Towards achieving an 100-hour Design Cycle: A Test Case

Daniel D Gajski [1]
Loganath Ramachandran [1]
Peter Fung [2]
Frank Vahid [1]
Sanjiv Narayan [1]

Technical Report #94-08
February 14, 1994

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Abstract

With recent success in logic synthesis tools, many designers are focussing on capturing the design with a Register Transfer Level (RTL) description and using synthesis tools to complete the rest of the design. Due to the large amount of detail, writing a RTL description requires a long time, making it unacceptable when designs have to be produced rapidly. In this paper we propose a design methodology that can shorten the design cycle significantly. This is achieved by specifying the design at the highest level of abstractions and using powerful tools to complete the rest of the design. We have used this methodology on an industrial application, where we have designed and implemented a fuzzy logic controller.

---

[1]University of California, Irvine, California
[2]Matsushita Electric Works Research and Development Laboratory, Inc. San Jose, California

Contents

List of Figures

1 Introduction

Recent successes of commercial logic synthesis tools has changed the focus of design methodologies. Instead of focussing on logic descriptions, many designers are focussing on specifying the designs at the Register Transfer (RT) level. In order to specify the design at the Register Transfer level, the designer must define: (i) the operations and the register transfers that should be executed during each clock cycle and (ii) the exact state transitions of the controller. However, this task is not easy because:

[a] The amount of detail to be specified at the RT level is quite large, particularly for designs with many states. Also, debugging of description errors is time-consuming.

[b] The number of design alternatives is large and their evaluation further increases the specification time.

[c] Verification is difficult because of lack of formal models at higher abstraction levels. This could even result in wrong tradeoffs.

For the above reasons, writing a correct behavioral description (with proper design tradeoffs) is time consuming further increasing the design cycle time. This long design cycle is not acceptable for rapid design turnarounds.

The main goal of our research is to drastically reduce this design cycle time to less than 100 hours. In this paper, we propose a design methodology, which is a step towards achieving this goal. We have focussed this methodology specifically for quick design turnarounds. We have been able to achieve this extremely short design time because of the following reasons:

1. **Specification of the design at a very high abstraction level.** This minimizes the amount the detail that the user has to specify manually. Moreover by keeping the specification close to the conceptualization level, we ensure that the specification can be done correctly with very little time and effort.

2. **Use of standard languages for input specifications.** By using standard languages like VHDL, (or front- ends based on VHDL) we exploit the vast support available for them. Moreover, designers can reduce the number of specification errors, since the syntax and semantics of the front-end language closely match design characteristics.

3. **Efficient tools for design space exploration at each abstraction level.** Since there are numerous design tasks from the specification to the implementation, we divide the tasks into three groups which are performed at the three abstraction levels. We have developed CAD tools for each abstraction level to carry out many of the exploration and optimization tasks.

4. **User interaction to guide synthesis.** Since complete automation is not possible, user interaction is provided at all levels to guide the exploration and refinement process. The designer may override the decisions of the automatic tools to satisfy special cases. In order to enable designers to guide the synthesis process, it is necessary to provide quality measures for the design, whenever required. Such quality metrics evaluations can also be used by the automatic algorithms when exploring the design space.

5. **Enabling design verification at all abstraction levels.** After completion of each task at an abstraction level, it is necessary to check the status of the design. Our methodology allows simulation at each abstraction level, to verify the correctness of the synthesis process.

For the remainder of this paper, we describe the details of the tasks at each abstraction level. We then illustrate the power and speed of this methodology by showing the design steps on an industrial design.

2 Our Design Methodology

2.1 Design Specification

In Figure 1, we show the important parts of our design methodology. The design is specified at the highest possible level of abstraction using a model that closely resembles the way designers would conceptualize a design. The actual design steps consists of three sets of tasks, which include, (a) **System design tasks** that deal with mapping the specified functionality into multiple chips. (b) **Architectural design tasks** that deal with designing the architecture (i.e., control and datapath) for each of the chips, (c) **Technology adaptation tasks** that deal with mapping the components in the chip to actual physical implementations. Detailed descriptions of each of these tasks are given in Section 2.

In order to shorten the specification time considerably, the design specification style must very closely resemble the way designers conceptualize the design. To make this possible, we have defined a model called **Program-state Machines (PSM)**, which is based on a combination of the hierar-

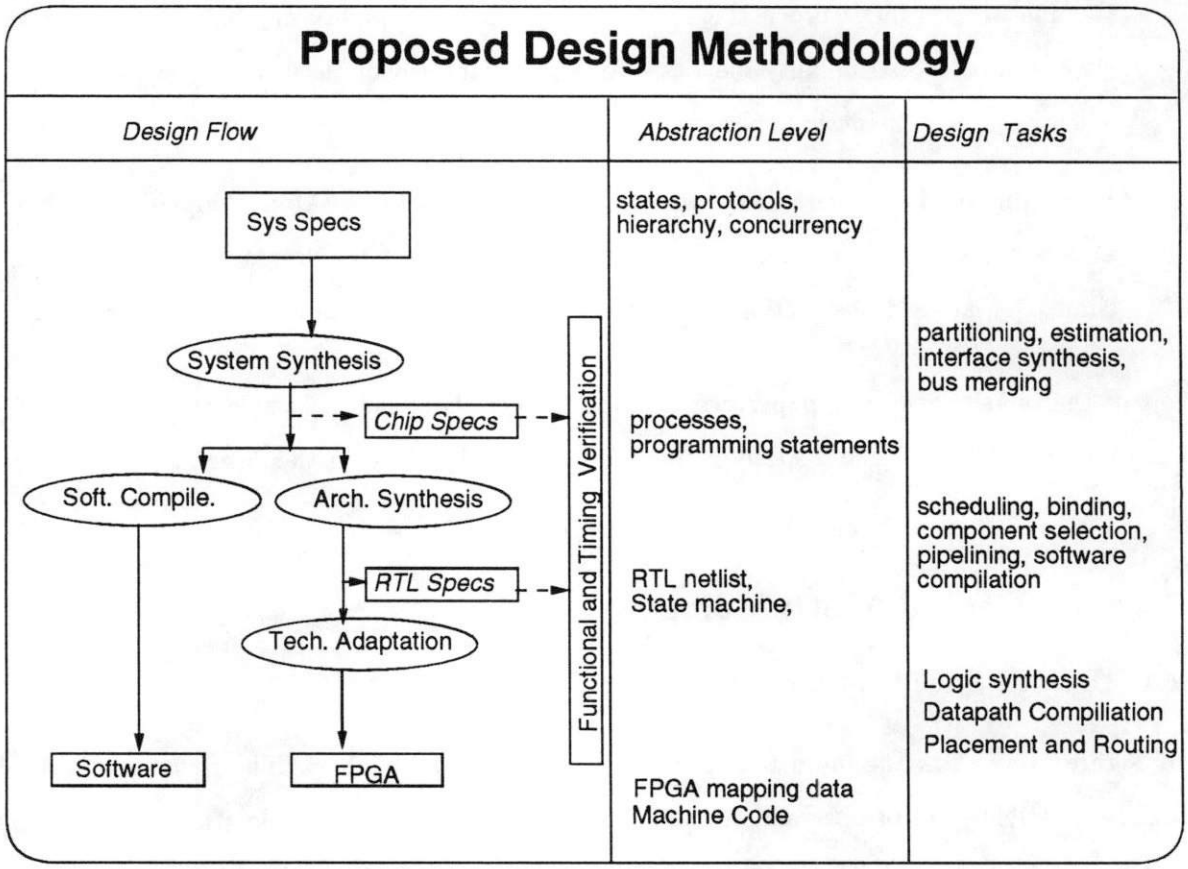


Figure 1: Top Down Design Methodology

chical finite-state machine and programming language paradigms [1]. With this model, a system can be specified as a hierarchy of program-states, where each program-state may either be a composite program-state or a leaf program-state. The composite program-states contains a set of concurrent program-states (all of them active) or sequential program states (only one of them active). The leaf program-states have a sequence of computations expressed using programming language statements.

We have developed a language called SpecCharts [2], which is based on this PSM model. The computations in the leaf program-states are expressed with VHDL programming statements. They contain three object classes: **variables** that store data, **behavior** that represents an algorithmic level computation, and **channels** that move data between behaviors. Based on the PSM model, the composite program states are constructed by combining two or more such VHDL based leaf program-states such that they execute sequentially or parallely.

SpecCharts is a high level specification language that can completely describe the behavior of a system. The amount of details at this level is very small, since the description consists of pure behavior, with no connotations to any hardware implementations. This simplifies the specification enabling us to possibly achieving our 100 hour design cycle goal.

SpecCharts is basically designed to serve as a front end for VHDL[3]. Hence the syntax and semantics of the language are well defined reducing the number of possible errors caused by language ambiguities. We have also developed translators to converts SpecCharts into VHDL, allowing for easy simulation of the specification.

2.2 System Design Tasks

System Design involves mapping the functionality, as captured in the SpecCharts specification to a set of chips, such that all design constraints are met. The tasks at the system level deal with a very coarse granularity of objects (i.e., variables, behaviors and channels in the specification). Our approach to system design consists of three well-defined tasks on these objects. (a) allocation of chips, to satisfy the overall cost constraints, (b) partitioning of objects in the specification to satisfy the constraints, and (c) refinement of the behavior, to ensure that the various partitions can be interfaced together.

Allocation selects different system components or chips for the implementation. Typical components allocated include RAMs, standard processors, FPGAs, physical buses or custom ASICs. It is neccessary to allocate sufficient resources for implementing all the objects in the specification within

the performance constraint. At the same time, the amount of resources allocated must not exceed the cost, power packaging and other constraints. Depending on the estimated quality metrics, the designer may try many different allocations, or allow the system to automatically choose a good allocation.

After allocating the chips in the system, the objects in the specification have to be partitioned and mapped onto these chips. Thus **partitioning** forms the next important task at the system level. During the partitioning process, variables are mapped to memories, behaviors are mapped to standard/custom processors, and channels are mapped to buses.

Generally, there are a large number of partitioning solutions for a given allocation. However in order to achieve our rapid design turnaround goal, we need partitioning algorithms that can rapidly explore this large solution space. Clustering based partitioning is an example of a fast partitioning algorithm. Use of such algorithms in the partitioning process, can result in a large number of solutions being explored quickly. Fast estimators are required to evaluate the quality metrics for each partitioning tried by the algorithm.

The partitioning criteria varies for different types of allocation. Partitioning a system onto a set of ASICs require a different cost metric compared to partitioning a system onto FPGAs. In order to enable the partitioner to explore different types of allocation, the closeness criteria for the partitioning algorithms can be selected by the designer.

Partitioning causes a regrouping of the objects in the specifications. As an example, many variables in the description may be moved into one partition and grouped onto a single memory. Or, a single behavioral description may be split across multiple chips. It is necessary to add behaviors to the individual chips to maintain correct functionality for the given allocation and partitioning. In our first example, since many variables are stored in the same memory, address translation must be introduced for each memory access. In the second example, where behaviors are split among chips, additional interface descriptions must be introduced into both the chips to maintain correct communication. These sets of tasks are called **refinement**.

A summary of the tasks at the system level of abstraction is shown in Figure 2. After performing these tasks, a refined specification that can be verified by simulation is generated. This refined specification consists of an executeable specification; one for each system component. Each component spec contains VHDL process level statements. The channels between two chips form the input and output ports on the chips. The details of the system level design process is given in [4].

Each component spec must be synthesized into an ASIC or compiled into an instruction set for the

Functional objects	*System–design tasks*		
	Allocation	Partitioning	Refinement
Variables	Memories	Variables to memories	Address assignment
Behaviors	Processors	Behaviors to processors.	Interfacing
Channels	Buses	Channels to buses	Arbitration/protocols

Figure 2: System - Design: Tasks

allocated processor. Thus system design partitions descriptions into multiple components, which can be implemented either in hardware or software.

2.3 Architectural Design Tasks

Architectural Design involves mapping each chip description to a set of microarchitectural components, such that the design constraints for the chip are met. We use the **Finite State Machine with Datapath (FSMD)** model [5] to represent the design at the architectural design phase. This model consists of: (a) a datapath to represent operations, and (b) a finite state machine to represent the sequencing of the operations.

A design based on the FSMD contains (a) a datapath that contains functional units such as ALUs, storage units, and interconnect units, and (b) a controller specified in terms of state transitions and the value of the control lines in each state.

The tasks in the architectural design phase are very similar to the tasks in the system design phase, except for the level of abstraction. While system design tasks generates system structure, the architectural design tasks generate the chip structure. Our approach to architectural design consists of four tasks on the behavior generated by system design, (a) selection of components from a library (b) scheduling all the operations in the description into control steps (c) binding the operations in the description to the selected components, (d) design optimization by pipelining the controller and the datapath.

Component Selection allocates appropriate resources from a library of RT components. The library could contain many different implementations of the same functional unit, with each implementation differing in its cost and performance characteristic. There are many tradeoffs possible during component selection. Selecting faster components may improve the performance and the clock, but may result in a vary large area. On the other hand selecting slower components may reduce the

area of the design, but may result in the violation of the performance constraint. Thus component selection determines the appropriate mix of fast and slow components that meet both the area and the performance constraint.

The component selection phase deals with the allocation of three types of components. (a) *functional units*, which are used to perform all the operations in the behavioral description. (b) *storage units*, which are used for storing the data. (e.g., multiported RAMs, Register Files and Registers). (c) *interconnect units* that are used for transfer of data between the storage and functional units.

The components are selected from a GENUS [6] library, which provides an extensive set of parameterized generic components. These generic components can be bound to any technology specific implementation.

Scheduling partitions all the objects in the description (i.e., variables, operations and data transfers) into various clock cycles. The important goal during scheduling is to achieve the best performance given the allocated resources. During the synthesis process, the designer may try many different component selections. The schedule must be computed quickly in order to provide instant feedback to the designer about the quality of the component selection.

Binding derives a mapping of each object in the description to the allocated resources. All variables in the description are mapped to storage units, all operations in the description are mapped to functional units and all data transfers in the description are mapped to buses. The important goal during binding is to minimize the number of additional interconnect units required for implementing the design. Instead of mapping each variable into separate storage units, it is possible to share many variables in the same same storage space, if their lifetimes do not overlap, or if their access times do not overlap. Address translations are required when storing multiple array variable into the same memory module [7].

Design Pipelining techniques can be used to perform further design optimization. It is possible to decrease the maximum register to register delay by increasing the number of pipelining registers. However, these pipelining registers can be introduced either in the datapath or in the control parts. In *datapath pipelining* the datapath units (e.g., functional units) are pipelined into many stages. In *control pipelining* the pipeline registers are introduced between the control and the datapath [8].

A summary of the tasks at the architectural level of abstraction is shown in Figure 3. After performing these tasks, a netlist of the controller and the datapath is available for further synthesis in the next level of abstraction. In addition to the netlist, a VHDL model and a logic level specification

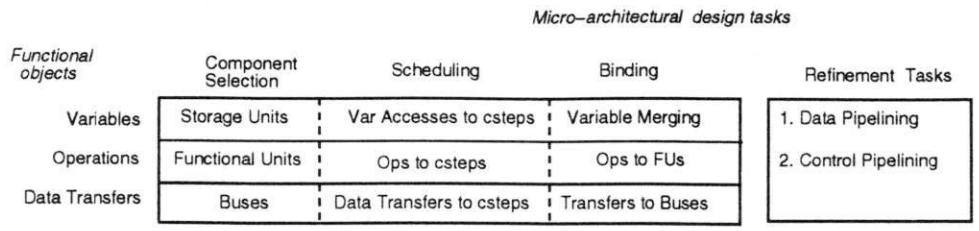


Figure 3: Architectural Design: Tasks

of each of the structural components is generated. This facilitates simulation of the synthesized design at this architectural level.

2.4 Technology Adaptation

The design in the previous phase uses generic components from the GENUS library. These generic components must now be implemented using real components from a vendor library. The **Technology Adaptation** phase performs this task of mapping all the generic components in the netlist to real components in a chosen technology.

It is possible to use custom, semicustom or programmable technologies for technology adaptation. However, in this report we concentrate on the field programmable gate array technology (FPGA) since it provides the fastest turnaround for design implementation.

FPGAs are regular structures consisting of programmable logic blocks (CLBs), memory elements and programmable interconnect blocks. An example of such an FPGA is the Xilinx set of chips [9]. Mapping a generic component into a FPGA block involves programming a set of logic and the interconnect blocks.

In order to facilitate mapping of components to specific FPGAs, the vendors provide (a) **Component Macros**, which are a set of mapping schemes for RT level components, (e.g., a vendor may supply component macros for 4 bit and 8 bit adders). (b) **Component Generators**, which is a set of tools for generating other RT components, (e.g., component generators may be available for generating Memory Modules of any size).

Our technology adaptation methodology exploits the availability of these high level components provided by the FPGA library vendor. Each generic component in our nelist is mapped into FPGA using three possible schemes:

1. If the same component is available as a Component Macro in the FPGA library, then the mapping

process is very simple. The generic component can be directly mapped into the Component Macro, since very efficient schemes for mapping the macro to the FPGA blocks is available with the vendor.

2. If a Component Macro is not available, then the list of Generators are checked to verify if a generator for that generic component is available. If the generators are available then the generic component can be easily synthesized using the component generators.

3. If both the Component Generators and the Component Macros are not available for the component, then a gate level netlist of the generic component is required for the mapping. Commercial logic synthesis tools are used to generate this gate level netlist, from the VHDL model of the component that is available in the GENUS library.

Although it is possible to automate the selection of one of these three implementation schemes for each generic component, this is done manually. Thus each component is identified with the appropriate mapping scheme (a) Component Macro, (b) Component Generators, and (c) Decomposition.

After each generic component is mapped into FPGA blocks, the blocks have to be placed on the physical FPGA chip, and interconnections between the physical blocks have to be completed. This forms the last task in the design process.

3 Fuzzy Logic Controller

In order to demonstrate the advantages of using the proposed methodology, we selected an industrial design used in consumer and industrial electronic applications. The particular application that we used was a fuzzy logic based temperature controller. Our goal was to specify the design at the system level and synthesize the entire design into a set of FPGAs. Another important goal was to guage the approximate time required for complete design from conceptualization to implementation.

In this section we provide the details of the fuzzy logic controller. We then discuss the fuzzy logic design experiment and the tradeoffs that were achieved in each stage of the design process.

3.1 Fuzzy Logic Controller: Basic Principles

The basic idea in fuzzy logic, [10] is everyday crisp values such as "temperature is 25 C" can be expressed as having a membership value in fuzzy sets such as "temperature is .9 hot" and "temperature

is .01 cold", where hot and cold are fuzzy sets and the values .9 and .01 represent the degree of membership in the respective sets.

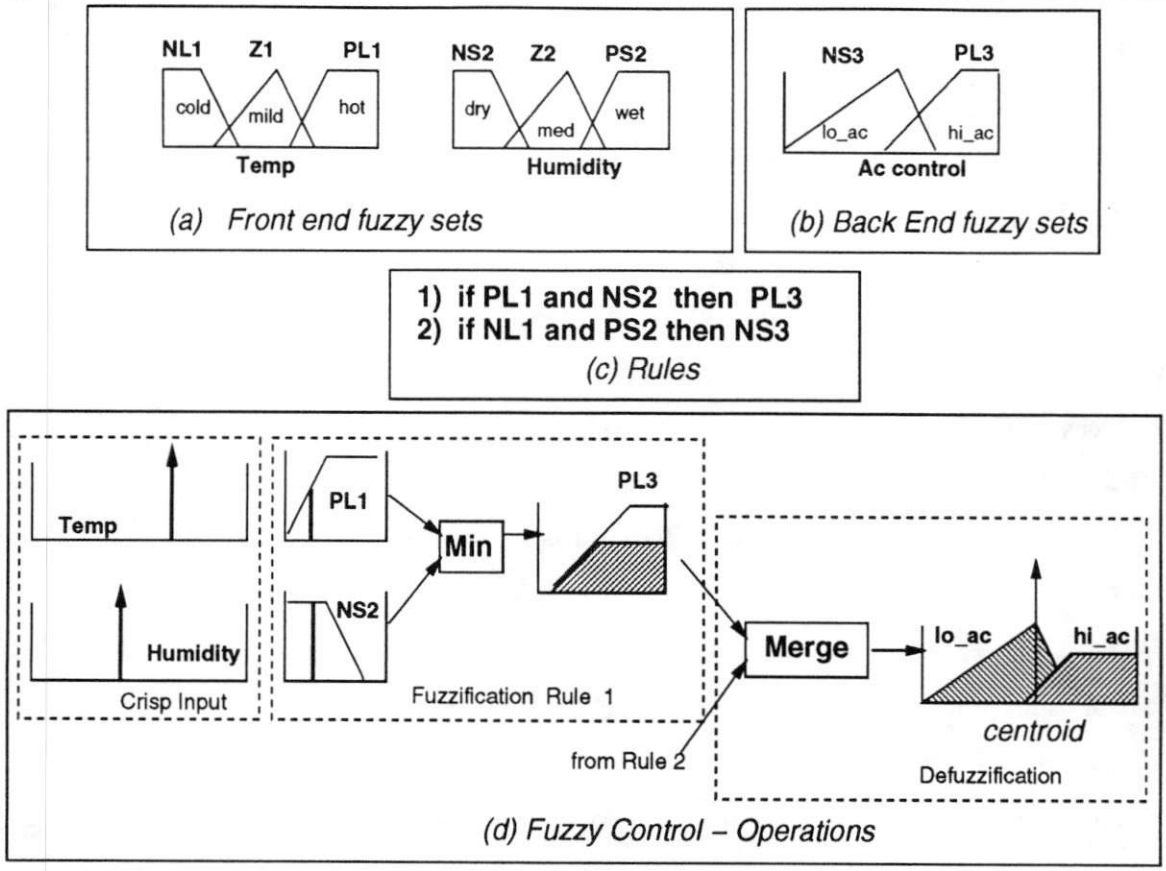


Figure 4: Fuzzy Controller Principles

In addition to the above fuzzy sets, each fuzzy logic controller incorporates two databases.

- **Sets of membership functions** to which the input facts belong in varying degrees. Typically, these sets take on names such as *Negative Large (NL), Negative Small (NS), Zero (Z), Positive Small (PS)* and *Positive Large (PL)*. Each variable has its own set of membership functions. In Figure 4(a) we show sets of membership for two variables *temp* and *humidity* in a sample fuzzy logic application.

- **A rule base** which governs the behavior of the fuzzy controller. Typically, these rules are in the form of *if..then* statements such as "if NL1 and NL2 then PS3", where NL1 and NL2 represent degree of membership in the membership functions. Figure 4(c) shows two such rules in our simple fuzzy controller.

The operations of the fuzzy controller are shown in Figure 4(d). Briefly, the fuzzy controller [11] performs the following functions: (i) **Fuzzification**, converts the crisp input values for the variables into a fuzzy value. The operations include many table lookups and comparisons. (ii) **Rule Application**, applies all the rules and produces a fuzzy output value. The operations include truncation and convolution of the membership functions. (iii) **Defuzzification**, converts the fuzzy output values produced by the rules. The operations here include computing the centroid of the back end membership functions.

3.2 Design specification

As explained earlier, we needed a VHDL based specification mechanism that supports behavioral hierarchy and concurrency. We used the SpecCharts language [2] for design specification since it satisfied all our requirements. The specification level is very close to the designers conceptualization for the design, and it is a front-end for VHDL based specifications. In addition it supports hierarchy and control flow.

The design specification consisted of five behaviors expressed using VHDL process statements. Four of these processes modeled the rules (R1 .. R4) in the system. The fifth behavior modeled the defuzzification function (i.e., centroid computation).

During specification, it was not clear whether to evaluate the four rules sequentially or concurrently. We decided to explore both these options and estimate the size and performance for each of these options during system design. Hence we wrote two different specifications of the design. In the first specification, we created a hierarchical state, in which the four rules were executed sequentially. In the second specification, we created a similar hierarchical state, but the four rules were executed concurrently. With SpecCharts it took less than 10 minutes to specify these two hierarchical descriptions from the leaf behaviors.

Since Specharts can be easily translated to VHDL, we were able to simulate both the sequential and the parallel specifications using a commercial VHDL simulator. It took us about 3 days to write both these specifications. The total lines of code for the specification was about 300 lines.

3.3 System Design

Specsyn [4] was used for partitioning the design onto several chips. Since our goal was a final mapping onto FPGAs, the system components that were allocated for partitioning purposes were only FPGAs.

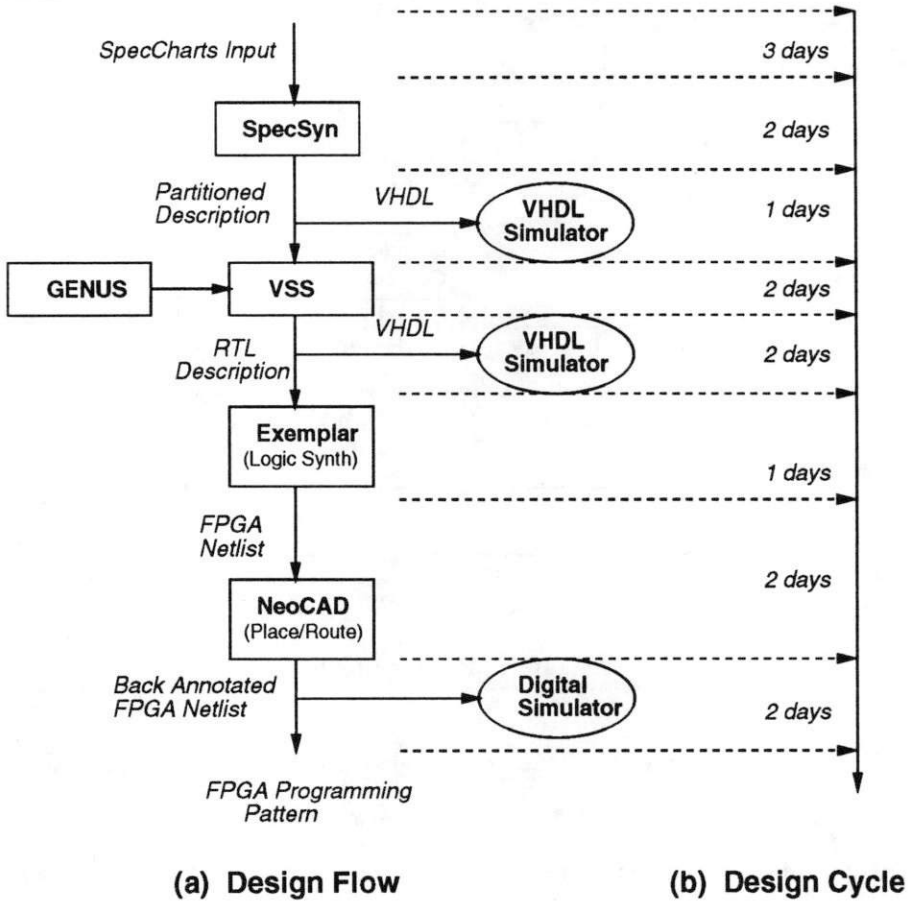


Figure 5: Fuzzy Controller - Design Steps

Many FPGA allocations were tried during system design. For each allocation provided to it, Specsyn partitioned the entire behavior into the allocated units such that the performance was optimal. In fact, Specsyn evaluates hundreds of different partitions in a few minutes, before deriving the most efficient partition. Specsyn contains an in-built performance and cost estimator, that provides quick and accurate on-the-fly estimates of these quality metrices. These estimates drive the optimization process during partitioning [12].

With a couple of days experimentation, we derived the best partition for the design. The final partition (as shown in Figure 6), consists of five major blocks, with appropriate interface protocols introduced between them for communication.

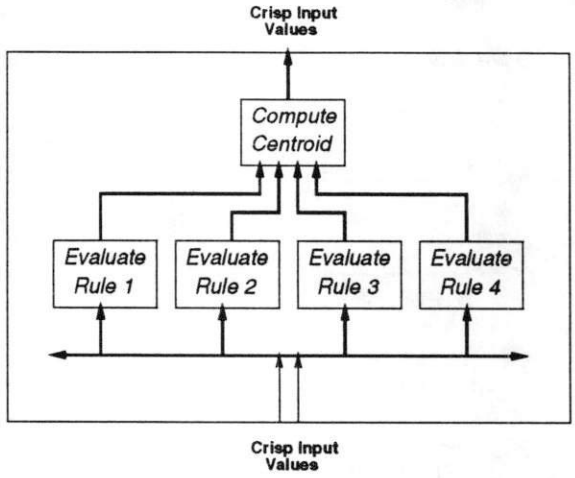


Figure 6: Partitioning Results

SpecSyn's performance estimation feature was also used to compare both the sequential and the concurrent rule evaluation styles. Partitioning and performance estimation not only revealed that the parallel algorithm is faster and requires more area than the sequential one, but also how this tradeoff impacts the overall system performance of the fuzzy controller (the true design criteria).

After partitioning all the objects in the specification to five chips, Specsyn was used to output a VHDL behavioral description for each of the five chips. This behavioral description was simulated to verify the partitioning and the interface synthesis done by SpecSyn.

3.4 Architectural Design

Behavioral descriptions for each of the five chips were obtained after system design. In Figure 7 we show fragments of the generated VHDL behavior for a single rule evaluation (EVAL_R1). We used

VSS [13] for further exploration of the design at the microarchitecture level and synthesis of the RTL structure. VSS performs scheduling, allocation, binding and array variable mapping and produces the RTL design. Components required during synthesis are obtained from the Genus library [6], which is a library of parametrized RTL components.

```
entity ch0e is
port ( f0: in integer;  f1: in integer;  gclk: in bit;  exec: in bit;
       done: out bit;  initr0m: in bit;  addrr0m: in integer;
       datar0m: in bv7; inittrRu0: in bit; datatrRu0: in bv7;
       addrtrRu0: in integer;  testdoutr0m: out bv7;
       testr0m: in bit;  testaddrr0m: in integer;  testtrRu0: in bit;
       testaddrtrRu0: in integer;  testdouttrRu0: out bv7);
end ch0e;

architecture ch0a of ch0e is
       type MembRuleArr is array (integer range <>) of bv7;
       type MembFunctArr is array (integer range <>) of bv7;
       signal r0m: MembRuleArr(383 downto 0);
       signal trRu0: MembFunctArr(127 downto 0);

begin
P0: process
       variable if1,itr, ir0mtr: integer;
       variable r0mf0, r0mf1, r0mtr, truncVal, temptr: bv7;
begin
   if exec = '1' then
       done <= '0';  if1 := 128+f1;  r0mf0 := r0m(f0); r0mf1 := r0m(f1);
       if r0mf0 < r0mf1 then truncVal := r0mf0; else truncVal := r0mf1; end if;
       itr := 0;
       while itr < 128 loop
           ir0mtr := itr+256;  r0mtr := r0m(ir0mtr);
           if truncVal < r0mtr then  temptr := truncVal;  else temptr := r0mtr;  end if;
           trRu0(itr); itr := itr+1;
       end loop;
       done <= '1';
   elsif initr0m = '1'    then  r0m(addrr0m) <= datar0m
   elsif inittrRu0 = '1'  then  trRu0(addrtrRu0) <= datatrRu0;
   elsif testr0m = '1'    then  testdoutr0m <= r0m(testaddrr0m);
   elsif testtrRu0 = '1' then   testdouttrRu0 <= trRu0(testaddrtrRu0;
   end if;
end process P0;
end ch0a;
```

Figure 7: Rule Description in VHDL

By varying the amount of allocated resources, we were able to explore a wide range of area - delay tradeoffs at the architectural level. We list some of the experiments attempted with the VSS system.

[a] We allocated different number of functional units. The performance of the design did not improve with more functional units, because this description (shown in Figure 7) does not contain many parallelizable operations.

[b] We changed the type of functional unit resources. Instead of providing a separate adder and a separate comparators we allocated a single ALU which can perform both these operations. This changed the design characteristic. Although the number of states increased because of fewer resources, the utlization of the allocated components increased substantially.

[c] Since the fuzzy logic control is a memory intensive application, changes in the allocation of memory resources drastically affected the final design. By increasing the number of ports in the memory we were able to significantly improve the performance. We also allocated slower and faster memories to consider the impact of memory access time on the overall design. We finally fixed the number of ports and the access time to correspond to the values in the Xilinx FPGAs. Thus synthesis constraints reflected the next task in the design cycle.

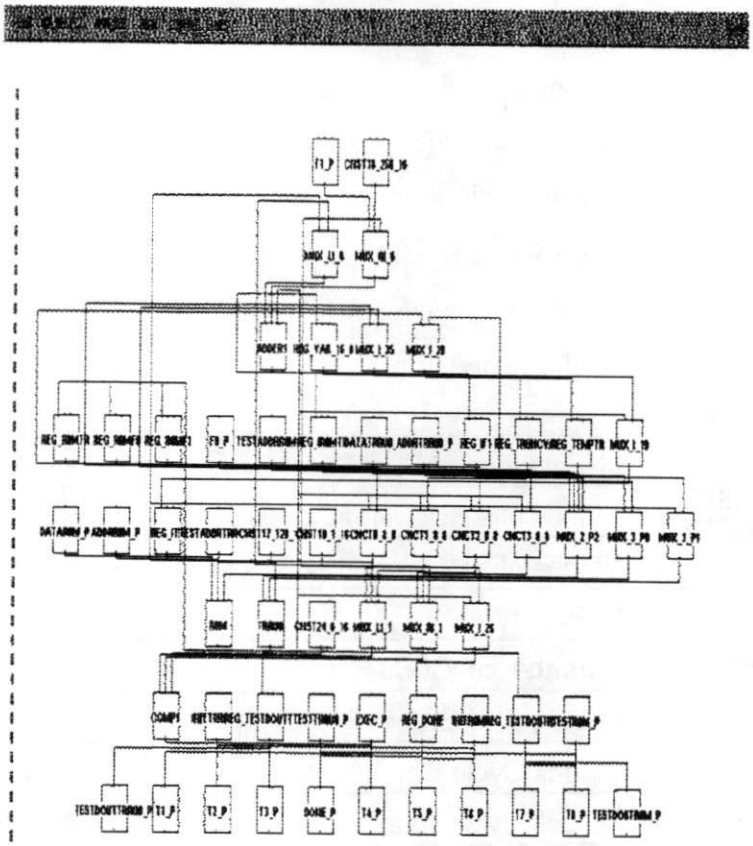
[d] We selected a 100 ns clock for the design, since most of the functional unit delays in the FPGA library was in the order of 30-40 ns, and data accesses were in the order of 60 ns.

Since VSS uses fast algorithms for scheduling, variable merging and binding, it was able to produce the architectural design for a given allocation within a minute. Thus we were able to synthesize and verify almost about 10 different allocations in less than a day. In Figure 8 we show one of the design netlist and the quality metrics for one of the FPGAs. This design was generated by VSS during the exploration process. The allocation constraints provided to VSS is also shown in the figure.

The output produced by VSS is fully simulatable. However the amount of VHDL code is quite large since the details are at the RT level. The VHDL code produced by VSS for one of the FPGAs was about 2500 lines, and the total lines of VHDL for all the FPGAs was over 10,000 lines. However this code was directly simulated on a commercial simulator to verify correctness of the synthesized design.

3.5 Technology Adaptation

The RTL VHDL output from VSS was the input for commercially available logic synthesis tools which produce gate-level schematics from the RT level. We used the logic synthesis tool from Exemplar for synthesizing the gate level design. The target synthesis architecture were the Xilinx FPGAs. A few minor modifications in the VHDL description of some of the GENUS components was necessary, in order to enable the logic synthesis tool to incorporate technology specific macros or semi-custom cells for complex components such as fast adders and memories into logic synthesis. These macros or cells



Design Constraints	
#Adders	1
#Comparators	1
#RAMs (1R, 1W)	2

Quality Metrics (Arch. Design)	
Estimated #Gates	7100
Performance	648 clks

Figure 8: Architectural Design for one FPGA (EVAL_R1)

greatly enhanced the performance of the system when compared to synthesizing the complex functions from scratch.

The technology-dependent schematic was the input for place and route tools, the final piece of the design puzzle. Another commercial tool from NeoCAD was used for placement and routing of the FPGAs. The role of these tools are well known. After place and route a backannotated circuit schematic was generated for final digital simulation. The digital simulation incorporated timing delays due to logic components and routing delays. This provides final verification before proceeding to actual silicon. The total time spent in the final phase of the design process was about 5 days.

However, during the final digital simulation of the back annotated design it was found that the wire delays were very large (almost 75 ns), making it impossible to satisfy the clock constraint of 100 ns. Thus a second iteration through the design process was necessary.

DESIGN DESCR	Num Lines of SpecChart Code	50 lines
	Lines of VHDL after SpecChart	100 lines
	Lines of VHDL after VSS	2500 lines
COST	FPGA type	Xilinx 4000
	Number of CLBs	360
	Number of Gates	9K
CLOCK	Clock Cycle Constraint	200 ns
	Clock Cycle After VSS	70 ns
	Clock Cycle After NeoCAD	145 ns
PERF	Performance Constraint	200 us
	Perf. Estimated by SpecSyn	155 us
	Perf. Achieved by VSS	155 us
	Perf. after NeoCAD	180 us

Figure 9: Results: EVAL_R1

3.6 Second Design Iteration

The design was resynthesized with a clock cycle of 200 ns. In Figure 9 we show some of the synthesis characteristics for one of the FPGAs (EVAL_R1). The final design occupied 360 CLBs in the FPGA. The clock constraint of 200 ns was achieved after placement and routing of the FPGA. The total time required to evaluate each rule was 180 us. The design produced during the second iteration was able

to run through the simulation successfully Figure 10.

Since the synthesis process was rerun from the VSS level onwards, another 9 days of work was required before the chip passed through the test vectors. In future the quality of the area and delay estimation tools have to be improved to avoid this second iteration of the design.

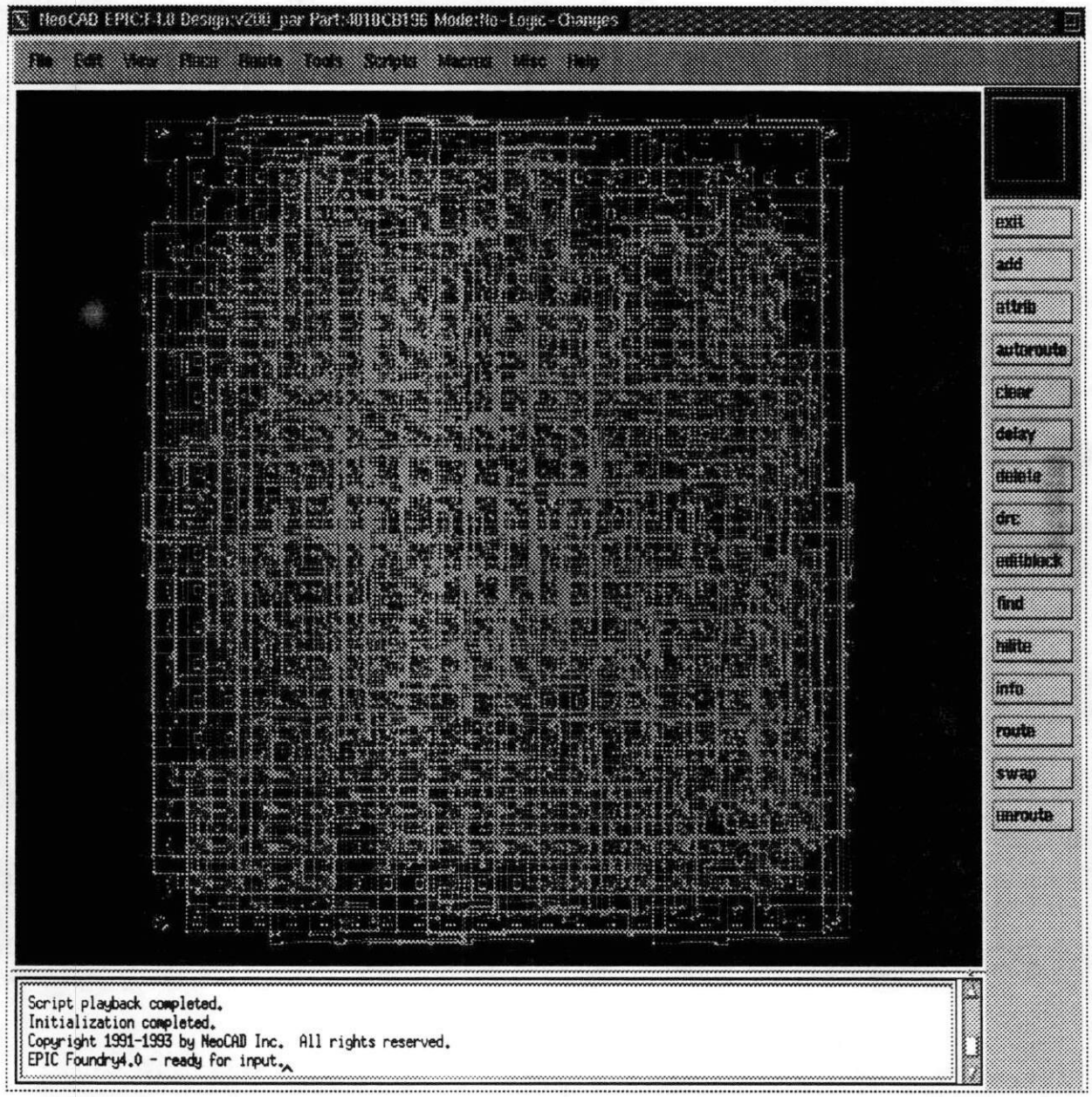


Figure 10: Placement and Routing Results : EVAL_R1

4 Conclusions

In this paper we have discussed a design methodology aimed at producing quick design turnarounds. We have illustrated the power of this methodology with an industrial design, that was (a) specified at the highest levels of abstractions using SpecCharts (b) synthesized at the system level with SpecSyn (c) further synthesized at the RT level using VSS and finally (d) implemented with FPGAs using commercial tools.

In this experiment the design quality is comparable to manual designs, but the design time is orders of magnitude less. Since the designer was dealing with higher levels of abstractions, a large design space could be explored by quickly changing some of the exploration parameters. For example, numerous partitionings in the system level and numerous microarchitectures were quickly designed (in a few minutes). Providing hooks to VHDL, allowed verification of the designs at each of the abstraction levels.

We are currently working on improvements to achieve a design cycle time of 100 hours, which will make this methodology extremely useful in cases where designs have to be delivered within a few weeks. To meet this demand, we envision that the design process would automatically move to higher abstraction levels, where the amount of detail is at a minimum.

5 Acknowledgements

The design of the fuzzy logic controller was supported by a grant from Matsushita Electric Works Research and Development Laboratory (San Jose, California). We thank their support. We would also like to thank Yoshii Shimmei (Matsushita Electric Works Research and Development Laboratory), for his suggestions and useful discussions during the course of this project.

We are extremely grateful to the Semiconductor Research Corporation, for funding the development of the SpecSyn and the VSS projects (Grant 92-DJ-146).

6 References

[1] D. Gajski, F. Vahid, and S. Narayan, "SpecCharts: A VHDL Front-End for Embedded Systems." Technical Report93-31, 1993.

[2] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A Language for System Level Synthesis," in *Proc. of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[3] *IEEE Standard VHDL Language Reference Manual*, 1988.

[4] D. Gajski, F. Vahid, and S. Narayan, "A System-Design Methodology: Executable-Specification Refinement," in *Proceedings in European Conference on Design Automation*, 1994.

[5] Daniel Gajski, Nikil Dutt, Allen Wu and Steve Lin, *High Level Synthesis.* Kluwer Academic Publishers, 1992.

[6] P. Jha, T. Hadley, and N. Dutt, "The GENUS User Manual and C Programming Library," technical report, Dept. of Information and Computer Science, University of California, Irvine, CA 92717, 1993. Technical Report : 93-32.

[7] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, "An algorithm for array variable clustering," in *Proc. of the EDAC94 Conference*, Feb 1994.

[8] L. Ramachandran and D. D. Gajski, "Architectural Tradeoffs in Synthesis of Pipelined Controls," in *Proc. of the European Design Automation Conference*, September 1993.

[9] Xilinx, *The Programmable Gate Array Data Book.* 1992.

[10] G.J.Klir and T.A.Folger, *Fuzzy Sets, Uncertainty and Information.* Prentice Hall, 1988.

[11] G.C.Lee, "Fuzzy Logic in Control Systems: Fuzzy Logic Controller - Parts I and II," *IEEE Transactions on Systems, Man and Cybernetics*, pp. 404–435, March/April 1990.

[12] F. Vahid and D. Gajski, "Specification Partitioning for System Design," in *Proc. 29th DAC*, 1992.

[13] L. Ramachandran, N.Holmes, and D. Gajski, "VSS-AA: VHDL Synthesis System with Architectural Allocator. Human Interface and the Design Process," tech. rep. 94-04, Dept. of Information and Computer Science, Univ. of California, Irvine, CA 92717, 1994.