

Tutorial on High-Level Synthesis

Michael C. McFarland, SJ
Boston College
Chestnut Hill, MA 02167

Alice C. Parker
University of Southern California
Los Angeles, CA 90007

Raul Camposano
IBM T.J. Watson Research Center
Yorktown Heights, NY

Abstract. High-level synthesis takes an abstract behavioral specification of a digital system and finds a register-transfer level structure that realizes the given behavior. In this tutorial we will examine the high-level synthesis task, showing how it can be decomposed into a number of distinct but not independent sub-tasks. Then we will present the techniques that have been developed for solving those subtasks. Finally, we will note those areas related to high-level synthesis that are still open problems.

1. Introduction

1.1 What is High-Level Synthesis?

The synthesis task is to take a specification of the behavior required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints. By *behavior* we mean the way the system or its components interact with their environment, i.e., the mapping from inputs to outputs. *Structure* refers to the set of interconnected components that make up the system – something like a netlist. Usually there are many different structures that can be used to realize a given behavior. One of the tasks of synthesis is to find the structure that best meets the constraints, such as limitations on cycle time, area or power, while minimizing other costs. For example, the goal might be to minimize area while achieving a certain minimum processing rate.

Synthesis can take place at various levels of abstraction because designs can be described at various levels of detail. The type of synthesis we will focus on in this tutorial begins with a behavioral specification at what is often called the *algorithmic level*. The primary data types at this level are integers and/or bit strings and arrays, rather than boolean variables. The input specification gives the required mappings from sequences of inputs to sequences of outputs. It should constrain the internal structure of the system to be designed as little as possible. From that input specification, the synthesis system produces a description of a *register-transfer* level structure that realizes the specified behavior. This structure includes a *data path*, that is, a network of registers, functional units, multiplexers and buses, as well as hardware to control the data transfers in that network. If the control is not integrated into the datapath – and it usually is not – the synthesis system must also produce the specification of a finite state machine that drives the datapaths so as to produce the required behavior. The control specification could be in terms of microcode, a PLA profile or random logic.

High-level synthesis as we define it must be distinguished from other types of synthesis, which operate at different levels of the design hierarchy. For example, high-level synthesis is not to be confused with *logic* synthesis, where the system is specified in terms of logic equations, which must be optimized and mapped into a given technology. Logic synthesis might in fact be used on a design after high-level synthesis has been done, since it presupposes the sorts of decisions that high-level synthesis makes. At the other end of the spectrum, there is some promising work under way on system level synthesis, for example on partitioning an algo-

rithm into multiple processes that can run in parallel or be pipelined. However, this work is still in its preliminary stages; and we will not report on it here.

1.2 Why Study High Level Synthesis?

In recent years there has been a trend toward automating synthesis at higher and higher levels of the design hierarchy. Logic synthesis is gaining acceptance in industry, and there has been considerable interest shown in synthesis at higher levels. There are a number of reasons for this:

- **Shorter design cycle.** If more of the design process is automated, a company can get a design out the door faster, and thus have a better chance of hitting the market window for that design. Furthermore, since much of the cost of the chip is in design development, automating more of that process can lower the cost significantly.
- **Fewer Errors.** If the synthesis process can be verified to be correct – by no means a trivial task – there is a greater assurance that the final design will correspond to the initial specification. This will mean fewer errors and less debugging time for new chips.
- **The ability to search the design space.** A good synthesis system can produce several designs for the same specification in a reasonable amount of time. This allows the developer to explore different trade-offs between cost, speed, power and so on, or to take an existing design and produce a functionally equivalent one that is faster or less expensive.
- **The design process is self-documenting.** An automated system can keep track of what design decisions were made and why, and what the effect of those decisions was.
- **Availability of IC technology to more people.** As more design expertise is moved into the synthesis system, it becomes easier for a non-expert to produce a chip that meets a given set of specifications.

We expect this trend toward higher levels of synthesis to continue. Already there are a number of research groups working on high-level synthesis, and a great deal of progress has been made in finding good techniques for optimization and for exploring design trade-offs. These techniques are very important because decisions made at the algorithmic level tend to have a much greater impact on the design than those at lower levels.

There is now a sizable body of knowledge on high-level synthesis, although for the most part it has not yet been systematized. In the remainder of this paper, we will describe what the problems are in high-level synthesis, and what techniques have been developed to solve them. To that end, Section 2 will describe the various tasks involved in developing a register-transfer level structure from an algorithmic level specification. Section 3 will describe the basic techniques that have been developed for performing those tasks. Finally, Section 4 will look at those issues that have not been adequately addressed and thus provide promising areas for future research.

2. The Synthesis Task

The system to be designed is usually represented at the algorithmic level by a programming language such as Pascal [27] or Ada [8], or by a hardware description language that is similar to a programming language, such as ISPS [2]. Most of the languages used are *proceduring* languages. That is, they describe data manipulation in terms of assignment statements that are organized into larger blocks using standard control constructs for sequential execution, conditional execution and iteration. There have been experiments, however, with various types of non-procedural hardware description languages, including applicative, LISP-like languages [11] and declarative languages such as Prolog.

The first step in high-level synthesis is usually the compilation of the formal language into an internal representation. Two types of internal representations are generally used: parse trees and graphs. Most approaches use variations of graphs that contain both the data-flow and the control flow implied by the specification [16], [26], [12]. Fig. 1 shows a part of a simple program that computes the square-root of X using Newton's method, along with its graphical representation. The number of iterations necessary in practice is very small. In the example, 4 iterations were chosen. A first degree minimax polynomial approximation for the interval $\langle 1/16, 1 \rangle$ gives the initial value. The data-flow and control flow graphs are shown separately in the figure for intelligibility. The control graph is derived directly from the explicit order given in the program and from the compiler's choice of how to parse the arithmetic expressions. The data-flow graph shows the essential ordering of operations in the program imposed by the data relations in the specification. For example, in fig. 1, the addition at the top of the diagram depends for its input on data produced by the multiplication. This implies that the multiplication must be done first in any valid ordering of the operations. On the other hand, there is no dependence between the $I + 1$ operation inside the loop and any of the operations in the chain that calculates Y, so the $I + 1$ may be done in parallel with those operations, as well as before or after them. The data-flow graph can also be used to remove the dependence on the way internal variables are used in the specification, since each value produced by one operation and consumed by another is represented uniquely by an arc. This ability to reassign variables is important both for reordering operations and for simplifying the datapaths.

```

...
Y := 0.222222 + 0.888889 * X ;
I := 0 ;
DO UNTIL I > 3 LOOP
  Y := 0.5 * ( Y + X / Y ) ;
  I := I + 1 ;
ENDDO ;
...

```

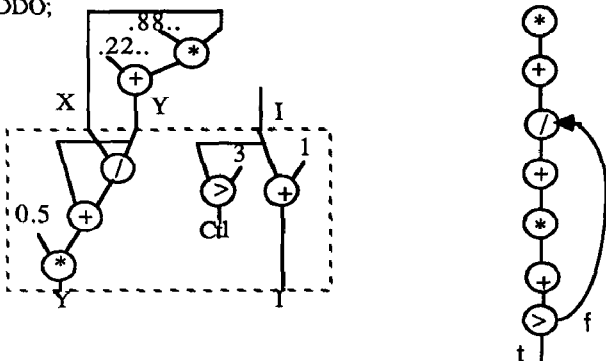


Figure 1. High-level Specification and graph for sqrt

The rest of this section outlines the various steps used in turning the intermediate form into a RT-level structure, using the square root example to illustrate the different steps.

Since the specification has been written for human readability and not for direct translation into hardware, it is desirable to do some initial optimization of the internal representation. These high-level transformations include such compiler-like optimizations as dead code elimination, constant propagation, common subexpression elimination, inline expansion of procedures and loop unrolling. Local transformations, including those that are more specific to hardware, are also used. In the example, the loop-ending criterion can be changed to $I = 0$ using a two-bit variable for I. The multiplication times 0.5 can be replaced by a right shift by one. The addition of 1 to I can be replaced by an increment operation. The internal representation after these optimizations is depicted on the left in fig. 2. Loop unrolling can also be done in this case since the number of iterations is fixed and small.

The next two steps in synthesis are the core of transforming behavior into structure: scheduling and allocation. They are closely interrelated and depend on each other. Scheduling consists in assigning the operations to so-called control steps. A control step is the fundamental sequencing unit in synchronous systems; it corresponds to a clock cycle. Allocation consists in assigning the operations to hardware, i.e. allocating functional units, storage and communication paths.

The aim of *scheduling* is to minimize the amount of time or the number of control steps needed for completion of the program, given certain limits on the available hardware resources. In our example, a trivial special case uses just one functional unit and one memory. Each operation has to be scheduled in a different control step, so the computation takes $3+4*5=23$ control steps. To speed up the computation at the expense of adding more hardware, the control graph can be packed into control steps as tightly as possible, observing only the essential dependencies required by the data-flow graph and by the loop boundaries. This form is shown in fig. 2. Notice that two dummy nodes to delimit the loop boundaries were introduced. Since the shift operation is free, with two functional units the operations can now be scheduled in $2+4*2=10$ control steps.

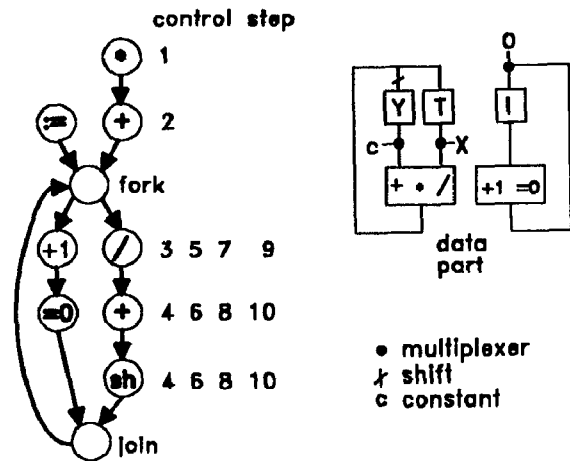


Figure 2. Optimized Control Graph and Schedule

In *allocation*, the problem is to minimize the amount of hardware needed. The hardware consists essentially of functional units, memory elements and communication paths. To minimize them together is usually too complex, so in most systems they are minimized separately. As an example, consider the minimization of functional units. Mutually exclusive operations, e.g. operations in different control steps, clearly can share functional units. The problem is then to group those operations which are mutually exclusive, so that the minimum number of groups results. If functional units are capable of performing only some operations, only those operations that can be performed by some common functional unit can be grouped. The allocation of functional units for the given schedule in fig. 2 is minimal in this sense. The problems of minimizing the amount of storage and the complexity of the communication paths for a given schedule can be formulated similarly.

In memory allocation, values that are generated in one control step and used in another must be assigned to storage. Values may be assigned to the same register when their lifetimes do not overlap. Storage assignment should be done in a way that not only minimizes the number of registers, but also simplifies the communication paths.

Communications paths, including buses and multiplexers, must be chosen so that the functional units and registers are connected as necessary to support the data transfers required by the specification and the schedule. The most simple type of communication path allocation is based only on multiplexers. Buses, which can be seen as distributed multiplexers, offer the advantage of requiring less wiring, but they may be slower than multiplexers. Depending on the application, a combination of both may be the best solution.

In addition to designing the abstract structure of the data path, the system must decide how each component of the data path is to be implemented. This is sometimes called *module binding*. For the binding of functional units, known components such as adders can be taken from a hardware library. Libraries facilitate the synthesis process and the size/timing estimation, but they can prevent efficient solutions that require special hardware. The synthesis of special-purpose full-custom hardware is possible, but it makes the design process more expensive, possibly requiring extensive use of logic synthesis and layout synthesis.

Once the schedule and the data paths have been chosen, it is necessary to synthesize a controller that will drive the data paths as required by the schedule. The synthesis of the control hardware itself can be done in different ways. If hardwired control is chosen, a control step corresponds to a state in the controlling finite state machine. Once the inputs and outputs to the FSM, that is, the interface to the data part, have been determined as part of the allocation, the FSM can be synthesized using known methods, including state encoding and optimization of the combinational logic. If microcoded control is chosen instead, a control step corresponds to a microprogram step and the microprogram can be optimized using encoding techniques for the microcontrol word.

Finally, the design has to be converted into real hardware. Lower level tools such as logic synthesis and layout synthesis complete the design.

The major problem underlying all these tasks is the extremely large number of design possibilities which must be examined in order to select the design which meets constraints and is as near as possible to the optimal design. The "design space" that needs to be

searched is multi-dimensional and discontinuous, and it is hard even to find a canonical set of operators that systematically take you through that space. Furthermore, the shape of the design space is often problem-specific, so that there is no methodology that is guaranteed to work in all cases.

Finding the best solution to even a limited problem such as scheduling is difficult enough. Many synthesis subtasks, including scheduling with a limitation on the number of resources and register allocation given a fixed number of registers, are known to be NP-hard. That means that the process of finding an optimal solution to these is believed to require a number of steps exponential in the size of the data set. Yet in high-level synthesis there are several such tasks, and they cannot really be isolated, since they are interdependent.

3. Basic Techniques

3.1 Scheduling

We distinguish two dimensions along which scheduling algorithms may differ: (1) the interaction between scheduling and operator and/or datapath allocation; and (2) the type of scheduling algorithm used.

3.1.1 Interaction with Allocation As noted earlier, scheduling and operator allocation are interdependent tasks. In order to know whether two operations can be scheduled in the same control step, one must know whether they use the same functional unit. Moreover, finding the most efficient possible schedule for the real hardware requires knowing the delays for the different operations, and those can only be found after the details of the functional units and their interconnections are known. On the other hand, in order to make a good judgement about how many functional units should be used and how operations ought to be distributed among them, one must know what operations will be done in parallel, which comes from the schedule. Thus there is a vicious circle, since each task depends on the outcome of the other.

A number of approaches to this problem have been taken by synthesis systems. The most straightforward one is to set some limit (or no limit) on the number of functional units available and then to schedule. This is done, for example, in the Facet system [28], in the early Design Automation Assistant [13], and in the Flamel system [27]. This limit could be set as a default by the program or specified by the user. A somewhat more flexible version of this approach is to iterate the whole process, first choosing a resource limit, then scheduling, then changing the limit based on the results of the scheduling, rescheduling and so on until a satisfactory design has been found. This is done, for example, under user control in the MIMOLA system [29] and under guidance of an expert system, with feedback from the datapath allocator, in Chippe [5].

Another approach is to develop the schedule and resource requirements simultaneously. For example, the MAHA [21] system allocates operations to functional units as it schedules, adding functional units only when it cannot share existing ones. The force-directed scheduling in the HAL [22] system schedules operations within a given time constraint so as to balance the number of functional units required in each control step. The number of functional units allocated is then the maximum number required in any control step. HAL also includes a feedback loop that allows the scheduling to be repeated after the detailed datapaths have been designed, when more is known about delays and interconnect costs.

The Yorktown Silicon Compiler (YSC) [4] does allocation and scheduling together, but in a different way. It begins with each operation being done on a separate functional unit and all operations being done in the same control step. Additional control steps are added for loop boundaries, and as required to avoid conflicts over register and memory usage. The hardware is then optimized so as to share resources as much as possible. If there is too much hardware or there are too many operations chained together in the same control step, more control steps are added and the datapath structure is again optimized. This process is repeated until the hardware and time constraints are met.

Finally, functional unit allocation can be done first, followed by scheduling. In the BUD system [17], operations are first partitioned into clusters, using a metric that takes into account potential functional unit sharing, interconnect, and parallelism. Then functional units are assigned to each cluster and the scheduling is done. The number of clusters to be used is determined by searching through a range of possible clusterings, choosing the one that best meets the design objectives.

In the Karlsruhe CADDY/DSL system [25], the datapath is built first, assuming maximal parallelism. This is then optimized, locally and globally, guided by both area constraints and timing. The operations are then scheduled, subject to the constraints imposed by the datapath.

3.1.2 Scheduling Algorithms There are two basic classes of scheduling algorithms: transformational and iterative/constructive. A transformational type of algorithm begins with a default schedule, usually either maximally serial or maximally parallel, and applies transformations to it to obtain other schedules. The transformations move serial operations in parallel and parallel operations in series. Transformational algorithms differ in how they choose what transformations to apply.

Barbacci's EXPL [1], one of the earliest high-level synthesis system, used exhaustive search. That is, it tried all possible combinations of serial and parallel transformations and chose the best design found. This method has the advantage that it looks through all possible designs, but of course it is computationally very expensive and not practical for sizable designs. Exhaustive search can be improved somewhat by using branch-and-bound techniques, which cut off the search along any path that can be recognized to be suboptimal.

Another approach to scheduling by transformation is to use heuristics to guide the process. Transformations are chosen that promise to move the design closer to the given constraints or to optimize the objective. This is the approach used, for example, in the Yorktown Silicon Compiler [4] and the CAMAD design system [23]. The transformations used in the YSC can be shown to produce a fastest possible schedule for a given specification.

The other class of algorithms, the iterative/constructive ones, build up a schedule by adding operations one at a time until all the operations have been scheduled. They differ in how the next operation to be scheduled is chosen and in how they determine where to schedule each operation.

The simplest type of scheduling, as soon as possible (ASAP) scheduling, is local both in the selection of the operation to be scheduled and in where it is placed. ASAP scheduling assumes that the number of functional units has already been specified.

Operations are first sorted topologically; that is, if operation x_2 is constrained to follow operation x_1 by some necessary dataflow or control relationship, then x_2 will follow x_1 in the topological order. Operations are taken from the list in order and each is put into the earliest control step possible, given its dependence on other operations and the limits on resource usage. Figure 3 shows a dataflow graph and its ASAP schedule. This was the type of scheduling used in the CMUDA system [10], in the MIMOLA system and in Flamel. The problem with this algorithm is that no priority is given to operations on the critical path, so that when there are limits on resource usage, operations that are less critical can be scheduled first on limited resources and thus block critical operations. This is shown in Figure 3, where operation 1 is scheduled ahead of operation 2, which is on the critical path, so that operation 2 is scheduled later than is necessary, forcing a longer than optimal schedule.

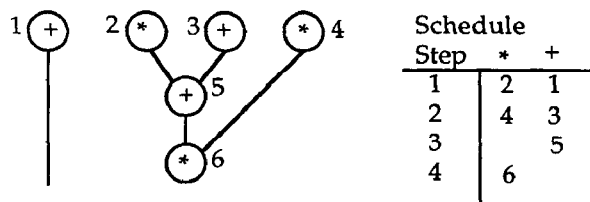


Figure 3. ASAP Scheduling

List scheduling overcomes this problem by using a more global criterion for selecting the next operation to be scheduled. For each control step to be scheduled, the operations that are available to be scheduled into that control step, that is, those whose predecessors have already been scheduled, are kept in a list, ordered by some priority function. Each operation on the list is taken in turn and is scheduled if the resources it needs are still free in that step; otherwise it is deferred to the next step. When no more operations can be scheduled, the algorithm moves to the next control step, the available operations are found and ordered, and the process is repeated. This continues until all the operations have been scheduled. Studies have shown that this form of scheduling works nearly as well as branch-and-bound scheduling in microcode optimization [6]. Figure 4 shows a list schedule for the graph in Figure 3. Here the priority is the length of the path from the operation to the end of the block. Since operation 2 has a higher priority than operation 1, it is scheduled first, giving an optimal schedule for this case.

A number of schedulers use list scheduling, though they differ somewhat in the priority function they use. The scheduler in the BUD system uses the length of the path from the operation to the end of the block it is in. Elf [8] and ISYN [19] use the "urgency" of an operation, the length of the shortest path from that operation to the nearest local constraint.

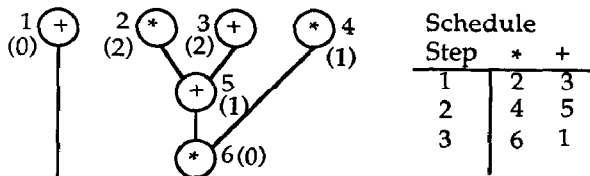


Figure 4. A List Schedule

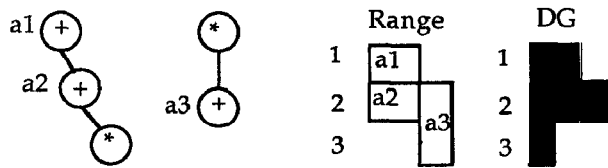


Figure 5. A Distribution Graph

The last type of scheduling algorithm we will consider is global both in the way it selects the next operation to be scheduled and in the way it decides the control step in which to put it. In this type of algorithm, the range of possible control step assignments for each operation is calculated, given the time constraints and the precedence relations between the operations. In freedom-based scheduling, the operations on the critical path are scheduled first and assigned to functional units. Then the other operations are scheduled and assigned one at a time. At each step the unscheduled operation with the least freedom, that is, the one with the smallest range of control steps into which it can go, is chosen, so that operations that might present more difficult scheduling problems are taken care of first, before they become blocked.

In force-directed scheduling, the range of possible control steps for each operation is used to form a so-called *Distribution Graph*. The distribution graph shows, for each control step, how heavily loaded that step is, given that all possible schedules are equally likely. If an operation could be done in any of k control steps, then $1/k$ is added to each of those control steps in the graph. For example Figure 5 shows a dataflow graph, the range of steps for each operation, and the corresponding distribution graph for the addition operations, assuming a time constraint of three control steps. Addition a1 must be scheduled in step 1, so it contributes 1 to that step. Similarly addition a2 adds 1 to control step 2. Addition a3 could be scheduled in either step 2 or step 3, so it contributes $1/2$ to each. Operations are then selected and placed so as to balance the distribution as much as possible. In the above example, a3 would first be scheduled into step 3, since that would have the greatest effect in balancing the graph.

3.2 Data Path Allocation

Data path allocation involves mapping operations onto functional units, assigning values to registers, and providing interconnections between operators and registers using buses and multiplexers. The decision to use ALUs instead of simple operators is also made at this time. The optimization goal is usually to minimize some objective function, such as

- total interconnect length,
- total register, bus driver and multiplexer cost, or
- critical path delays.

There may also be one or more constraints on the design which limit total area of the design, total throughput, or delay from input to output.

The techniques used to perform data path allocation can be classified into two types, *iterative/constructive*, and *global*. Iterative/constructive techniques assign elements one at a time, while global techniques find simultaneous solutions to a number of assignments at a time. Exhaustive search is an extreme case of a global solution technique. Iterative/Constructive techniques gen-

erally look at less of the search space than global techniques, and therefore are more efficient, but are less likely to find optimal solutions.

3.2.1 Iterative/Constructive Techniques Iterative/constructive techniques select an operation, value or interconnection to be assigned, make the assignment, and then iterate. The rules which determine the next operation, value or interconnect to be selected can vary from global rules, which examine many or all items before selecting one, to local selection rules, which select the items in a fixed order, usually as they occur in the data flow graph from inputs to outputs. Global selection involves selecting a candidate for assignment on the basis of some metric, for example taking the candidate that would add the minimum additional cost to the design. Hafer's data path allocator, the first RT synthesis program which dealt with TTL chips was iterative, and used local selection [9]. The DAA used a local criterion to select which element to assign next, but chose where to assign it on the basis of rules that encoded expert knowledge about the data path design of microprocessors. Once this knowledge base had been tested and improved through repeated interviews with designers, the DAA was able to produce much cleaner data paths than when it began [13 pages 26-31]. EMUCS [10] used a global selection criterion, based on minimizing both the number of functional units and registers and the multiplexing needed, to choose the next element to assign and where to assign it. The Elf system also sought to minimize interconnect, but used a local selection criterion. The REAL program [15] separated out register allocation and performed it after scheduling, but prior to operator and interconnect allocation. REAL is constructive, and selects the earliest value to assign at each step, sharing registers among values whenever possible.

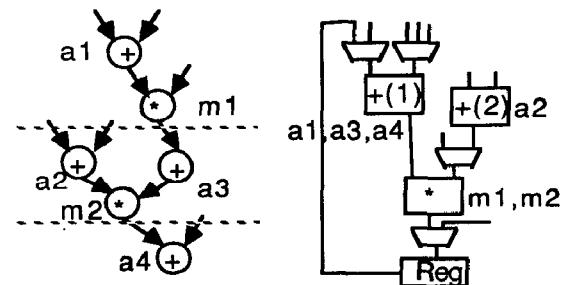


Figure 6. Greedy Data Path Allocation

An example of greedy allocation is shown in fig. 6. The dataflow graph on the left is processed from earliest time step to latest. Operators, registers and interconnect are allocated for each time step in sequence. Thus, the selection rule is local, and the allocation constructive. Assignments are made so as to minimize interconnect. In the case shown in the figure, a2 was assigned to adder2, since the increase in multiplexing cost required by that allocation was zero. a4 was assigned to adder1 because there was already a connection from the register to that adder. Other variations are possible, each with different multiplexing costs. For example, if we had assigned a2 to adder1 and a4 to adder1 without checking for interconnection costs, then the final multiplexing would have been more expensive. A more global selection rule also could have been applied. For example, we could have selected the next item for allocation on the basis of minimization of cost increase. In this case, if we had already allocated a3 to adder2, then the next step would be to allocate a4 to the same adder, since they occur in different time steps, and the incremental cost of doing that assignment is less than assigning a2 to adder1.

3.2.2 Global Allocation Global allocation techniques include graph theoretic formulations and mathematical programming techniques. One popular graph theoretic formulation [28] involves creating graphs in which the elements to be assigned to hardware, whether they are operations, values, or interconnections, are represented by nodes, and there is an arc between two nodes if and only if the corresponding elements can share the same hardware. The problem then becomes one of finding those sets of nodes in the graph all of whose members are connected to one another, since all of the elements in such a set can share the same hardware without conflict. This is the so-called clique finding problem. If the objective is to minimize the number of hardware units, then we would want to find the minimal number of cliques that cover the graph, or, to put it another way, to find the maximal cliques in the graph. Unfortunately, finding the maximal cliques in a graph is an NP-hard problem, so in practice, greedy heuristics are employed. Figure 7 shows the graph of operations from the example shown in Figure 6. One clique is highlighted, showing that the three operations can share the same adder, just as in the greedy example.



Figure 7. Example of a Clique

Formulation of allocation as a mathematical programming problem involves creating a variable for each possible assignment of an operation, register or interconnection to a hardware element. The variable is one if the assignment is made and zero if it is not. Constraints must be formulated that guarantee that each element must be assigned to one and only one element, and so on. The objective then is to find a valid solution that minimizes some cost function. Finding an optimal solution requires exhaustive search, which is very expensive. This was done by Hafer on a small example [9], and recent research by Hafer indicates that heuristics can be used to reduce the search space, so that larger examples can be considered.

3.3 Summary

The one issue plaguing synthesis researchers is how to reduce the computations required during the entire data path synthesis task, while still obtaining good designs. The goal of most of the techniques described above is to process the search space efficiently, and produce a near-optimal solution. The techniques described here perform well for each synthesis task in isolation, but the problem of finding good solutions to all tasks simultaneously is still an open one.

Two interesting, different approaches to cutting down the search space have been investigated recently. The first of these is the use of expert knowledge to guide the design process. The DAA was the first expert system which performed data path synthesis. The second approach is to narrow the problem domain so that more domain-specific knowledge can be used. The digital signal processing domain has been explored by several groups, for example. The CATHEDRAL system [7] is an example of a very successful effort in that area. Other programs have been able to get very good results by focusing on microprocessor design, for example the SUGAR system [24]. Synthesis of pipelined data paths is a design domain which has now been characterized by a foundation of theory [20] and implemented by the program Sehwa.

4. Open problems

There are a number of open problems yet to be solved in the synthesis area.

Human factors refers to the place of the designer in the design process. Some of the important issues yet to be settled are how the designer is to input design specifications and constraints, how the system is to output results, what decisions the designer should make and what information the designer needs in order to make them, and how the system is to explain to the user what is going on during the design process. Some user interaction is now allowed with EMUCS [19], and user interaction is used to guide the search in Sehwa. Mimola supports user interaction, particularly in restricting resources. These efforts, however, are only the beginning of research involving this aspect of synthesis.

Design verification involves the proof that a detailed design implements the exact design stated in the specification. This might involve verifying the design produced by the system against the initial specification, or it could mean verifying the synthesis process itself by showing that each step in the synthesis process preserves the behavior of the initial specification. McFarland and Parker [18] have used formal methods to verify a number of the optimizing transformations used at the beginning of the synthesis process, but much more needs to be done in this area.

Integrating levels of design means a number of things. For one, it means performing physical design, including floorplanning, along with the synthesis of the logical structure, as the BUD program does. Second, to make realistic evaluations of design tradeoffs at the algorithmic and register transfer levels, it is necessary to be able to anticipate what the lower level tools will do. Estimation of performance and area at the layout level is performed by BUD, and PLEST [14] performs area estimation, but more research on this topic is needed. Third, integration across design levels means maintaining a single representation which contains all levels of design information, as the ADAM Design Data Structure does [12]. Finally, it should be possible to allow parts of a design to exist at a particular time at different levels, which no current synthesis system performs to any degree.

A number of design tasks are still open problems. Interface design and handling local timing constraints have been researched by Nestor [19] and Borriello [3], with many problems still to be solved. System-level issues like trading off complexity between the control and the data paths and breaking a system into interacting asynchronous processes are new problems. High level transformations on the behavior have been classified and studied, but when to apply the transforms and in what order is an open problem. Trickey found an effective way of ordering and searching through a limited number of transformations [27]; but it is not clear that that method would generalize to a richer and less orderly set of transformations.

In summary, high-level synthesis defined as an abstract, limited problem of scheduling and allocation is well-understood, and there are a variety of effective techniques that have been applied to it. However, when it is seen in its real context, opening up such issues as specification and designer intervention, the need to handle complex timing constraints, and the relation of synthesis to the overall design and fabrication process, there are still many unanswered questions. Much work needs to be done before synthesis becomes really practical.

REFERENCES

1. Barbacci, M.R. Automated Exploration of the Design Space for Register Transfer (RT) Systems. PhD Thesis, Carnegie-Mellon University, 1973.
2. Barbacci, M.R. Instruction Set Processor Specifications (ISPS): The Notation and its Applications. *IEEE Transactions on Computers C-30*, 1 (January, 1981), 24-40.
3. Borriello, G. and Katz, R.H. Synthesis and Optimization of Interface Transducer Logic. *Proceedings of the International Conference on Computer-Aided Design* (November 9, 1987), 274-277.
4. Brayton, R.K., Camposano, R., DeMicheli, G., Otten, R.H.J.M., and vanEijndhoven, J. The Yorktown Silicon Compiler. In *Silicon Compilation*, D.D. Gajski, Ed. Addison-Wesley, Reading, MA, 1988, pp. 204-311.
5. Brewer, F.D. and Gajski, D.D. Knowledge Based Control in Micro-Architecture Design. In *Proceedings of the 24th Design Automation Conference*, ACM and IEEE, June, 1987, pp. 203-209.
6. Davidson, S., Landskov, D., Shriver, B.D., and Mallett, P.W. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers C-30*, 7 (July, 1981), 460-477.
7. DeMan, H., Rabaey, J., Six, P., and Claesen, L. Cathedral II: A Silicon Compiler for Digital Signal Processing. *IEEE Design and Test* 3, 6 (December, 1986), 13-25.
8. Girczyc, E.F. Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions. PhD Thesis, Carleton University, July, 1984.
9. Hafer, L.J. and Parker, A.C. Register-Transfer Level Digital Design Automation: The Allocation Process. In *Proceedings of the 15th Design Automation Conference*, ACM and IEEE, June, 1978, pp. 213-219.
10. Hitchcock, C.Y. and Thomas, D.E. A Method of Automatic Data Path Synthesis. In *Proceedings of the 20th Design Automation Conference*, ACM and IEEE, June, 1983, pp. 484-489.
11. Johnson, S.D. Synthesis of Digital Designs from Recursion Equations. PhD Thesis, Indiana University, 1984. MIT Press.
12. Knapp, D., Granacki, J., and Parker, A.C. An Expert Synthesis System. In *Proceedings of the International Conference on Computer-aided Design*, ACM and IEEE, September, 1984, pp. 419-24.
13. Kowalski, T.J. *An Artificial Intelligence Approach to VLSI Design*. Kluwer Academic Publishers, Boston, 1985.
14. Kurdahi, F.J. and Parker, A.C. PLEST: A Program for Area Estimation of VLSI Integrated Circuits. In *Proceedings of the 23rd Design Automation Conference*, ACM and IEEE, June, 1986, pp. 467-473.
15. Kurdahi, F.J. and Parker, A.C. REAL: A Program for REGISTER ALLOCATION. In *Proceedings of the 24th Design Automation Conference*, ACM and IEEE, June, 1987, pp. 210-215.
16. McFarland, M.C. The VT: A Database for Automated Digital Design. DRC-01-4-80, Design Research Center, Carnegie-Mellon University, December, 1978.
17. McFarland, M.C. Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. In *Proceedings of the 23rd Design Automation Conference*, IEEE and ACM, June, 1986.
18. McFarland, M.C. and Parker, A.C. An Abstract Model of Behavior for Hardware Descriptions. *IEEE Transactions on Computers C-32*, 7 (July, 1983), 621-36.
19. Nestor, J.A. Specification & Synthesis of Digital Systems with Interfaces. CMUCAD-87-10, Department of Electrical and Computer Engineering, Carnegie-Mellon Uy, April, 1987.
20. Park, N. and Parker, A.C. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on Computer-Aided Design of Digital Circuits and Systems* 7, 3 (March, 1988), 356-370.
21. Parker, A.C., Pizarro, J., and Mlinar, M. MAHA: A Program for Datapath Synthesis. In *Proceedings of the 23rd Design Automation Conference*, ACM and IEEE, June, 1986, pp. 461-466.
22. Paulin, P.G. and Knight, J.P. Force-Directed Scheduling in Automatic Data Path Synthesis. In *Proceedings of the 24th Design Automation Conference*, ACM and IEEE, June, 1987, pp. 195-202.
23. Peng, Z. Synthesis of VLSI Systems with the CAMAD Design Aid. In *Proceedings of the 23rd Design Automation Conference*, IEEE and ACM, June, 1986, pp. 278-284.
24. Rajan, J.V. and Thomas, D.E. Synthesis by Delayed Binding of Decisions. In *Proceedings of the 22nd Design Automation Conference*, ACM and IEEE, June, 1985, pp. 367-73.
25. Rosenstiel, W. and Camposano, R. Synthesizing Circuits from Behavioral Level Specifications. In *Proceedings of the 7th International Conference on Computer Hardware Description Languages and their Applications*, C. Koomen and T. Moto-oka, Eds., North-Holland, August, 1985, pp. 391-402.
26. Snow, E.A., Siewiorek, D.P., and Thomas, D.E. A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs. In *Proceedings of the 15th Design Automation Conference*, ACM and IEEE, 1978, pp. 220-226.
27. Trickey, H. Flamel: A High-Level Hardware Compiler. *IEEE Transactions on CAD CAD-6*, 2 (March, 1987), 259-269.
28. Tseng, C. and Siewiorek, D.P. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-5*, 3 (July, 1986), 379-395.
29. Zimmermann, G. MDS—The Mimola Design Method. *Journal of Digital Systems* 4, 3 (1980), 337-369.