Chippe: A System for Constraint Driven Behavioral Synthesis

by

Forrest Brewer
Daniel Gajski

Information and Computer Science Department
University of California, Irvine
Irvine, CA 92717

TR No. 88-09 April 2, 1988

Abstract: This report describes the Chippe system, gives some background previous work and describes several sample design runs of the system. Also presented are the sources of the design tradeoffs used by Chippe, an overview of the internal design model, and experiences using the system.

TABLE OF CONTENTS

1. Introduction

Several design systems have appeared which act as design aids to an engineer. In these systems the basic tasks of refining the design are done semi-automatically, with the engineer performing supervisory monitoring. When constraints are not met or one of the design tasks fails to complete, the engineer is required to manually modify the design to help fit the desired goals. The design aids enable the engineer to perform the design task more rapidly than before, but there is no automated method for closing the design loop.

To build a closed loop design system there has to be a method of comparing the results of designs with the desired behavior and constraints. A simple method is to build evaluators that, given the desired function and the design, measure the performance in terms of the higher-level functional description. This allows simple and direct interpretation of the design results vs. the goals. A hierarchical design system would then have evaluators which determine the quality of the design on a given level and abstract this information to allow interpretation at the higher level.

These evaluations must then be compared with the high-level design goals and appropriate action taken. There are several possibilities for the "appropriate action" in a closed loop design. The design goals can be changed (reallocation of resources), the design itself can be modified (optimization), or the constraints to the design refinment tools can be changed. Modification of the goals opens the possibility that the present level design will not meet the constraints of the higher level design or of the system as a whole. Direct modification of the design requires a great deal of knowledge about the various available options and how each is implemented. This approach results in either extremely complex algorithms or in simplified design models. The third possibility requires the design of specialized design tools capable of design within a wide variety of constraints. Such tools can allow a much broader scope of designs by performing constrained refinement of the designs rather than direct design optimization. [Pang88] In this scheme, the designs are optimized by changing the constraints of the refinement tools in response to evaluations of the design. [BrGa86] [GaBr87] This approach has the advantage that the redesign is handled by the refiner itself so that the knowledge about design required to correctly refine the design is separated from the design knowledge of how to adjust the design to meet constraints.

In this paper we will first outline some previous work in this field and the relative advantages and problems encountered. Then the Chippe design models and sources for its directed tradeoffs are discussed. An overview of the implementation of the Chippe system is presented next followed by an operational description of the system, some experimental results and finally experiences with this system.

2. Previous Work

There are several other micro-architecture design systems; of these only a few attempt to design within preset constraints. In this chapter we present a short exposition on the design methodologies of each.

2.1. Algorithms to Silicon Project

The Algorithms to Silicon project is primarily a collation of three separate projects BUD [McFa86] , FRED [Wolf86] , and DAA. [Kowa84] BUD is a system that performs global analysis of the behavioral requirements for a design from the ISPS [Barb81] behavioral description. It is used with FRED to augment the design capabilities of the DAA system. BUD provides the design system's resource allocation and scheduling facilities, and the ability to design within constraints. BUD's inputs consist of the *Value-Trace* [Snow78] a behavioral representation, a list of branching probabilities from which operation probabilities can be generated, and a cost function to be minimized. The intent is to place the design constraints in the cost function and then use the probabilities and the operation requirements to search for an allocation and schedule which minimize this cost.

BUD uses a heuristic solution to this problem. First it builds a cluster tree of function units. The clustering metric is based on the cost of merging operations, the number of common data-flow sources and whether the operations can be performed in parallel. Then the tree is cut at various distances from the root and each cut provides a new hardware allocation. Each prototype allocation is analyzed and compared to the requirements. This is continued exhaustively for each possible cut of the partition tree. The allocation which best matches the constraints is then passed to DAA for completion. This has the effect of searching a much larger space of possible allocations since the clustering tree admits only those partitions favored by the clustering metric.

FRED supports BUD by providing an object oriented data-base of design components. The components are described not only by attributes but by procedural methods which allow calculation of component parameters from various partial descriptions.

DAA is an expert system which takes the hardware allocation and schedule and creates the interconnect for the design. Prior to the advent of BUD, DAA worked directly from the VT description. The difficulty in creating designs meeting specific constraints led to the incorporation of BUD's global analysis to aid the design process.

2.2. ADAM

ADAM [Knap86] [Knap87] is a high level planning system designed to control several synthesis tools. Its input is a set of constraints and a Flow-Graph[1] representation of the design. It uses these along with rules about the tools to complete a design plan of operations of the tools and settings of parameters which will complete the design. The planning phase is carried out in an abstract design space using estimators to evaluate the design as each of the (abstract) tools in the plan is applied. The constraints used in ADAM differ significantly from those of the other systems considered. In ADAM, high level decisions about the design such as pipelining, the system clock, and design technology are entered as design constraints. ADAM then tries to create a design plan consisting of arguments to the design tools and an ordered sequence of tools to run. The plan is constructed using a knowledge base of properties based on constraint implications, and abstract estimator models for the various tools. To facilitate the planning process, the estimators are all intentionally monotone which increases the planning efficiency. Once completed, the plan could then be executed by running the tools on the input data-flow graph and supplying the the tools the arguments set by the planner.

[1]A behavior representation similar to an ASM chart or VT body.

Because of the use of monotone estimators in the planning process, the actual design produced may not match the constraints of the original design. In particular, the design estimates cannot take into account the constraints of the real design space since these constraints are not known during the planning phase. It was proposed that once a design had been completed, the design could be evaluated and these results fed back to the planner to annotate the estimates used in the planning stage.

2.3. Cathedral

Cathedral [DeRS86] [Raba87] [Catt87] is a system specialized for designing digital signal processing chips. Specifically, Cathedral-II is designed to implement multi-processor chips with a regular interconnect and synchronous data-passing protocol. This top level constraint forces certain design decisions in the lower level scheduling and chip design. Among these is the design of modules which all execute in a given fixed clock cycle. These modules are mapped to the input data-flow graph by an expert system trying to meet cycle requirements. Since the clock is fixed, and the operator class is small, a large amount of time can be well spent in the scheduler phase to perform the design in the smallest amount of time on the mapped hardware. If the design passes the cycle count limit, then final symbolic microcode and control units are designed. In the case of failure, a user is given the present state of the design and is allowed to make "pragmas" or assertions to the expert mapper. These assertions allow the user to steer the system toward desired architectural goals that the user thinks will improve the performance.

Cathedral has several levels of optimization which are all aimed at reducing the final cycle count or equivalently, maximizing the performance. Since the design space is limited to digital signal processors, these optimizations are not difficult to make. That is, the properties of the controller, the execution modules and the input schedule are all known in advance of the design process. This allows a great deal of fine tuning for the particular problems to be designed. At present such systems (those taking a small segment of the design space) are the most successful in performing high level synthesis. It is hoped that the basic idea of design styles can allow similar fine tuned strategies in more general problems. Even in this system, however, there is not direct evaluation of the design or method which can make use of the earlier design efforts other than manually.

2.4. HAL

HAL [PaKG86] [Paul87] is a time-constrained micro-architecture design system. It is composed of three procedural phases and a data-base. The system input is a Flow-Graph of the behavior and a constraint on how long the graph execution should take. In the first phase the graph is scheduled into the time constraint (if possible) and the operations are scheduled by use of a force-directed heuristic. In this scheduler, the mobility of operations off the critical path is modeled by a probability for the operation to be scheduled in a particular cycle. Then when the graph is scheduled, the sums of the probabilities acted as "forces" which are balanced to determine the clock-cycle in which the operation is actually scheduled.

In the second phase, the scheduled graph is allocated physical hardware using an expert system. The allocation is to minimize the interconnection and mux costs while using as little hardware as possible. The allocation is done sequentially, each invocation allocating a new hardware device to the design.

Finally, the interconnect for the design and the final binding of operation to unit is done. The operations are assigned to units to reduce interconnect using the information in the DFG. Then storage operations are added to the graph as needed by the schedule. Clique partitioning is used to cluster the storage operations and function units into clusters which are then allocated to the design. Thus, in the HAL system the driving force is the number of clock cycles allowed in the execution of the given design.

2.5. Flamel

Flamel [Tric87] is a design system which specializes in the modification of the control structure of the behavioral description. Flamel's input is Pascal (with fewer operators). Flamel then builds a structure of control operators and blocks of straight line code. This structure is modified by control-flow operations to increase the parallelism available for the design. In Flamel only one transformation is applicable to the control-flow graph at a time. This is a consequence of the original Pascal structured input, and heuristic rules. The object is to reduce an interior sub-graph of the control-flow to a single block increasing the available parallelism to the scheduler. This is always possible if the control structures are limited to 'if' constructs and constant iteration loops. These modifications result in a sequence of equivalent flow-graphs, each with fewer control blocks. Internally, each data-flow graph is flattened to extract the maximal parallelism available.

Each graph generated in this way corresponds to a potential design which can be evaluated for area and performance. This is done by allocating hardware to each operation in the graph, and then merging the hardware until the area constraint is reached. First, exclusive[2] units are merged and then other compatible units are merged while adding more states to the schedule of operations. The data-path is then designed in a bit-slice style. The placement of the bit-slice elements is made using a Kernighan and Lin style clustering algorithm. From this layout area and time estimates can be directly estimated.

The blocks of the original control graph form the leaves of a clustering tree. As the transformations are applied, the new blocks formed are placed into the tree as the parents of the blocks that formed them. This process is continued until no further transformations can be applied. Each node is then evaluated by the method above. Finally, given the desired constraints, Flamel assigns resource constraints and starting with the top of the tree recursively searches till the fastest implementation within the resources is found. This procedure finds the best global design of those designs within the tree.

2.6. Limitations

Several of the difficulties encountered in these systems are common to all. Here we summarize those problems which are addressed by the design process model.

a) Several of the above systems (most notably Flamel) make use of limited design models to simplify the algorithms and allow a simple strategy for coercing the design closer to the tradeoffs. In Flamel the operations are limited to allow simple code re-structuring, the control structures are limited to constant iteration loops etc. This has the effect of simplifying the design tradeoffs to the point where simple heuristic ordering of modifications can build all of the potential designs. Another common assumption is the use of unit time scheduling. Unit time scheduling would allocate the same time for

[2]Units whose operations are scheduled in different clocks.

a bit-wise AND operation as for a parallel multiply.

b) Other limitations on the design models are common, for example, restricted control unit design. These limitations correspond to choosing a single design style to implement the control design. The advantages of designing within a limited model are speed and simplicity of the design algorithm. Several systems are designed for special applications by enforcing a single style of implementation of the final design. The Cathedral system is specialized for signal processing while SYCO [JVJC86] and the original DAA system are tuned for microprocessor design. These system level design limitations result in simplification of the design strategy since the direction of many design tradeoffs is predefined by the imposed design style.

c) Nearly all of the above systems run in an open loop manner. After the selected design is complete none of the systems evaluate the design to see if it actually met the constraints. More importantly, if it did fail, none of the systems has a method for fixing the final design. In BUD, ADAM the global analysis is done before the design is implemented using estimations of the component and interconnect constraints. Thus the prototype designs are selected on the basis of estimated values and then implemented. For such a scheme to work either the estimations must be very good or the design model must not allow small changes to cause large performance differences. In most of the systems all hardware resources are set prior to the scheduling or interconnection phases and cannot be changed. Thus most of the possible design tradeoffs are made very early in the process, when there is little data available to make such decisions. In effect, this places the entire success or failure of the design on the ability of the allocation algorithm to correctly determine the hardware needed before the design is built.

d) The reason that these systems choose not to use iterative design is the large amount of design time spent completing a design. The design tools have been carefully crafted to make the most of their input and exhaustively search for the best solutions. This is especially true of the ADAM tools: Sehwa, MAHA. [PaPa86] [PaPM86] However, the design estimations on which the time allocations to these tools are based may not be accurate enough to ensure that this time is being spent on the appropriate design. Especially when using feedback to correct the designs, a method for obtaining "cheap and dirty" designs is necessary. If these designs are produced quickly then they can be evaluated directly, obviating the need for better estimation. After the design goals are approximated, the design can be optimized to a better degree by judicious optimizations.

3. Chippe Design Model

3.1. Requirements for Micro-Architecture Design

The Micro-architecture design problem starts with a behavior level description of a machine and produces a register transfer level design with modules, control units, and appropriate interconnection between the modules. This design problem contains many tradeoffs and design decisions such as: number and type of functional modules, control unit type, clock frequency, interconnection style, and register allocations. Each function described in the behavioral specification must be represented in the modules, but the number of modules and the achieved degree of parallelism is determined by tradeoffs. The basic tasks in this design process are: creation of a schedule of operations, allocation of the modules, registers, and busses, binding of operation to unit, allocation of connections between the modules, and creation of the control unit or units. All of these tasks are interdependent and for this reason micro-architecture makes a good test bed for the new design model.

3.2. Refinement Tasks

The total micro-architecture design process can be subdivided into four weakly coupled tasks. These are: allocation of the control unit and the data-path, scheduling the operations, building the interconnect, and performing first-cut layout. The 'weak' coupling of these tasks merely means that by carefully constraining each task based on the results of others, reasonable results can be achieved. In general, better results can be had by merging several of these processes and performing them simultaneously. However, the complexity of the resulting tasks may make unfortunate design time vs. design quality tradeoffs.

Allocation refers to the task of selecting the hardware resources (i.e. function units) that perform the functional operations. In addition the allocation task must select an appropriate control unit for the design. These selections comprise a large part of the design systems total tradeoff potential for area vs. time. Most other hardware compilers [DeRS86] [PaKG86] [PaPa86] actually bind the operations to the units in this task. This simplifies the scheduling and interconnect tasks at the cost of poorer design quality. More importantly, these systems bind hardware to match a pre-defined schedule of operations, severly limiting the opportunity for directed resource tradeoffs.

Scheduling takes the operations in the CDFG and determines the time slot for each. It must necessarily take care of all dependencies, the operation time of each unit, and the clock cycle time while trying to minimize the total time used to realize the desired behavior. Since the schedule is resource based, the best schedule is the one which minimizes the number of operation cycles.

Interconnection seeks the minimal cost interconnect for each of the units and the registers under the constraint of matching the schedule. Several schemes have been described to perform this task [TsSi84] [PaKG86] [Kowa84] based on different styles of interconnect. In contrast to these systems the interconnection task here also performs the unit to operation binding. This is consistent with the resource based approach and allows for better results.

The *Layout* task refers to the first cut floor planning task used to refine the values of the area usage of busses and function units. At present the units have area and time bounds which are simply added, but for real chips with two dimensional constraints, a better model is needed.

To design within the constraints determined by the design process model, the refinment and optimization tools must meet strong requirements. Specifically, they must allow constraints of resources and global parameters which control the design, and they must allow completion of partial designs. For example, the scheduler must allow changes in the design and number of components that it can use, or constraints on the clock cycle time for the design. This has the effect of allowing simple constraint decisions to force the design into different design tradeoff regimes. The requirement of ability to deal with partial solutions stems from the iterative nature of the design. While it is possible the simply rebuild the entire design from scratch after a modification this is inefficient since often much of the design is not effected by such a change.

3.3. Module, Timing, and Control Models

The scope of this system is the design of modules with defined external communication protocols. These modules can be represented as finite state machines with the proviso that the state space may be very large. Our model presented here is restricted to the design of single modules within the constraints of communication and the system imposed physical constraints. The communication at the module level is assumed to be part of the behavior of the module, that is, the communication protocol is described directly in the behavioral language.

The modules themselves are split internally into control and data-path sections. Generally, the control section is concerned with the sequencing of operations over time, while the data path provides the necessary hardware to implement the functions. This partitioning need not be enforced, however, as several key tradeoffs arise from the selective transfer of suitable operations between these sections. Figure 1 shows the dependence of the two sections in the model. The control is assumed to transform its input variables, the state, input signals, and condition signals into a new state and control outputs for the data-path. The data path similarly transforms its own data and control signals to new data and condition codes for the controller. The registers in the state and data loops are necessary to the model to prevent races in operations. The other registers are added only if appropriate for the style of design. This model does not define the time (or number of states) required to complete a cycle, just that the data is stored appropriately on clock transitions. The only requirements are that the control output a valid control signal to the data-path on each cycle, and that the internal storage of these sections (pipelines etc.) be scheduled accordingly. To assure that the time dependence of the control is mapped into the schedule,

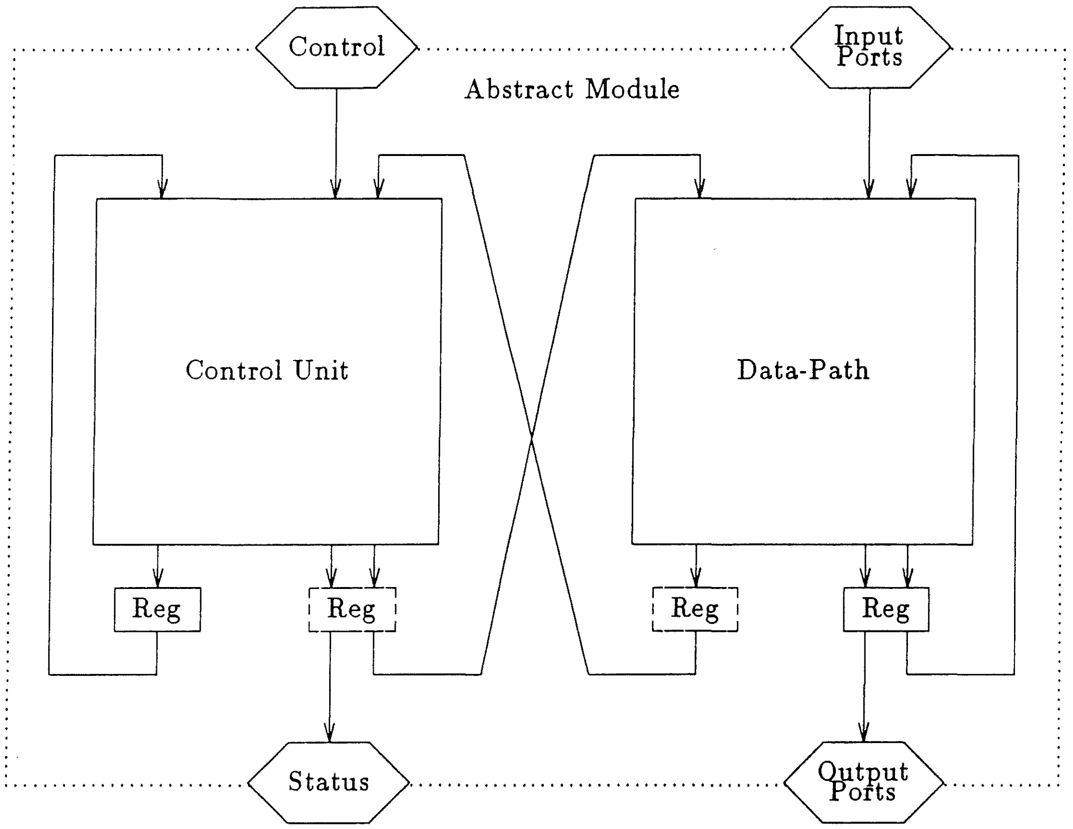


Figure 1. Abstract Module Model

either the scheduler must directly include the control constraints or they must be inserted into the compiled behavior. To see how this is done it is first necessary to discuss the abstract behavioral model.

The behavior specified in the input must be put in a form suited to the design problem. Commonly, this information is represented as a control-data flow graph (CDFG). This corresponds to the *Value Trace* of the CMU efforts [McFa86] [Snow78] and closely to the CFG used by Trickey. [Tric87] The structure of the CDFG comes from the structured programming paradigm. The behavior is organized into blocks corresponding to conditional control state transitions. These blocks are interconnected by directed arcs representing possible successor blocks. There is no limit to the number of possible successor arcs[3], or where those arcs may terminate. Loops are represented by cycles in the control flow. In this way loops are unwound and conditional execution of future blocks is explicitly noted. Thus, the entire behavior is represented as straight line (sequential) sections which are connected to the possible successor blocks by allowed control transitions. To insure correct dependence handling there are two requirements. First, the block transitions must occur on a state timing transition. Second, all of the values communicated to other blocks must be stored into an ordered set of registers at the end of each block. This allows looping of a block onto itself, the values are assumed to be in the appropriate registers. These requirements allow the separate scheduling of each block in the graph as long as the global value storage requirements have been met.

Figure 2 shows the interaction of control and state timing in the model. Because of the need to support several control styles, the model needs several methods for constraining the schedule. After each state transition the control may have a delay before the signals are valid. This is modeled as adding time to start of the cycle before any operations can fire. This time is denoted the control setup time. In addition the control may require additional delays or (in the case of pipelined control) state transitions to calculate the successor block in a conditional control transfer. In this case the constraints are modeled as delay operations[4] which are scheduled into the graph. This has the effect of pushing the operation producing the condition higher in the data flow graph, hopefully allowing earlier execution. If the condition producing operation is on the critical path then the scheduler will

[3]Actually, a particular control unit style may require a limit to the number of arcs since it may be impossible for the unit to generate an arbitrary number of possible 'next state' addresses.

[4]A delay operation is a method of forcing the scheduler to wait a predetermined time before scheduling the successor operations.

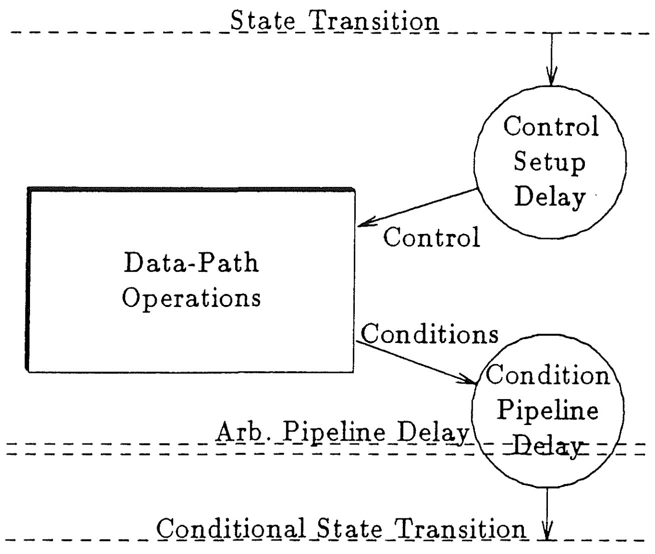


Figure 2. Control Timing

add states (possible no-ops) to satisfy the timing constraints.

Figure 3 shows the operation schedule timing. Each operation in the behavior is bound to a function unit which physically performs the operations. This binding determines the time required for execution of the operation. The scheduler uses this information and the dependencies to select which operations fire in which particular state. The timing model allows for operations which extend beyond a single clock cycle, and for pipelined operations. In addition, if there is sufficient time in a cycle, the model provides for direct execution of a unit on the completion of another. This is referred to as "Operation Chaining". [PaGa86] To make correct schedules, the units which are scheduled across state transitions (multi-clock operations) must have either internal storage, or an input latch to hold the data constant for the extended period. Failing either of these, the scheduler must write

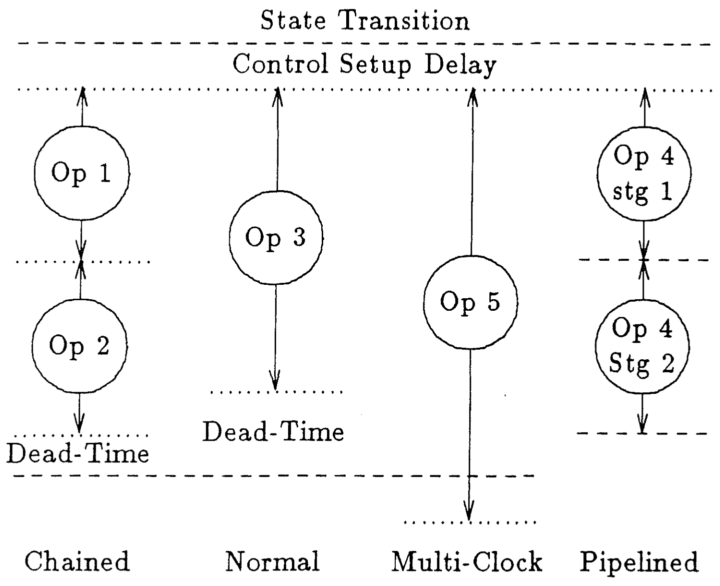


Figure 3. Operation Timing

the inputs of the operation for consecutive cycles until the operation is completed. (The scheduler will do this but it usually requires more interconnect and is incompatible with certain control and data-path styles).

All of the functions of the module are performed on function units and registers. A function unit is an implementation model of a digital circuit which performs operations that map to those required by operations in the behavior. A function unit can perform more than one operation and can have multiple inputs and outputs. Typical function units are, adders, ALUs, multiply, barrel shift, select (multiplexors), decoders, memory units etc. The functions performed are selected by the control inputs and (sometimes) by the previous state. Function units provide the ability to bind realizable physical unit models into the micro-architecture. Each operation performed on a unit can have its own execution time, both clock transitions, and absolute delays. This allows the representation of reconfigurable pipeline units. Finally, function units are modeled with arbitrary internal storage to allow pipelining and memory operations. This storage is separate from the registers introduced to store variables across state transitions, but must be similarly modeled in the scheduler.

A microarchitecture consists of a set of function units, registers, and interconnect. The model for interconnect allows both multiplexor and bus based connection. This is accomplished by using a parameterized two-level interconnect scheme. The outputs of the devices are connected to a connection matrix which further connects to the busses. The busses in turn connect to another matrix which connects to the inputs of the devices. The matrix connections are realized as simple representations of multiplexors, by counting the number of inputs on each bus. This scheme admits several cost functions based on bus number, input mux number, output mux number, total number of connections or combinations of the above. Using this model it is a simple matter to define a cost function which follows the schedule and minimizes the desired quantity. [Pang87]

In previous register-transfer level synthesis tasks very little attention was placed on the importance of layout and geometric considerations on the design. An exception to this is made by the BUD system which does have a notion of placement via physical allocation of clusters. Without some model for interconnect costs and geometric constraints, a design system will consistently underestimate design costs. An example of this problem is the merging of two similar exclusive functions in a design. Without layout cost estimation, the added cost of the interconnect bussing to move the operands to the new unit cannot be estimated. This cost may be larger than the area gain from the merge. In Chippe the interconnect delay is obtained from knowledge of the bus loading and a worst case estimate of the bus length. This estimate is obtained by assuming a water-filling placement for the function units.

3.4. Design Tradeoffs

There are several sources of design tradeoffs in computer architecture synthesis, some are related to the instantiation of the operations, the remainder come from re-interpretation of the behavior. Operator instantiation includes resource allocation, setting of global parameters, and control style selection. By interpretation of behavior, we admit tradeoffs which depend on changes of the representation of the operands or on the interpretation of the control or data operations. Such tradeoffs include modifications of the control structure of the graph (re-interpreting the sequential behavior), algebraic manipulations of the operations to allow fewer operations or to increase the available parallelism, and direct changes in the representations of operand used to emulate the desired behavior.

Resource allocation includes the selection of the number and type of the units which do the operations, number and type and style of busses for the interconnect, and style desired for the control unit. Since we advocate resource based control of the design, changing the number of a certain operator may change the parallelism of the data-flow graph and hence the performance of the architecture. Changing the type of operator includes modifications of the timing by adding latches to simplify the communication, pipelining the unit to increase the parallel throughput, or selecting different implementations of the units to change the combinatorial delay. For example, a 32-bit ALU used in an address calculation can be implemented as one of several carry-lookahead or precharged options. Often, if the address calculation is not on the critical path, area can be saved by using a slower adder with smaller area. This improves the area usage with no penality in performance. As an aside, it is easy to see how such a modification is done in the iterative design paradigm since the design itself is available. However, it is extremely difficult to add this kind of optimization to non-iterative design methods. Initial global analysis might point out the need for two adders, but unless the scheduling is redone after the change it is likely that both adders will appear on the critical path.

Other less direct resource constraints include limiting the number of busses. The bus limits described earlier allow a much greater span of design styles by enforcing bus limits at the scheduling level. If this is not done, then the minimal number of busses is determined by the number of simultaneous arcs crossing the state transitions in the schedule. Since it is otherwise to aim of the scheduler to parallelize the operations as much as possible, designs without bus constraints will all be 'connection heavy'. It is difficult to evaluate the importance of bus constraints without performing a floorplan to at least determine the bus lengths. Once this is done, constraints can be fed back into the scheduler and interconnect tools to better accommodate the design goals. It would be still better to perform interconnect and layout simultaneously. This would allow direct application of the constraints, but this task would also require simultaneous modifications to the schedule and so would be exceedingly complex. It is hoped that by style based control of the design processes, relatively good designs can be created which maximize the options of the later tools, so that comparable refinement can be made iteratively.

Control Style selections are resource selections of a simpler type. Instead of manipulating the resources available for the control unit design, we simply select a particular style of control based on the constraints. Then the constraints for this style of control are communicated to the scheduler and the control unit is implemented directly. This restriction on the types of control unit stems from fairly incomplete design knowledge about control in general. Ideally, the control and data-path could be designed together from the desired behavior, but this would require a general model of control behavior and algorithms to partition the behavior and data path operations. A simpler model is to determine tradeoff regimes for several control styles. Then a parameterized model for control interaction with the data-path can be defined and used in the data-path design, allowing tradeoffs of each of the styles. These tradeoffs include ROM based or PLA based control, pipelined control with automatic no-op re-scheduling, Moore or Mealy machine control, and random logic control for smaller machines.

There are several parameters global to the design styles available in this model. Probably the most important is the system clock time. Other parameters include testability merit figures, and global style selections such as technology. Tradeoffs of the system clock are based on the timing model of the scheduler. Since the scheduler allows chaining of the operations on the critical path, long clock times are not necessarily bad. The effect of a faster clock is to reduce the granularity of the control operations. Thus faster clocks can support better timing of the operations. Those operations longer than the clock are simply allowed to extend into subsequent cycles, the inputs are either held or latched as required. This is at the cost of much greater power consumption, especially for CMOS technologies. Another problem is that for certain control styles, the control lines are not active for a significant period after the state transition. This further reduces the time available for operations to take place, and lowers the efficiency. Longer clocks can sometimes allow chaining of important operations, thus the performance loss may not be bad for larger, more parallel systems.

These design tradeoffs arise mainly from modifications of the CDFG of the design, several of the tradeoffs are standard compiler optimizations of the operations. [KKPL81] In the case of directed design, many of these modifications become tradeoffs instead of optimizations. For example, algebraic manipulation of the data-flow graph to minimize the tree height can now include addition of more parallel operations. This allows faster operation of the design at the cost of greater area and power consumption. Chief among these tradeoffs is the decision of whether a control transition should be handled sequentially in the control unit or in parallel on the data-path. Control block merging can greatly increase the parallelism available to the design, at the cost of greater numbers of operations. [Tric87] [Dutt88] [BrGa87] Given a particular control style, there is a family of design tradeoffs based on direct manipulation of control block partitions. Examples are block merging, formation of multiway branch constructs, loop unwinding and folding. Finally, there are tradeoffs which move operations in control expressions directly into the control. For example, a few status lines may be compared with a constant to determine future control, these status lines could be moved directly into the control using a latch. This trades the decoder area in the data-path and associated busses with the extra are required by the control. For small fast machines with random logic controllers, these tradeoffs are especially valuable.

4. System Overview

A prototype system to perform micro-architecture design has been implemented to study the issues mentioned above. This system (Chippe) implements the design model and several of the design tradeoff strategies discussed above. Figure 4 depicts the general structure of the Chippe system. Input comes from the Hardware Description Language, which in the present version of Chippe is similar to Pascal and ISPS. The language has a few

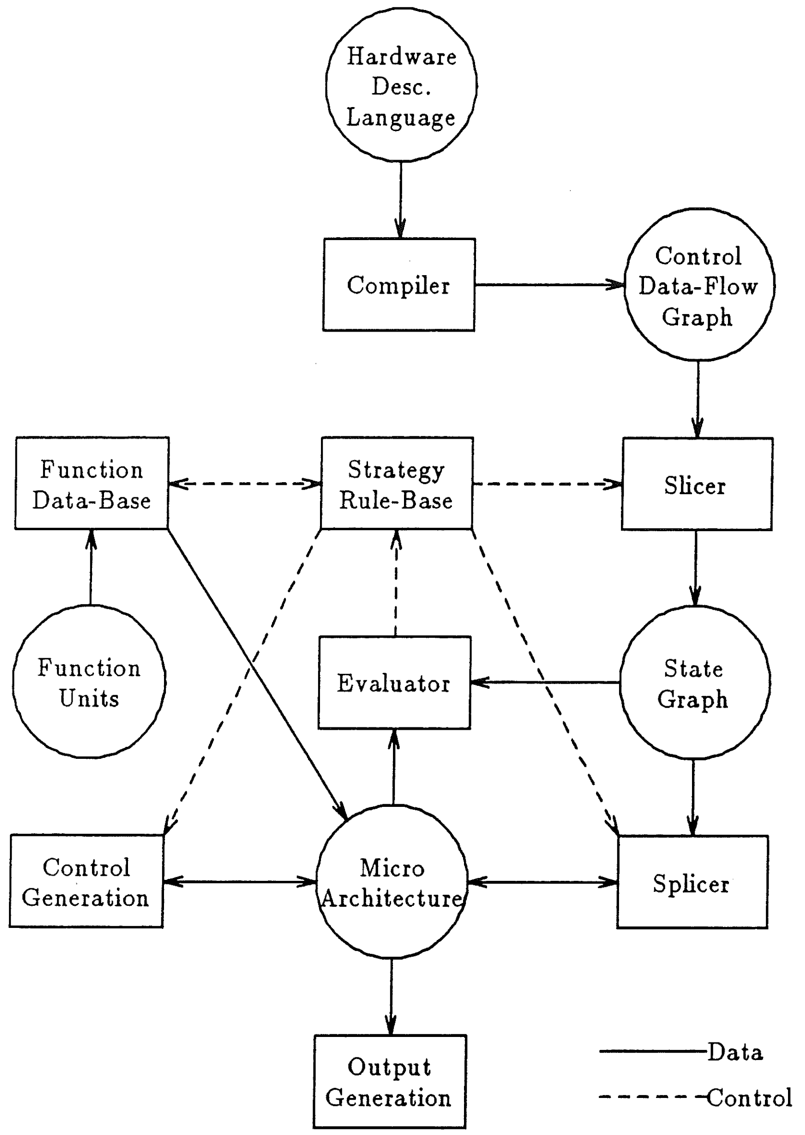


Figure 4. Chippe System Structure

extensions to allow description of I/O protocols and timing constraints, and many operators. The language is compiled into an internal control-data flow graph (CDFG) representation in two passes. The first pass builds a set of operators for the language and creates nodes for the local sections of straight line code. The second pass of the compiler creates the CDFG itself and adds the necessary dependency arcs. The second pass also adds registers for control block transitions and modifies the CDFG to accommodate the particular selected control style. At this point in the design the effect of control styles is to produce delay nodes between the condition codes generated in the data path and subsequent state block transitions. Modification of the control flow at this point (before the schedule) simplifies the correct design of the schedule by adding only the constraints produced by the control model. At this point the original CDFG is saved on a stack of potential designs. This allows backtracking earlier failed design tradeoffs.

Chippe represents the present state of design as both a CDFG and a "partial design" structure which keeps the parameters, function units, interconnect and registers. Design tradeoffs in Chippe result in modifications to these design structures. However, it is inefficient to completely redesign the entire structure after each change, so the design refinement is done in stages. After each stage the partial design is annotated with the results of the refinement. Then when a design tradeoff modification is desired, only those stages which need to be redone are performed. Evaluated partial designs are stacked to allow backtracking at certain decision points. However, the present general strategy favors greedy tradeoffs to save both space and design time. To simplify the above figure, the CDFG, state graph and partial design are drawn separately, however, they are all part of the design data structure stored for each stack element.

Slicer and Splicer [PaGa86] perform state scheduling and interconnection respectively for Chippe. Both of these processes are controlled by the expert via parameters in their activation. Their results are recorded in the partial design. These tasks allow design tradeoffs in the scheduling and interconnect for the designs. Scheduling tradeoffs are made by modifications of the resources available to the design and settings of the global parameters. In addition the scheduler must take into account the constraints from the particular control style selected. The interconnect tradeoffs include various interconnect cost functions and heuristic search orderings allowing a few distinct styles and ability to trade design quality vs. design time.

Control Unit generation and modifications to the state graph are performed by the control unit generation task. [Dutt86] This task is driven by the expert to selectively modify the control structure of the graph and the global control unit selection. The selected control unit style is accessed from the data base and all necessary constraints are added to the graph. The iterative design process allows the scheduler to use timings derived from the control model and the actual schedule of the data-path operations. This allows a more efficient schedule than that derived from a worst case analysis.

All of Chippe is controlled by the expert rule-base which determines the strategies and resource allocations for each of the other tasks. Its view of the design is based on direct examination of the CDFG and on execution of evaluation functions. The expert's design strategy follows from a controlled iterative approach. [BrGa86] In this approach all of the actual designing is performed by controlled algorithmic tools. These tools maintain correctness of the design and ensure that the behavior is preserved. Design tradeoffs are made by adjusting the controls to these tools. For example, the module resources available can be set and thus modify the action of the scheduler. This frees the task controlling the tools from need to understand how to make correct changes to the design. Similarly, by separating out all of the technology dependent design data into a separate data-base, the system can be made relatively technology independent. Thus, all the expert need to concern itself with is the analysis and evaluation of the design and determining a proper course for future modifications.

In Chippe, the expert first determines where the present design iteration is in terms of the constraints. The present constraints are area, time, and power. When one or more of these values is violated, a set of rules designed to correct the situation is searched for an appropriate change or set of changes to the next iteration. In this way the iterative design provides for opportunistic modification of the design structure, based on timely evaluations of their suitability. Figure 5 contains a table of the tradeoffs supported by the present

Tradeoff	Supported	How	Effect
Number of Units	yes	Alloc. Rules, Scheduler	More Units => more area, more perf.
Type of Units	yes	Alloc. Rules, Database	Local Area/Time of Unit
Pipeline Units	yes	Alloc. Rules, Database	Pipeline increases operator parallelism
Merged Units	yes	Alloc. Rules, Database	Trade Perf. for Area
Latched Units	yes	Interconnect Rules	Trade Unit area for Interconnect
Compiler Optims.	no	Need New Tool	Optimization Task
Operator Modification	some	Special Rules	Optimization Task
Macro Expansion	no	Need New Tool	Allow sharing of functions of operator
Flow Merging	yes	Transfuse	Increase graph parallelism and storage
Line Merge	yes	Transfuse	Increase graph parallelism
Loop Unwind	no	Need New Tool	Increase parallelism, generally complex
Layout Style	no	Need Layout Tool	Style to minimize area
Control Style	yes	Cogent	Style to fit Design Rqts.
Interconnect Style	yes	Splicer cost functions	Bus vs. Mux Tradeoff
Cycle Time	yes	Cycle Time Rules	Granularity vs. Power (dep. on Control)
Search Limits	yes	Splicer, Layout	Design Time Optimization

Figure 5. Design Tradeoffs Presently Supported in Chippe

implementation of Chippe.

The Evaluator is a set of routines which are interactively called by the expert to determine the state of the design and to focus attention on possible future design modifications. They analyze the partial design and return numeric quality measures. In a sense these functions provide the means for rational decisions in the expert system by performing global analysis on the present design. It is not sufficient to know that there is a problem in failing to satisfy a constraint, in addition, information about what is at fault and how to make appropriate changes is necessary. This information is provided by the specialized functions in the evaluator. [Brew88] Examples of representative functions are usage statistics for functions units, execution overlap measurements of units, clock dead time, and relative measures of resource usage by function units, multiplexors, busses, and the control unit. These measures are activated dynamically by the expert system as it searches for appropriate rules to apply.

The Function Unit database is a collection of units and operator bindings for the design. During physical resource allocation, the data base is queried about possible units to perform operations or groups of operations subject to certain parameters. For example, an adder for 13-bit operands will produce a list including both ripple, and carry lookahead adders, adder and complement units, and full ALU's. If the request was for an adder and logical op combination, only ALU's would satisfy the request. The database is parameterized for input latching, pipelining, bitwidth, speed etc. Finally, the database stores models of the components to allow evaluation of the design. At present the timing and state behavior, geometric and gate usage, and power dissipation are modeled.

Finally, the output generation and user interface routines allow interactive sessions with the system, and final output design production.

5. Experimental Results

To allow somewhat realistic evaluations of the designs produced by Chippe and also to allow reasonable design tradeoffs, the component data-base must contain models which mimic reality. It was thought that since the layout section of Chippe was the most rudimentary, structures built in gate-arrays would be most suitable for the designs. Without proper layouts, the bus loading and interconnection costs are difficult to estimate. To help this problem gate array cells are designed to have sufficient drive to run fairly long lines, while trading off the speed possible for very small loadings. Also, in gate array designs, the natural unit of space is a "gate" which simplifies the estimation problems for the data-base. This decision admittedly removes the potential for layout based evaluation and tradeoffs in the design, but was made to reduce the size, complexity, and turnaround time for the implementation.

5.1. Sample Design Walk-Through

To illustrate the operation of Chippe, the following simple design is presented and annotated at the key decision points in the design. Figure 6 shows the hardware description for a small fixed-point calculation loop. This particular test case is from Girczyk, Knight, and Paulin [PaKG86].

Figure 7 traces the evolution of the small design test case. The goals for the system were area$<$3000 gates and delay$<$1.0 uSec. These constraints are shown as the vertical

```
program diffeq(input,output);
type    integer = {0..11};
reg     three : integer;
        five  : integer;
var     a, dx, x, u, y, y1, u1, u2, u3, u4, u5, u6 : integer;
begin
   if (x < a) then
        repeat
           u1 := u * dx;
           u2 := five * x;
           u3 := three * y;
           y1 := u * dx;
           x  := x + dx;
           u4 := u1 * u2;
           u5 := dx * u3;
           y  := y + y1;
           u6 := u - u4;
           u  := u6 - u5;
        when x < a
end.
```

Figure 6. Hardware Desc. Language for Hal example

dashed box on the left side of the figure. The figure shows the general flow of the design, from larger to smaller designs. This is an artifact of the initialization rules which produce sufficient units to execute all of the operations in a block of the graph simultaneously. This is done to get an idea of the relative strengths of the area and time requirements. It also allows a quick view into the internal constraints produced by data dependencies and timing constraints. In the initial runs of a design, Chippe picks a system clock equal to the longest combinatorial delay plus the control and bus latency predicted for the initial control style. As can be seen from the figure, the performance of the circuit did not change under the first few modifications. This is due to the scheduler making use of the mobilities of the function units's executions to re-schedule the operations into smaller numbers of units without lengthening the critical path. The goals for this design are very restrictive on the number of gates allowed for implementation and the present unit allocation is much too

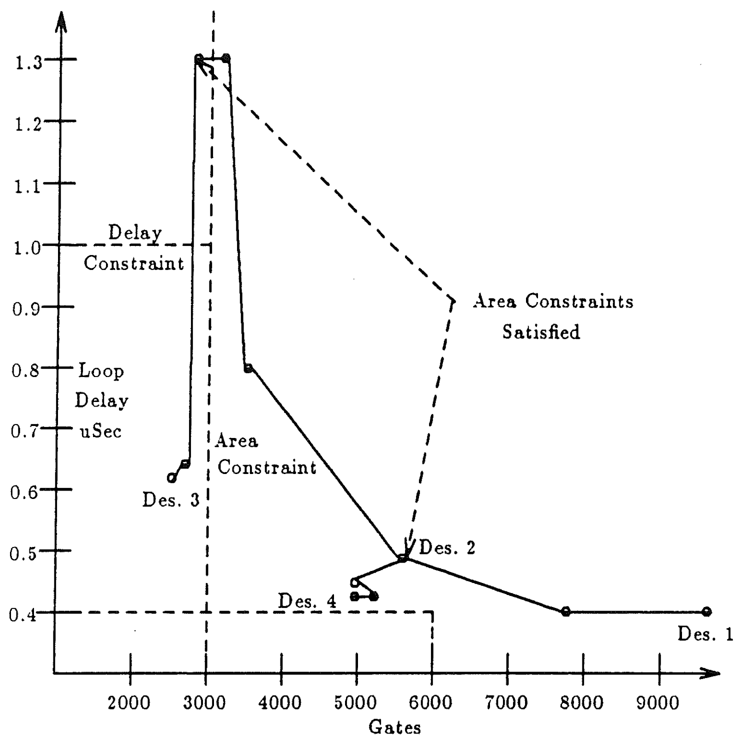
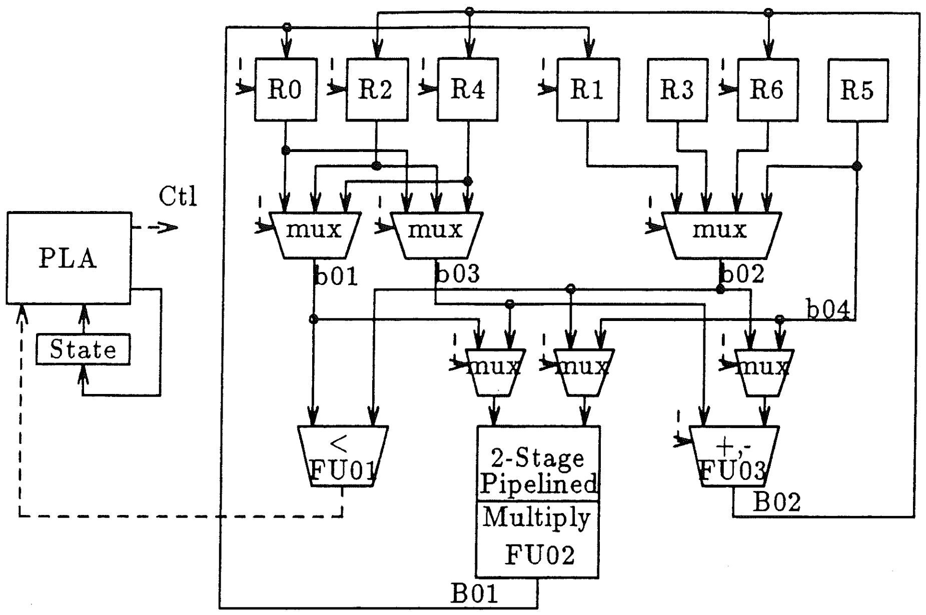


Figure 7. Design Evolution

large, so the strategy selected is to reduce the area until the area goal is met. This procedure resulted in the sequence of designs with performances decreasing as the area is reduced.

After the area goal was satisfied (or if this goal had failed) the strategy changed to trying to increase performance without much area increase. This was accomplished by pipelining the multiplier unit and adjusting the system clock to take advantage of this change. After this modification, the area could be reduced still further by eliminating a unit made exclusive by the change in the schedule. Figure 8 shows the final design achieved by Chippe for this example. After the area, time (and power) were satisfied for this design, the interconnect task was automatically set to greater look-ahead and iteration limits to enhance the quality of the final design. The actual time for this design is about 12



MI#	ACTIONS	Nxt	Conditions
1 a	FU01(<:r002,b01;r003,b02)	2 1	x < a : TRUE x < a : FALSE
2 a	FU02(*:r004,b01;r005,b02)	3	
3 a	r001,B01 = FU02(*:) FU02(*:r002,b01;r001,b02)	4	
4 a	r000,B01 = FU02(*:) FU02(*:r000,b01;r006,b02)	5	
5 a	r000,B01 = FU02(*:) FU02(*:r000,b01;r001,b02) r002,B02 = FU03(+:r002,b03;r005,b04)	6	

6 a	r001,B01 = FU02(*:)
	FU02(*:r004,b03;r005,b04)
	FU01(<:r002,b01;r003,b02)
		7	

7 a	r000,B01 = FU02(*:) FU02(*:r000,b01;r005,b04) r004,B02 = FU03(-:r004,b03;r001,b02)	8	
8 a	r001,B01 = FU02(*:) r006,B02 = FU03(+:r000,b03;r006,b02)	9	
9 a	r004,B02 = FU03(-:r004,b03;r001,b02)	2 1	x < a : TRUE x < a : FALSE

Figure 8. Final design for Hal Example

CPU seconds on a SUN 3/140. The first 7 designs were complete in about 5 seconds, about

7 seconds were spent optimizing the final interconnect.

The table that appears under the figure is the output symbolic microcode for this design. The symbolic microcode and the micro-architecture contain sufficient data to build the control unit. Estimates of the control unit size based on the control-unit style and the micro-code are thus reasonably accurate. Each numbered block corresponds to a state of the machine while the lines describe which units are accessed and where the results are placed. The dashed divisions represent chaining partitions of single states, this mechanism allows the direct chaining of function units if there is sufficient time left in the cycle. The FUxx, rxxx, bxx, and Bxx are function units, registers, input and output busses respectively. Operands are supplied to the function-units on the indicated busses. In this example (to conform to the original Hal paper) the initial values for the registers are assumed to be stored at the start of the code fragment. In a more realistic case these values could be loaded from a constant ROM or from external ports in the environment. Also, in this design the loop nature of the code fragment is explicitly used to assign registers used to store the values between cycles of the loop so that the values appear in those registers each cycle as needed.

The Final design shown in Figure 8 shows the design after the inclusion of a 2-stage piped multiply unit. This design modification occurred because the number of sequential multiplies became large enough for a pipe to be efficient. The design parameters for this design are 3000 gates and 636 nS loop performance, well within the desired goals. Notice that the two-level muxing structure has resulted in a design with four input busses and two output busses. The optimization of this design clearly splits the registers into two structural units, R0, R2, R4, and R1, R3, R5, R6. Additional rules could create register arrays for these partitions.

Changing the design goals to time < .4 uSec and area < 6000 gates resulted in design 4 in the figure. The evolution to this design started out the same as in the previous one but deviates as soon as the area goal is satisfied. Several attempts to achieve the required time were made, including pipelining the (two) multipliers and changing the clock. These changes are depicted in the design evolution chart as the line moving toward design 4. In

```
program ellip(input,output);
        /* Written from Benchmarks for Highlevel Synthesis Workshop */
type    integer = {0..15};
reg     t2, t13, t18, t33, t39, t26, t38,
        m21, m24, m9, m30, m40, m36, m16, m6 : integer;
port    In, Out : integer;
var     a, b, c, d, e, f, g, h, i, j, k, o : integer;
begin           /* Block automatically solves loop boundaries */
   i := In;     /* port read */
   a := i + t2;
   b := a + t13;
   g := t33 + t39;
   e := g + t26 + b;
   d := (m21*e) + b;
   f := (m24*e) + g;
   t26 := f + d + e;
   c := m9*(b + d) + a;
   h := m30*(f + g) + t39;
   j := t18 + c + d;
   k := t38 + f + h;
   o := m40*(h + t39);
   t39 := o + h;
   t38 := t38 + (m36*k);
   t33 := t38 + k;
   t18 := t18 + (m16*j);
   t13 := t18 + j;
   t2 := c + i + m6*(a + c);
   Out := o;  /* Port write */
end.
```

Figure 9. Source for Elliptic Filter

this case the design attempt failed to meet the goals and then returned the best design found.

5.2. Digital Elliptic Filter Example

The second example was chosen from the recent literature high level synthesis benchmarks. [Borr88] The application is a 5th order elliptic digital filter. Figure 9 shows the Chippe source code for this filter. Note that it is composed entirely of adders and multipliers, and there is no overt control. This design was studied to see the effects of very high levels of pipelining on the performance, area usage, and power consumption.

The assumption for Chippe in its early stages was that area could be traded for time in a design. The generality of the design model for Chippe allowed modification of the system clock as well as the allocation of units. When pipelining is introduced in a suitable design, area is traded for increased power consumption as operations are done in shorter clock periods. In fact, another tradeoff appears to be power vs. performance since the increase of operator parallelism (area) also increases power. To test this, several designs of the elliptic filter were run and the results plotted in Figure 10. In this figure, the boxes represent designs in the minimal area class (about 10k gates) with variations needed to the control units for the changing clock times. The circles represent designs with a multiplier and 2 or more adders (about 11k gates), while the triangles represent designs with 2 parallel multipliers and any number of adders (about 17k gates). Note that in terms of power-delay-product, the best designs are in the intermediate area category. That is the designs fare less well when either too small or too large an implementation is attempted. To see how well this works when area optimization is applied, the above designs were re-plotted, this

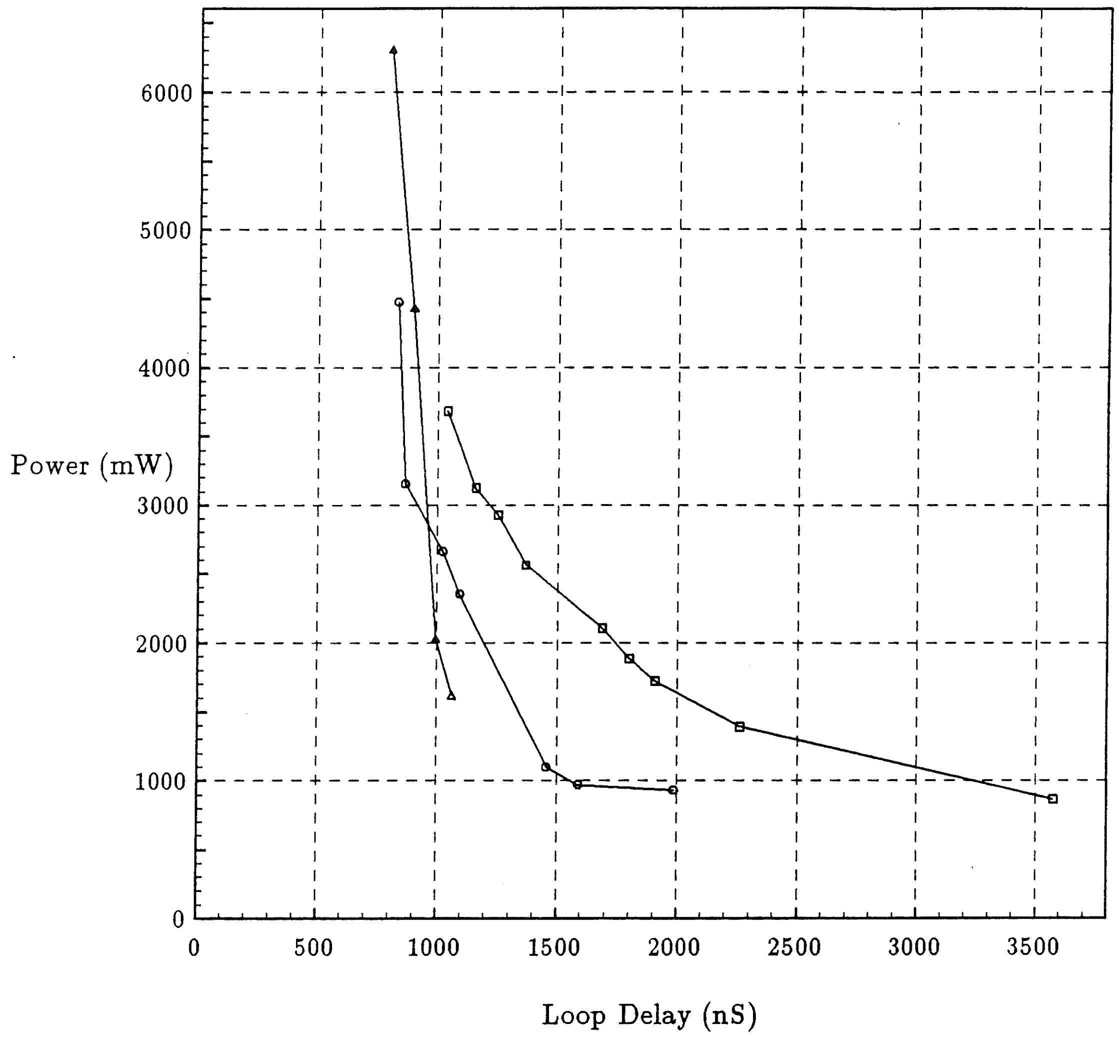


Figure 10. Power vs. Performance Tradeoff

time using total area vs. power times loop delay. Figure 11 depicts the result. The graph generally indicates that the largest designs aren't too good -- but also indicates that some of the smallest are really quite excellent. Sadly, the 'curve' also shows that simple 2-d tradeoffs on the designs may behave quite randomly. This is not as big of a problem (unless global optimization it needed) as it appears. The points in Figure 11 and in Figure 10 represent the same designs. The reason for the large changes in power delay product for

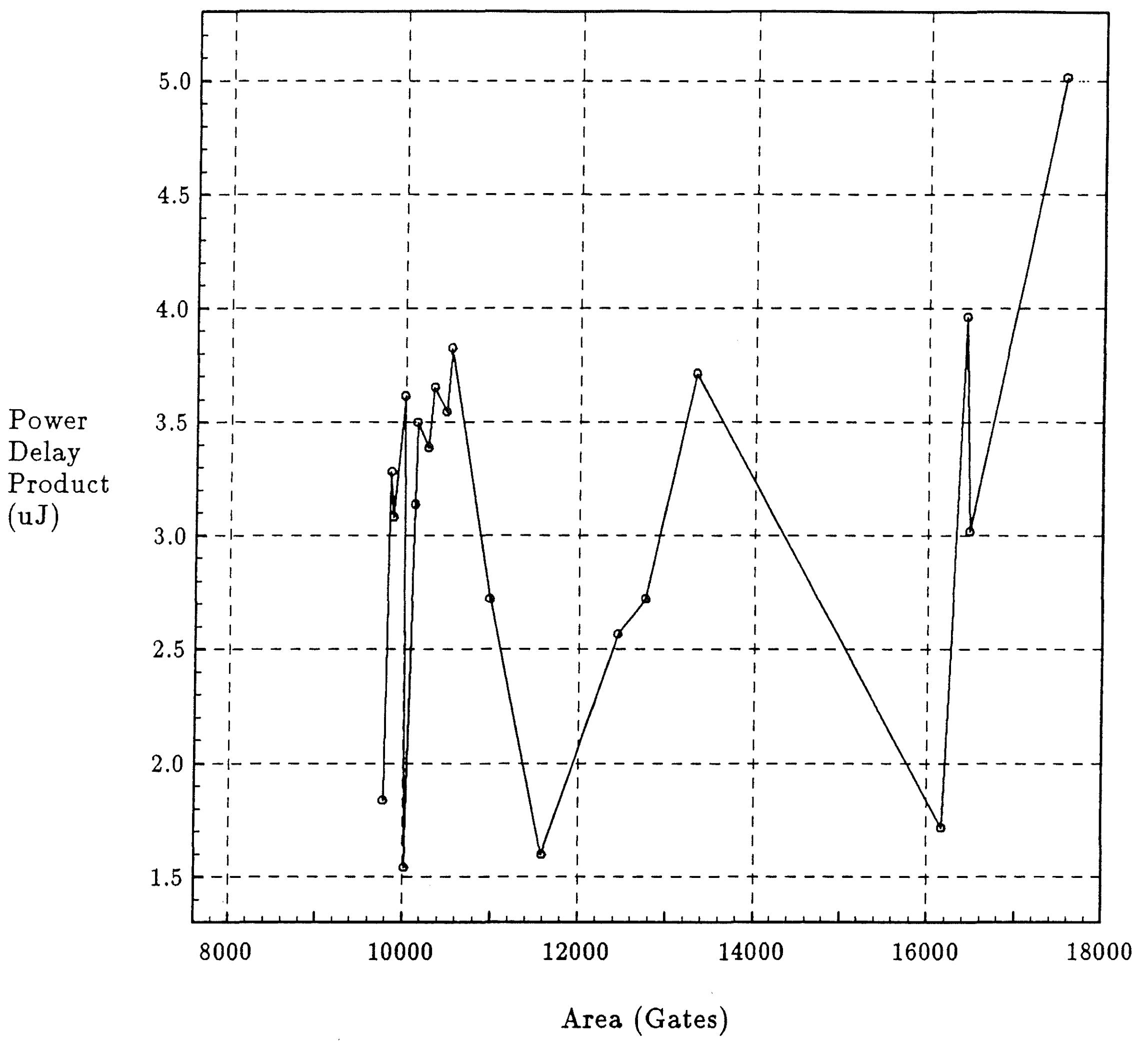


Figure 11. Power Delay Product vs. Implementation Area

small changes in area are that the designs represent different architectures satisfying similar global constraints. For example, the two lowest points on the right hand side represent designs where the larger controller and high speed clock were replaced by an extra adder. This allowed very efficient schedules for the designs although the actual performance suffered. The design at 11600 gates is actually a very good design in several senses, it is high performance at relatively low power and reasonable area. From Chippe's point of view these designs are actually fairly widely separated as performance can be used to differentiate the close points.

One last experiment was run on the Elliptic filter example, this time forcing the system clock to a preset value and letting Chippe supply sufficient (non-pipelined) units to schedule the graph as quickly as possible. This was done to evaluate all of the possible

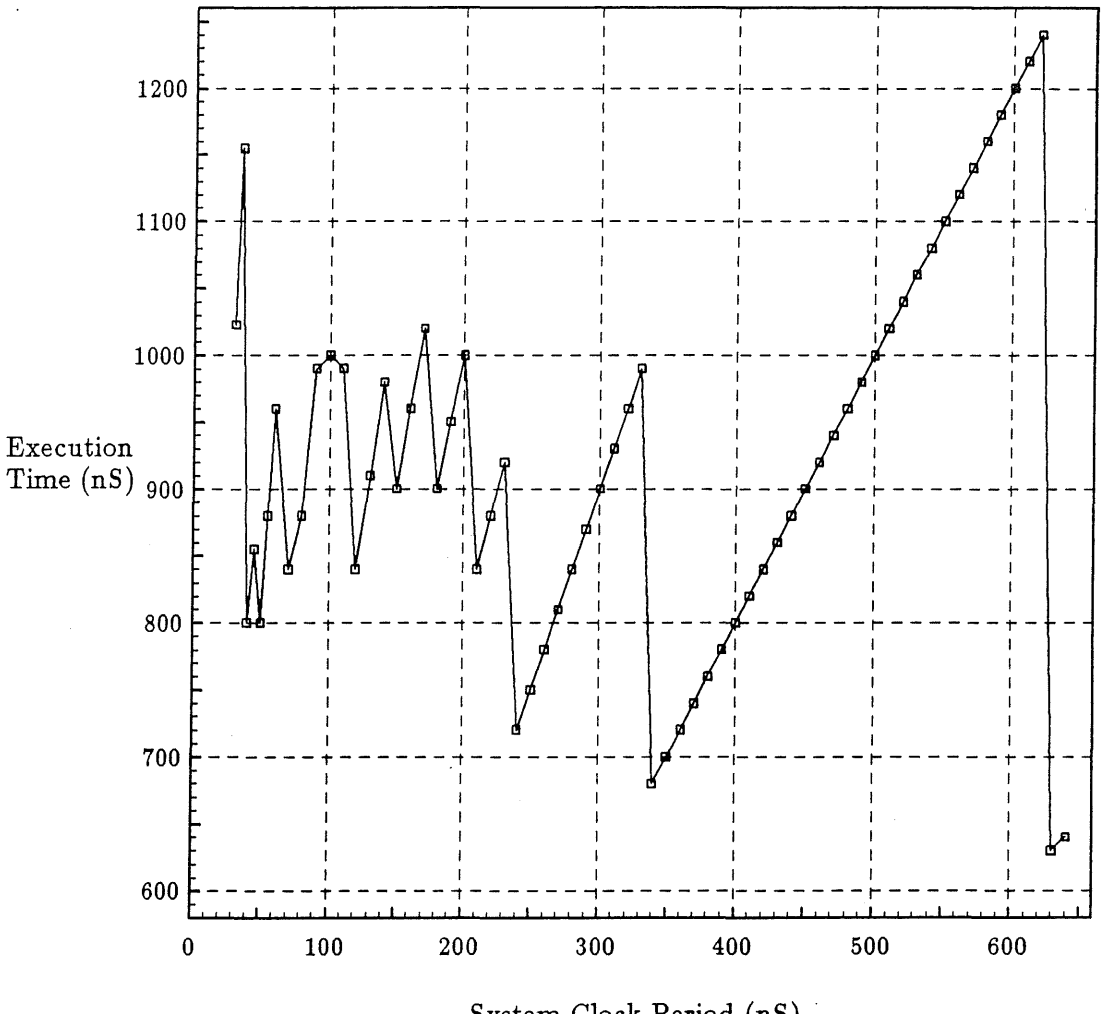


Figure 12. Elliptic Filter Clock Rate vs. Performance

non-standard clock cycles to see if particularly interesting ones other than those found by Chippe existed. Figure 12 shows the results of these designs. In the figure, the designs with very long clock cycles correspond to sufficient hardware to chain the entire design, resulting in the fastest possible design for the filter (at least for this scheduler). This design used over 52000 gates but dissipated only 0.975W. Other interesting designs occurred at 340nS (1/2 630 when latency added) and 240nS (1/3 ..) and 120nS. The designs at faster clocks were left in for completeness but mainly dissipated far too much power. The design at 120nS was found by Chippe for goals of 25000 gates and 1.0uS cycle. Each of these performance maxima correspond to minima in the dead-time function used by Chippe to set the system cycle time. This is not surprising since the dead-time is a direct measure of the time wasted at the end of each cycle. It is naturally minimized by cycle lengths for which good schedules exist.

5.3. TMS320 Example

The TMS320 is a commercial digital signal processing chip designed to run at least 5 MIPS on 32-bit data. It contains a 16x16 multiplier and a 32-bit ALU and is designed to execute most instructions in 1 200nS clock. This is achieved by the used of a Harvard architecture for the computer which allows simultaneous access to instructions and data in to separate memory storage areas. To make use of this ability, the TMS320 is internally pipelined so that as many as 3 instructions can be read, executing, and pending at one time. This design was chosen for automatic implementation by Chippe to explore the issues of large scale microprocessor design.

The Chippe source code for the TMS320 [Brew88] is an imperfect representation of the chip functionality. Although all of the instructions were implemented, several 'features' were not. These include the accumulator 'saturation' where at behest of a bit, all positive overflows result in an output of the maximum positive 32-bit number, and a similar case for negative overflows. Also, in storing the auto incremented and decremented values of the data address pointers into the data itself, the 320 uses counters which are clocked in mid instruction and are difficult to emulate. The Chippe code assumed that all auto-increment and decrement activity takes place in the fetch cycle of the machine, not the execute/write cycle.

The TMS320 design was run with several area, time, and power constraints and the results shown in Surprisingly, the available area/time unit tradeoffs did not affect the schedule at all. This is due to two causes in the design. The first is that although the TMS320 is itself highly pipelined operationally, there is little parallelism in the operations preformed on its internal data-path. Specifically, the specified instructions often allow chains of several internal function units but offer no other possibility for parallel operations. This is not surprising since to allow parallel execution of different functions, the instruction set would somehow need to specify the sources and destinations (or equivalent) for all parallel operations. This would be quite cumbersome for complex instructions and only a small number of parallel function units. (For simple instructions this implies that the "opcode" is actually providing a condensed version of the micro-code; this is exploited in VLIW architectures.) [Elli85] The second cause for lack of unit tradeoffs is that the parallel operation of the address generation for instructions and data and the execution of the data-path all made use of operators of dissimilar bit-widths. Thus Chippe cannot tell that these units could be combined. What Chippe did instead was to insert slower units in the areas of the design where speed was not an issue.

Interestingly, although pipelining is usually a good idea-- there is no point in pipelining the multiply of a TMS320. The reason for this is the nature of the present design representation. In this implementation, at least 6 cycles separate successive multiplies in the worst case. This allows more than enough time for completion of a multiclock multiply cycle. An unusual feature of the design is that the arithmetic and logic units were designed as separate instantiations. The reason for this is an artifact of the TMS320 design. In this machine the logic operations extend for only 16 of the 32 bits in the accumulator. Thus, in Chippe's implementation, these are left as separate units. The total design just described ran in about 2 min of time on a SUN3/140 workstation. The final design parameters are shown in Figure 13.

```
Design 1: Clk 52nS
        Ctl: Piped_Mealy, 58 Ctl_lines, 297 States, Area 34.7k, Pwr 311mW
        Conn: Style_2, 15/27 Bus Conn, 42 Muxes, 124 Muxinputs, Area 21.1k, Pwr 1.07W
        Data-Path: Area 46052, Pwr 1.78W
        Totals: Area 101942, Power 3.167W, Avg. Cycle 389.2nS

Design 2: Clk 62.4nS
        Ctl: Piped_Moore, 49 Ctl_lines, 138 States, Area 13.6k, Pwr 222mW
        Conn: Style_4, 25/35 Bus Conn, 32 Muxes, 126 Muxinputs, Area 28.8k, Pwr 836mW
        Data-Path: Area 46212, Pwr 1.49W
        Totals: Area 88623, Power 2.55W, Avg. Cycle 681.5nS

Design 3: Clk 47.1nS
        Ctl: Piped_Moore, 48 Ctl_lines, 139 States, Area 13.6k, Pwr 298mW
        Conn: Style_4, 24/35 Bus Conn, 31 Muxes, 123 Muxinputs, Area 28.3k, Pwr 1.14W
        Data-Path: Area 46372, Pwr 2.09W
        Totals: Area 88397, Power 3.527W, Avg. Cycle 500.1nS
```

Figure 13. Final TMS 320 Design Results

Three designs are presented in the results, the differences between the designs mainly resulting from differing control unit and interconnection styles. Design 1 has the fastest average cycle found for designs using less than 4W (chosen to reflect package limits). The time constraint forced a tradeoff of area in the controller to a Mealy machine with a pipeline register to the data-path. The large number of states is a result of encoding the conditional returns into the state number. The connection style "Style_2" selects an option to minimize busses first and the the number of muxes in the connection heuristic. (The other designs used Style_4 which strictly minimizes the muxes). This selection increases the controller area but reduces the total area in interconnect. The Bus Conn parameter measures the number of point to point busses used in the design. The large numbers result from using bus wiring even for single bit signals. Area is in terms of equivalent gates in each category as the system data-base style is gate array. Since all of the functions had to implemented in a gate-array, certain structures (such as the PLA control) were inefficient. This is especially true for the internal memory which shows up as function unit usage. The TMS320 has 1536*16-bit ROM and 144*16-bit RAM on board. These require about 30000 equivalent gates and dominate the function units.

The other two designs show that even with essentially the same data-path, there are interconnect control-unit and system clock tradeoffs. These designs resulted from an easing of the time constraint and tightening of the area limits. Design 2 was limited to 3W of power while Design 3 was allowed 4W. This clearly shows the ability to trade power for speed while using essentially the same areas for two designs.

5.4. System Limitations and Future Research

There are several limitations in the implementation of Chippe. Some stem from micro-architecture design model oversights while others were caused by incomplete or limited tool and expert implementations.

1) The first problem encountered with the TMS320 design was the common usage of bit fields of arbitrary size as the need arose. In Chippe these bus select and concatenate operators are modeled as function units to preserve bus bit width integrity and to have a means for keeping which bits are being selected. (Chippe is very careful to ensure that bit-widths of busses and units match. When bit selection is required Chippe requires a special unit to map the connections.) Unfortunately, it is very difficult to determine a priori where such units should be introduced into the interconnect. To solve this problem, these units would have to become part of the interconnection refinement along with an approach for combining busses of different widths. The result of these two effects in the present version made the problem much larger than it would seem. Instead of 10-20 different function units the design had to deal with about 100 and the interconnect required a minimum of 20-30 input and output point-to-point bussing connections.

2) A second problem, more basic to the Chippe design representation model was noticed. In Chippe, there is presently no way to allow the operation of a function unit to cross a block control transition[5]. Specifically, all multi-clock operations must be complete before the block is considered finished and execution of the next block can start. For example, a multiclock operation started in a short block must finish and write it's

[5]i.e. a branch transition of the control graph, not a state transition which is expressly allowed.

output before the block can end. The scheduler will add states as necessary to ensure that all operands are latched before a control transition. This is not strictly necessary as an operation could be started in a block and finished in a successor block. The problem stems from the present scheduler in Chippe which schedules only linear blocks. In an unconstrained physical implementation, the pending operations could be executed in parallel to the operation of the system controller even during a branch-- a design model oversight in Chippe. For Chippe's design model to allow this, there would have to be a mechanism to force timing constraints across control branch transitions. A better method would be to simply include the entire graph (including loops) as a possible input and redesign the scheduler.

3) The greatest limitations (from the tradeoff ability of Chippe) come from the lack of bus and interconnection modeling in the resource driven scheduler. Specifically, the system has no way of constraining the number and type of busses used in a design. The number of busses is determined by the number of data-path arcs which are used in the most parallel instruction scheduled. Since the scheduler is resource based, a possible resource addition would be bus limitations. At present the scheduler will maximally parallelize the data path to get the fastest schedule possible on the given number of function units. Since busses with many connections can use large amounts of area, a constraint reducing the number of allowed busses (and possibly reducing the performance) would enhance Chippe greatly.

4) The number of possible interconnect styles is more limited than it should be, primarily because of the schedule problems above but also because of the strict bit-width matching criteria. Operands of many sizes should be allowed to be passed on a bus,

although this greatly increases the design search space. Furthermore, busses should be allowed to go between any sources and destinations and need true bi-directional control. This problem is extremely difficult as the number of possible combinations of directions and data packings into such busses grows extremely rapidly with problem size.

5) The distinction between registers and functions units and interconnection units should be dropped. Since function units can contain state and pipeline registers, there is no need to have separate "registers". This change would enable Chippe to use shift registers, counters, and other "registers" with functions. At present, registers are the repository of state during all state transitions and so are handled specially by the scheduler and interconnect tasks. Function units can contain pending operations but "latching" is not considered an operation. Furthermore, registers can be simultaneously read and written to with differing operands. This can already occur for pipelined units so a simple extension of the function units would allow their use as registers.

6) Future research is needed to perform directed graph tradeoffs and optimizations, both of the restructuring type and in terms of operator mapping. The generality of the function unit model allows for complex units including small FSM's. Once suitable graph partitions are found, tools could be built to custom create local components for those partitions. For example, a "if" statement could be implemented in the data path by a suitable combination of logic gates and a mux-- all of which could be chained into a single cycle. Once built, these units could become part of the high level design in the same mapping context as present data-base units are allocated. This would allow the tradeoff of custom component design vs. implementation in the data-base

set.

7) The large size and domination of the controllers in the highly parallel designs showed the desperate need for other control strategies. While the PLA works fine for central state control, it is clear that size reductions can be made by introducing nano-coding and local distributed control into the designs. This would require that the distinction between data-path function and control function be dropped and both parts would need to be simultaneously scheduled. Then fast pipelined units could have their own mini-controller running at a different clock from that of the main controller.

6. Conclusions

This paper described a system implementing a simple mechanism for performing closed loop design based on a knobs and gauges approach to feedback. The intent is to allow opportunistic optimization and design refinement in a design environment which not only has tools for modifying the potential design, but also has evaluators and means of making new design decisions. This approach is inherently iterative as the modifications to the potential designs cause changes which can propagate throughout the design. To avoid this problem, fast refinement and design decision tools were implemented to allow redesign with the new constraints.

The implementation of Chippe made use of another key idea: separation of design implementation knowledge from design analysis knowledge. In Chippe, the implementation knowledge resides in the algorithmic tools thus freeing the expert system to analyse and correct constraint violations without having to know how to make the changes directly. This is a result of the expert controlling "knobs" of the refinement tools. The idea is to put the easily organized implementation knowledge into algorithmic tools and put the tradeoff knowledge which is less well organized into the expert. Making the entire program algorithmic or rule-based would have increased the complexity and developement time.

Although the present implementation of Chippe has many deficiencies, the implementation has shown that closed loop designs can be generated completely automatically, from settings of global goals. These designs are comparable if not superior to the best of the present day synthesis systems, when such comparisons can be made. Unfortunately, many of the designs produced use pipelined components and/or pipelined controllers and are difficult to compare with published examples, as most other systems cannot make use of these components. Since Chippe uses units with operation times which can vary over 2 orders of magnitude comparing schedules is also difficult. Most comparable systems assume all operations are of a unit time. Lastly, Chippe is fast-- the longest iterative sequences (The TMS320) took about 5 min. on a SUN3/140. In these sequences, the design tradeoffs were done in the first minute, all the rest of the remaining time was spent optimizing the interconnect.

7. References

Barb81
M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications" *IEEE Transactions on Computers* **30**(1)(Jan, 1981).

Borr88
G. Borrello et al, "Private Communication: High Level Synthesis Workshop" *25th IEEE Design Automation Conference* (To be held June, 1988).

BrGa86
F. D. Brewer, D. D. Gajski, "An Expert System Paradigm for Design" *23rd IEEE Design Automation Conference* pp. 62-68, Las Vegas, NV (July, 1986).

GaBr87
D. D. Gajski, F. D. Brewer "Towards Intelligent Silicon Compilation" *Design Systems for VLSI Circuits* ed. G. de Micheli, A. Sangivanni-Vincentelli, P. Antognetti,

Martinus Nijhoff Pub. (1987).

BrGa87
F. D. Brewer, D. D. Gajski, "Knowledge Based Control in Micro-Architecture Design" *24th IEEE Design Automation Conference* Miami, Fl (July, 1987).

Brew88
F. D. Brewer "Constraint Driven Behavioral Synthesis" *PhD. Thesis, University of Illinois, Urbana-Champaign, Dept. of Computer Science.* (May, 1988).

Catt87
F. Catthoor, "Architectural Design Strategies for Complex DSP Systems in an Automated Synthesis Environment" *Phd Thesis, Katholieke Universiteit Leuven: Dept. of Electrical Engineering. (IMEC)*, (May, 1987).

DeRS86
H. DeMan, J. Rabaey, P. Six, "CATHEDRAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip" *NATO Study Institute on Logic Synthesis and Silicon Compilation forVLSI design*, L'Aqulia, Italy (July, 1986).

Dutt86
N. Dutt, "COGENT: A Parametrizable Control Generator fo Constraint Driven Microarchitecture Synthesis" *PhD Qual. Exam, University of Illinois Urbana-Champaign* (Nov, 1986).

Elli85
J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures," Ph.D. dissertation. Yale University, (Feb, 1985).

GrKP85
J. Granacki, D. Knapp, A. Parker, "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface," *22nd Design Automation Conference* (June, 1985).

JVJC86
A. A. Jerraya, P. Varniot, R. Jamier, B. Curtios, "Principles of the SYCO Compiler" *23rd Design Automation Conference* IEEE ACM, Las Vegas, NV, (July, 1986).

KnPa86
D. W. Knapp, A. C. Parker, "A Design Utility Manager: The ADAM Planning Engine" *23rd Design Automation Conference* IEEE, Las Vegas, NV, pp. 48-54, (July, 1986).

Knap86
"D. W. Knapp" "A Planning Model of the Design Process" *PhD. Dissertation, University of Southern California* (Dec., 1986).

Kowa84
T. J. Kowalski, "The VLSI Design Automation Assistant: A Knowledge-Based Expert System" *PhD. Dissertation, Carnegie-Mellon University* (April, 1984).

McFa86
M. J. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital

Hardware from Abstract Behavioral Descriptions" *23rd Design Automation Conference* pp. 474-480, (July, 1986).

Pang87
B. Pangrle, "A Behavioral Compiler for Intelligent Silicon Compilation" *PhD Dissertation, University of Illinois, Urbana-Champaign* (June, 1987).

PaGa86
B. Pangrle, D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation" *Proceedings ICCAD*86 Santa Clara, CA, (Oct, 1986).

Pang88
B. Pangrle, "A Heuristic Approach to Connectivity Binding" *Proceedings DAC*88 Anaheim, CA, (July, 1988).

PaPa86
N. Park, A. C. Parker, "SEHWA: A Program for Synthesis of Pipelines" *23rd Design Automation Conference* IEEE, Las Vegas, NV, (July, 1986).

PaPM86
A. C. Parker, J. Pizarro, M. Milnar, "MAHA: A Program for Datapath Synthesis" *23rd Design Automation Conference* IEEE, Las Vegas, NV (July, 1986).

PaKG86
P. G. Paulin, J. P. Knight, E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis" *23rd Design Automation Conference* IEEE, Las Vegas, NV, pp. 263-270, (July, 1986).

PaKn87
P. G. Paulin, J. P. Knight, "Force-Directed Scheduling in Autonmated Data Path Synthesis" *24th Design Automation Conference* IEEE, Miami, FL, pp. 195-202, (July, 1987).

Raba87
J. Rabaey, "CATHEDRAL-II: Computer Aided Synthesis of Digital Processing Systems" *IEEE Custom Integrated Circuits Conference* Portland, OR, (May, 1987).

Snow78
E. A. Snow, "Automation of Module Set Independent Register-Transfer Level Design" *PhD Dissertation, Carnegie-Mellon University* (April, 1978).

Tric87
H. Trickey, "A High-Level Hardware Compiler" *IEEE TRAN. on Computer Aided Design* **CAD-6**(2), (March, 1987).

Wolf86
W. Wolf, "An Object Oriented, Procedural Database for VLSI Chip Planning" *23rd Design Automation Conference* IEEE, pp. 744-751, Las Vegas, NV, (July, 1986).