to the output of the FSM. The control can be further optimized by a technique based on *resynchronization* of operations that can be applied to minimize the area of the control implementation, while still satisfying the timing constraints [8].

# 6 Implementation and Design Experiences

*Hercules* and *Hebe* have been implemented in C, with approximately 140,000 lines of code. They are interfaced to the logic synthesis, simulation and technology mapping tools of the *Olympus* synthesis system [12]. Hercules and Hebe have been tested on the benchmark circuits for high-level synthesis. Although many of these examples do not take full advantage of Hebe's ability to support detailed timing constraints, they serve as comparisons with existing systems. We would also like to remark that the HardwareC descriptions of these examples have been fully simulated to verify the functional correctness; the combined control and data-path have been synthesized and mapped by logic synthesis techniques.

The results of applying Hebe to some benchmark examples are shown in Figure 8. The designs include the DAIO receiver (*DAIO_rv*) and phase decoder (*DAIO_ph*), encoder (*ECC_enc*) and decoder (*ECC_dec*) portions of an

| *Example* | *Sequencing graph model* | | | | | *Implementation* | | Resources[a] |
|---|---|---|---|---|---|---|---|---|
| | $G^b$ | $N^c$ | $Lp^d$ | *Cnd* | *Expr* | *Area*[e] | *Latency*[f] | |
| DAIO_rv | 16 | 64 | 7 | 4 | 16 | 1931 | unbound | none |
| DAIO_ph | 9 | 58 | 2 | 3 | 13 | 1796 | unbound | none |
| ECC_enc | 4 | 47 | 1 | 1 | 9 | 1586 | 17 | E3(6),E4(3)[g] |
| | | | | | | 657 | 23 | E3(1),E4(1) |
| ECC_dec | 4 | 53 | 1 | 1 | 14 | 2923 | 18 | E4(9) |
| | | | | | | 940 | 27 | E4(1) |
| Frisc | 26 | 140 | 1 | 5 | 6 | 12583 | 23 | $+_{16}(1),-_{16}(1)$ |
| Gcd | 10 | 37 | 3 | 3 | 7 | 1111 | unbound | none |
| Traffic | 2 | 7 | 1 | 0 | 1 | 191 | unbound | none |
| Tseng | 1 | 11 | 0 | 0 | 2 | 3101 | 5 | $*_8(1),/_8(1),$ $+_8(1),-_8(1)$ |
| | | | | | | 2901 | 4 | $*_8(1),/_8(1),$ $+_8(3),-_8(1)$ |
| Diffeq | 2 | 17 | 1 | 0 | 1 | 401 | 12 | $+_8(2),-_8(1),$ $*_8(4)$ |
| | | | | | | 365 | 8 | $+_8(2),-_8(2),$ $*_8(6)$ |
| Elliptic | 1 | 37 | 0 | 0 | 0 | 8710 | 52 | $+_{16}(1),*_{16}(1)$ |
| | | | | | | 9023 | 45 | $+_{16}(4),*_{16}(1)$ |
| | | | | | | 10823 | 30 | $+_{16}(4),*_{16}(2)$ |

[a] Multiplier $*_8$ ($*_{16}$) requires 5 cycles, with area cost 2012 (8910).
[b] $G$ is the number of sequencing graphs.
[c] $N$ is the number of vertices in all graphs.
[d] $Lp$ and *Cnd* are the number of loops and conditionals.
[e] Based on LSI Logic Compacted Array 10K library costs.
[f] Number of cycles, *unbound* means unbounded execution delay.
[g] E3 (E4) is combinational logic function with area cost 60 (90).

Figure 8: Results of applying Hebe to benchmark examples.



error-correction module, 16-bit RISC-style microprocessor (*Frisc*), 8-bit greatest common divisor (*Gcd*), traffic light controller (*Traffic*), Tseng's 8-bit example (*Tseng*), 8-bit differential equation solver (*Diffeq*), and the elliptic filter with arbitrary 16-bit coefficients (*Elliptic*). This table gives for each example information related to the sequencing graph model: $G$ denotes the number of sequencing graphs in the model, $N$ denotes the total number of vertices, *Lp*/*Cnd* denotes the number of data-dependent loops and conditionals, and *Expr* denotes the number of logic expression blocks.

Synthesis by Hebe is based on a cycle time of $50ns$, where *Area* is the area cost of the final implementation in the LSI Logic Compacted Array (LCA) 10K library, and *Latency* is the number of cycles to execute the design. The resources that are used by a design are also shown. For example, combinational 16-bit adders ($+_{16}$) and 16-bit multipliers ($*_{16}$) requiring 5 cycles to execute are used in the Elliptic example. Some design points are shown for designs with non-trivial design space, such as for Diffeq and Elliptic. The control is based on the shift-register implementation described earlier. Note that in the Tseng example, the size of the implementation with resource sharing is larger than the size of the dedicated implementation due to the cost of interconnect. The execution times of Hercules for most examples range from a few seconds to several minutes, running on a DecStation 5000/200. The execution times of Hebe depends both on the extent to which the design space is searched and on the time spent on logic synthesis. Synthesizing one binding configuration requires up to a few minutes to execute, with most designs requiring several seconds.

In addition, the system has been used to design three ASIC circuits at Stanford University, namely a Bi-dimensional Discrete Cosine Transform (BDCT) chip [14], a Digital Audio Input Output (DAIO) chip [10], and a decoder chip for the Multi-Anode Microchannel Array (MAMA) detector for the space telescope [5]. The BDCT chip is used for video compression applications. An $8 \times 8$ BDCT architecture was synthesized and implemented in a compiled macro-cell design style as a $9 \times 9$ $mm^2$ image in $2\mu$ CMOS technology. The DAIO chip provides an interface, following the Audio Engineering Standard (AES) protocol, between a standard 16/32 microprocessor bus with audio devices, such as compact disk or digital audio tape player. The DAIO specification in HardwareC was compiled and mapped into a logic netlist suitable for implementation in LSI Logic 9K-series sea-of-gates technology. The logic specification had about 6000 equivalent gates. The MAMA chip is designed to discriminate the information generated by a multi-anode detector in a space telescope. Also described in HardwareC, it was synthesized and fabricated with LSI Logic 9K-series sea-of-

gates technology.

Hercules and Hebe are part of the *Olympus Synthesis System*. For availability information, please send electronic mail to olympus@chronos.stanford.edu.

## 7 Acknowledgments

Rajesh Gupta generated and tested the benchmark examples, Dave Filo implemented the register folding and control optimization techniques, Thomas Truong implemented a graphic display package and simulator for the SIF graph, and Frederic Mailhot implemented the logic synthesis interface and technology mapper *Ceres*, as used by Hebe to evaluate the cost of a design. Their contributions and helpful discussions are gratefully acknowledged. This research was sponsored by NSF/ARPA, under grant No. MIP 8719546, by AT&T and DEC jointly with NSF, under a PYI Award program, and by a fellowship provided by Phillips/Signetics.

## References

[1] G. Borriello and R. Katz. Synthesis and optimization nof interface transducer logic. In *ICCAD, Proceedings of International Conference on Computer-Aided Design*, pages 56–60, November 1987.

[2] R. Camposano and W. Rosenstiel. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on CAD/ICAS*, Vol. 8(No. 2):171–180, February 1989.

[3] M. Garey and D. Johnson. *Computers and Intractability*. W. Freeman and Company, 1979.

[4] S. Hayati, A. Parker, and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *ICCD, Proceedings of International Conference on Computer Design*, pages 382–387, October 1988.

[5] D. B. Kasle. High resolution decoding techniques and single-chip decoders for multi-anode microchannel arrays. *Proceedings of Int'l Society of Optical Eng.*, Vol. 1158:311–318, August 1989.



[6] D. C. Ku and G. De Micheli. Hardwarec - a language for hardware design (version 2.0). *Stanford University CSL Technical Report*, CSL-TR-90-419, April 1990.

[7] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints. In *DAC, Proceedings of Design Automation Conference*, pages 59–64, June 1990.

[8] D. C. Ku and G. De Micheli. Control optimization based on resynchronization of operations. In *DAC, Proceedings of Design Automation Conference*, June 1991.

[9] D. C. Ku and G. De Micheli. Optimal synthesis of control logic from behavioral specifications. *Journal of VLSI Integration (To appear)*, 1991.

[10] M. Ligthart, A. Bechtolsheim, G. De Micheli, and A. El Gamal. Design of a digital audio input output chip. In *CICC, Proceedings of Custom Integrated Circuits Conference*, pages 15.1.1–15.1.6, May 1989.

[11] M. J. McFarland. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *DAC, Proceedings of Design Automation Conference*, pages 474–480, June 1986.

[12] G. De Micheli, D. C. Ku, F. Mailhot, and T. Truong. The olympus synthesis system for digital design. *IEEE Design and Test Magazine*, pages 37–53, October 1990.

[13] J. Nestor and D. Thomas. Behavioral synthesis with interfaces. In *DAC, Proceedings of Design Automation Conference*, pages 112–115, June 1986.

[14] V. Rampa and G. De Micheli. The bi-dimensional dct chip. In *ISCAS, Proceedings of International Symposium on Circuits and Systems*, pages 220–225, May 1989.

[15] E. A. Snow. *Automation of module set independent register-transfer level design*. Ph.D. Dissertation, Carnegie Mellon University, April 1978.

[16] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on CAD/ICAS*, Vol. CAD-6:259–269, March 1987.

[17] R. Walker and D. Thomas. Behavioral transformation for algorithmic level ic design. *IEEE Transactions on CAD/ICAS*, Vol. 8:1115–1128, October 1989.