Application-driven Design Automation for Microprocessor Design*

Iksoo Pyo, Ching-Long Su, Ing-Jer Huang, Kuo-Rueih Pan, Yong-seon Koh,
Chi-Ying Tsui, Hsu-Tsun Chen[†] Gino Cheng[‡] Shihming Liu, Shiqun Wu, Alvin M. Despain
Department of Electrical Engineering - Systems
University of Southern California
aspian@zelea.usc.edu

Abstract

ADAS is an Application-driven Design Automation System for microprocessor design. The goal of ADAS is to automatically explore the design space and synthesize a single chip VLSI processor from a high-level specification of the Instruction Set Architecture (ISA) written in a subset of standard Prolog. Our idea is to develop a design automation system which considers both microprocessor hardware design and design of the corresponding language compiler concurrently. Benchmark programs are used to motivate design decisions and optimize performance. Compiler optimizations are considered during the design of hardware. Our system spans language design, compiler design, instruction set design, microarchitecture, and VLSI implementation. Another goal of our project is to determine the feasibility of applying formal methodology to design automation and the usefulness of formal syntax and semantics to define the meaning of specifications. We have exercised our system on a real industrial example, the TDY-43 processor.

1 Introduction

In this paper, we present an overview of the ADAS system. ADAS is an extension of the ASP system developed at Berkeley [1]. Many changes were necessary in order to adapt ASP to the needs of our industrial partners. In addition, many other design tools were integrated to enhance the utility and to improve the efficiency and performance of ASP. Throughout ADAS, we have employed formal notations and methods to aid us in controlling the system development and to aid us in formal validation of designs.

ADAS accepts a specification of the Instruction Set Architecture (ISA) as input, and produces both layout specified in Caltech Intermediate Form (CIF), and a re-order table for the language compiler as output. In ADAS, the design process has been partitioned into 45 design stages. These design stages are categorized into three major domains - the behavioral domain, the logic and circuit domain, and the geometric domain. The ADAS system is shown in Figure 1.

From an abstract ISA specification, Fiper and Piper determines the hardware resources and the control necessary

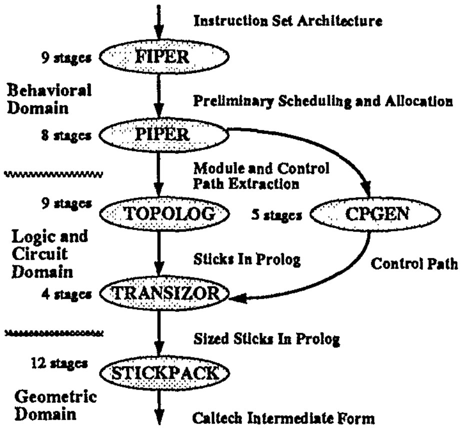

Figure 1: The Major Domains of ADAS

to implement the instructions. Important features of our behavioral synthesis system will be discussed in sections 3, 6 and 11.

CPGen performs logic minimization on the controller specification and creates a symbolic layout implementation for the controller. Details of CPGen will be discussed in section 7.

In Topolog the hardware functions are implemented in a datapath which is translated into a symbolic layout. The major tasks in Topolog are cell allocation, cell placement, and routing. In section 8, we describe this process.

Circuit performance is enhanced by TranSizor, an automatic transistor sizing program. Transistor sizes are adjusted such that the delay constraints can be satisfied. In Section 9, a detailed explanation of TranSizor is presented.

Symbolic layout is translated into mask geometries, placed on a die, and routed by StickPack. A pad frame for the die is also created automatically. These procedures will be discussed in section 10.

2 Design Philosophy and Framework

2.1 ADAS Structure

The design issues of ADAS are:

1. How do high level implementation methods influence low level design decisions?
2. How is new design knowledge incorporated into an automated design environment?

*This research was partially supported by the California Competitive Technologies Program Grant No. C88-150, the Defense Advanced Research Projects Agency under Contract No. N00014-88-K-0579, TRW/ESL, and Teledyne Systems Company.

†FM3-95, Intel, 1900 Prairie City Road, Folsom, CA 95630
‡visitor from University of California, Berkeley

3. How can information derived at high levels of design be used at lower levels to improve design quality?

4. How can an automated design environment efficiently redesign an implementation which does not meet specification?

In order to better understand these problems, we have developed a hierarchy of formal representation languages which span the entire gamut of processor design. Traditionally, the design problem has been decoupled into relatively independent design tasks, each of which is automated separately. Tools focus upon very specific design problems such as detailed routing, compaction, logic synthesis, and resource allocation without regard for the overall design. In ADAS, the design task is viewed as a sequential series of design transformations and is optimized as such. Design techniques at one stage of design may be used to simplify tasks at other stages. For example, higher level design problems such as resolving scheduling conflicts can be solved by providing new lower level components such as a priority generation module.

At each stage of the design process, the design specification is successively transformed into equivalent, more detailed representations, and ultimately to one which can be manufactured. Breaking the design process into several stages simplifies the overall design problem and makes it more manageable. Conceptually, each stage consists of a Semantic Design Generator (SDG), a simulator, an evaluator, a translator, a design library, and a design rule module as shown in Figure 2.

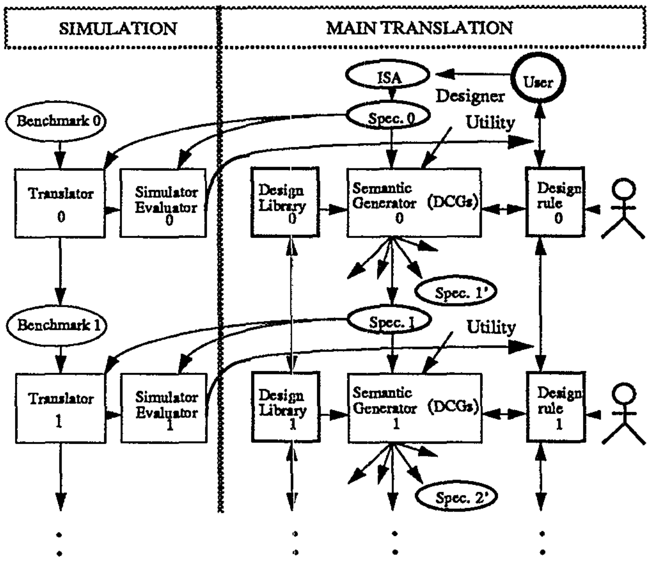

Figure 2: The Conceptual Structure of ADAS

The semantic design generator decomposes a design specification into more primitive components by applying transformations specified in the design rule module. For a given design, many alternatives may be possible. The most favorable transformation is selected by evaluating each alternative with the metrics important to that design stage.

2.2 Benchmark Driven

ADAS is a benchmark driven synthesis system. A collection of benchmark programs [2] are provided with the initial design specification. These programs are compiled into machine code and ultimately to binary test sequences. Benchmark program representations at the behavioral, RTL, and switch level are applied to the design specification at the appropriate level. By comparing simulation results, design consistency can be verified. Information extracted from benchmark programs is also used to motivate design decisions. Performance evaluation from benchmarks is more realistic than the best and worst case analytic estimation models.

3 Formal Methods

To ensure that a design specification implements the behaviors intended by the designer, it is important that the design specification languages throughout the design process be formal. In order to formalize a language, the syntax and semantics of the language must be formally defined.

In ADAS we used the Definite Clause Grammar (DCG) - a form of attribute grammar - to formally specify both the syntax and semantics of a design language. The syntax of these languages is specified by a BNF (Backus Normal Form) grammar. The semantic specification of a design is defined by attaching well defined "semantic attributes" to each production of the BNF grammar.

In a DCG system the correctness of a design specification may be verified by parsing. During parsing, a design which is represented in terms of "non-terminal" structures, is decomposed to a collection of "terminals". Terminals are objects for which implementation methods exist at the next stage of the design process. The decomposition process is guided by well defined translation rules.

As an example, valid behavioral specifications are verified by a parser which decomposes the specification into a list data structure, checks the syntax, verifies linear recurrence, and applies rules which translate the specification into one which is read at the next design stage. A fragmentation of the ISA parsing code is described in [3].

4 The Specification of Behavior

The design description for the ADAS system is specified in subset of Prolog. In this manner, instruction set architecture specifications can be both *simulated* and *synthesized*. This general approach is similar to that taken with the MacPitts system for LISP [4] and with the Flamel system for Pascal [5]. The descriptive power of ADAS corresponds to that of ISPS [6].

The specification of a design, the SM1 (a simple eight instruction processor) is illustrated in Figure 3. During each instruction cycle, an opcode and address are fetched, the PC is incremented, and the instruction specified by the opcode is decoded (implicitly) and executed. The behavior of each instruction is specified in an *execute* clause. The specification is directly *executable* in Prolog, thus special simulators are not necessary.

```
% Symbolic representation of Instruction Set Architecture

sm1( AC, PC , Memory ):-
      fetch( PC, Op, Adr, Memory ),
      P1 is PC + 1,
      execute( [Op], Adr, AC, P1, Memory, AC_next, PC_next,
               Memory_next ),
      sm1( AC_next, PC_next, Memory_next ).

fetch( PC, Op, Adr, Memory ):-
      memory_read( PC, Memory, [Op,Adr] ).

execute( [add], X, AC, PC, Memory, AC_next, PC, Memory ) :-
      MemAR <- [constant(0)/15-13,X/12-0],
      memory_read( MemAR, Memory, MemDR ),
      AC_next is AC + MemDR.

execute( [and], X, AC, PC, Memory, AC_next, PC, Memory ) :-
      MemAR <- [constant(0)/15-13,X/12-0],
      memory_read( MemAR, Memory, MemDR ),
      AC_next is AC ∧ MemDR.

execute( [shr1], X, AC, PC, Memory, AC_next, PC, Memory ) :-
      AC_next is AC >> 1.

execute( [store], X, AC, PC, Memory, AC, PC, Memory_next ):-
      MemAR <- [constant(0)/15-13,X/12-0],
      memory_write( MemAR, AC, Memory, Memory_next ).

execute( [load], X, _, PC, Memory, AC_next, PC, Memory ):-
      MemAR <- [constant(0)/15-13,X/12-0],
      memory_read( MemAR, Memory, MemDR ),
      AC_next = MemDR.

execute( [jump], X, AC, _, Memory, AC, PC_next, Memory ):-
      PC_next <- [constant(0)/15-13,X/12-0].

execute( [brn], X, AC, PC, Memory, AC, PC_next, Memory ) :-
      (AC < 0 -> PC_next <- [constant(0)/15-13,X/12-0];PC_next = PC).

class([reg,master_slave,r1,16]).
class([memory,ram,m1,[16,64]]). % 16bit x 64K

attributed([r1,ac,r,sm1,1]).
attributed([r1,pc,r,sm1,2]).
attributed([m1,m, m,sm1,3]).
```

Figure 3: Instruction Set Architecture of SM1

Instructions must be deterministic (with only shallow backtracking), and linear tail-recursive. True recursion is not allowed. Furthermore, only a limited set of built-ins and user-defined predicates are currently supported. However, special predicates are provided for creating and manipulating registers, bit fields, and for managing the state those registers contain.

5 Simulation, Analysis and Verification

Benchmark programs [2] specified in the ISA of a processor are compiled into target machine code which is applied to simulate the design specification through different abstract design levels.

Simulation of designs at the functional and behavioral level are based upon an event-driven finite-state machine model. Benchmark programs are simulated cycle by cycle. Clock ticks and interrupts represent events.

An analyzer calculates the frequency of both individual instructions and sequences of contiguous instructions. Results are used to optimize scheduling and resource allocation in behavioral synthesis. Frequently used instructions are given priority during scheduling while infrequent instructions are scheduled to maximize resource sharing. Data dependencies determined by instruction pattern analysis are used to evaluate latency in pipelined designs.

Test vectors automatically produced from the benchmarks are applied to networks extracted from the layout. Machine state traces are recorded. Verification programs compare these traces with the results predicted from the high level specification using the circuit-level simulation tools - *ESIM*, *IRSIM* and *SPICE*.

6 Behavioral Level Synthesis

6.1 Behavioral Level System Overview

After parsing an ISA in section 3, Fiper creates a global control/data graph describing the execution flow of each instruction by dependency analysis. Based upon this graph, instructions are scheduled by applying the scheduling/allocation algorithm illustrated in Figure 4. This algorithm attempts to minimize the number of states necessary to traverse the most frequently used RTL paths based upon the given *latency*, *stage-time limit* and *the available resources*. Current scheduling/allocation algorithms focus on the reduction of hardware resources and the cycle time independent of program application. By using instruction frequency information, resources are allocated and scheduled to reduce time for critical instructions. Fiper finally produces an optimized state transition graph which describes the sequential behavior of the machine under design.

```
Step 1: Evaluate frequencies of RTL patterns of fixed latency
        and Create an allocation table.

N = 1.
Repeat until all RTL patterns are scheduled
Step 2: Find path(s) which has the most frequently used RTL pattern
Step 3: Try to schedule the pattern within N stage(s).
        If fail, N is N + 1 and go to Step 3.
Step 4: Update the allocation table.
        Delete the scheduled path (RTL sequence).
        N = 1.
End Repeat
```

Figure 4: ADAS Scheduling/Allocation Algorithm

From the state transition graph produced by Fiper, Piper performs pipeline dependency analysis and decouples the state graph into stages, each stage implementing concurrent operations (pipeline stage assignment). Dependencies between pipeline stages are detected and resolved. Operations at each stage are then bound to functional units for a data path allocated. Additional hardware resources are allocated to resolve scheduling conflicts and to control buses.

Functional units in the data path are described in terms of a netlist of modules. The control is represented by symbolic boolean equations and special hardware modules which resolve scheduling conflicts. Additional hardware is also needed to interface the data path with the control path.

6.2 Important Features

Different pipeline latencies will result in designs which vary in size and performance. Piper applies the benchmark set to evaluate both the peak performance and the average performance of a pipelined processor for several latencies and selects the latency that is most cost and performance effective. Examples are described in section 11.

In a pipelined design, the dependency between consecutive instructions may prevent the processor from operating at its maximal instruction firing rate [7, 8]. These dependencies are also dependent upon the latency of the pipeline. The performance of a pipelined design may be improved by properly inserting NOPs between instructions or, preferably, reordering instructions to eliminate dependencies. Piper solves the dependency problem by creating priority modules to handle the physical signal conflicts and generating reorder information for the language compiler.

More details of benchmark evaluation and the derivation of micro-architecture dependent back-end compilers can be found in [9].

7 Control Path Synthesis

CPGen is the control path generator for both pipeline and non-pipeline designs. A data stationary control model [10] is used for pipeline designs. In this model the instruction opcode is transmitted through each stage of the pipeline in synchronization with the data. Control signals for each stage are generated by decoding the instruction which corresponds to that stage.

Rather than encoding the next state address of a state transition graph in the combinational logic, the address may be encoded with a counter triggered by the clock. In this manner, state changes as the counter is incremented. The number of counter cycles is determined by maximum number of clock cycles necessary to implement an instruction. This scheme is particularly useful for pipeline designs which use separate controllers for each pipeline stage, they may be synchronized by the sequence counter.

Pipelined designs provide a natural partitioning for large controllers. Unfortunately, a controller may still be too large to be implemented in a single PLA. Another level of logic partitioning is done to further subdivide the control logic such that a multiple PLA implementation is possible. Also, by partitioning a PLA into smaller components, a better chip floorplan is possible.

8 Data Path Synthesis

Given a register-level specification of functional units, Topolog produces a symbolic layout in Sticks in Prolog (SIP). This is achieved by reducing the functional units to gates which are defined by transistor netlists, placing the transistors onto a bit slice, and routing together signals which are electrically equivalent [11]. Figure 5 shows an overview of Topolog.

For each function supported by Piper, a module specification exists in Topolog which translates a RTL representation of the function into a gate-level description. In addition to defining the gate structure, the module specification also distributes gates into the appropriate bit-slices. The modules supported by ADAS include logic gates, tristates, ALUs, adder/subtractors, multiplier, divider, encoder/decoder, mux/demux, priority encoder, shifter, barrel shifter, comparators, registers, shift registers, clock divider, etc.

By representing gates in terms of transistor netlists rather

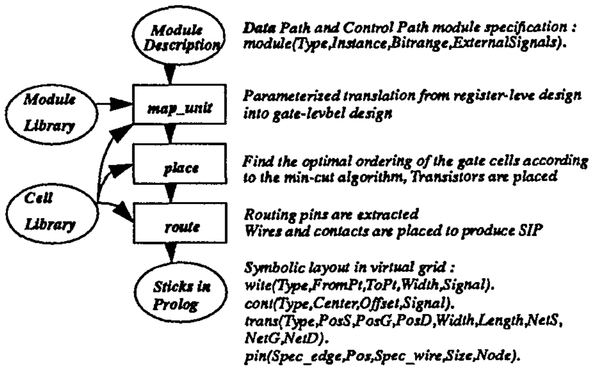
Module Description
Data Path and Control Path module specification :
module(Type,Instance,Bitrange,ExternalSignals).
Module Library
map_unit
Parameterized translation from register-leve design
into gate-levbel design
place
Find the optimal ordering of the gate cells according
to the min-cut algorithm, Transistors are placed
Cell Library
route
Routing pins are extracted
Wires and contacts are placed to produce SIP
Sticks in Prolog
Symbolic layout in virtual grid :
wite(Type,FromPt,ToPt,Width,Signal).
cont(Type,Center,Offset,Signal).
trans(Type,PosS,PosG,PosD,Width,Length,NetS,
NetG,NetD).
pin(Spec_edge,Pos,Spec_wire,Size,Node).

Figure 5: Topolog System

than layout cells, greater layout density is possible by merging connected transistor chains between gates.

Transistor islands are placed in a bit slice to minimize the routing density. A placement tool based on the Min-Cut algorithm [12] has been implemented. The global-max gain pair is obtained by searching all possible pairs from partitioned groups. The local-max gain pair is obtained by searching the max-gain cell from each partitioned group. The performance of the algorithm has been improved by dynamically determining the pair of cells to interchange [13]. Consequently, the complexity of the placement algorithm improves from O(N^2) to O(N) while still producing acceptable solutions. Layout quality is further refined by applying post-placement which is non-pair-wise-exchange after transistor islands have been placed by Min-Cut. Results are shown in Table 1.

In addition to the number of routing tracks, timing is also considered during placement. The critical paths through the datapath are identified from the benchmark programs and placed together to reduce delay.

IMPROVED MIN-CUT	MC	TC	CPU Time
Global Max.	15	1294	756s
Local Max.	24	1844	11s
Dynamic GL Max.	21	1670	96s

a) Modified Min-Cut on SM1 bit-5 (134 cells)

PLACEMENT	MC	TC
Min-cut Placement	21	1670
Post-Placement	17	1306

b) Post-Placement on SM1 bit-5 (134 cells)

GL: Global/Local, MC: Max Cut, TC: Total Cut
MC and TC are defined in [14].

Table 1: Improvement result

After placement, gate netlists are expanded into symbolic layout and routed. The signals within each bit slice are routed by a modified left edge first channel router [15] while control signals between adjacent bit slices are connected together by river routing [16]. During placement, location of feed-through blocks is placed to reduce the number of routing tracks necessary for river routing.

9 A Transistor Sizing

ADAS includes a transistor sizing program named TranSizor for optimizing the delay of symbolic layout components within an area constraint. Given a layout specification in a sticks language, TranSizor produces a symbolic layout specification with sized devices.

TranSizor uses high level information provided by Piper in order to determine the critical paths of a circuit. This is computationally less expensive and more accurate than deriving all possible circuit paths as many false paths, critical paths which do not occur, are not considered.

TranSizor uses PTA (Prolog Timing Analyzer) to calculate the time delay of each path. Using a lumped RC delay model [17], PTA produces symbolic equations for each delay path. Equations are differentiated symbolically to avoid expensive numerical computation. The resulting equations are solved to find the transistor sizes which yield the minimum delay. Two techniques, simulated annealing [18] and critical-path heuristic [19] are used to resize the transistors along each critical path.

10 Geometrical Level Synthesis

Given a collection of symbolic layout modules [20] created by Topolog and CPGen and modified by TranSizor, StickPack produces, a technology dependent mask layout description of the chip in CIF. StickPack contains a number of design tools: a technology-independent compactor, a power/ground bus size estimator, a joiner that joins together cells by pitchmatching and river routing, a global placer, and a global router and a channel router.

A technology independent split-grid compactor [21] is used to compact symbolic layout cells before they are placed and routed. Novel features such as automatic wire jog insertion, contact offsetting, and wire minimization are incorporated in the compactor. Hierarchical compaction is supported. The chip design is described in a hierarchical manner. Connecting pins between cells are identified. Constraints are then added during the compaction process to make sure that the abutting pins are in the same physical location after compaction. A hierarchical symbolic editor is developed to help the user the describe the design hierarchically.

Once the symbolic layout modules have been compacted, they are placed and routed to form the final chip, a constructive initial placement algorithm is used for the macromodule placer. Slicing structure [22] is maintained during the placement phase to facilitate the later channel definition and channel ordering in the routing stage. After all the blocks are placed, a global router which uses a depth first search algorithm to find the shortest path of connecting each net is called to assign nets to the channels. A detail router based on left edge greedy algorithm is called to route each channel in the order pre-determined by the slicing structure.

11 Synthesis Example

Figure 6, Figure 7, and Figure 8 show examples of two-stage pipelined SM1, six-stage pipelined SM1, and two-stage pipelined SM3 which are synthesized by ADAS, where SM stands for simple machine. SM1 has eight instructions with about 6K transistors. SM3 has forty instructions with about 10K.

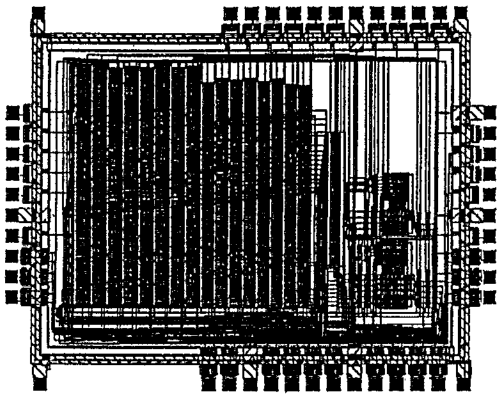

Figure 6: SM1 2 Pipeline-Stage Plot

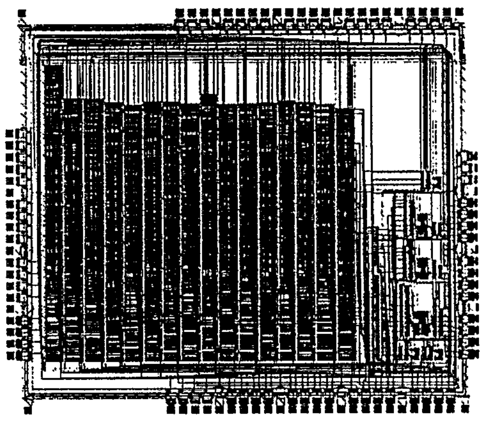

Figure 7: SM1 6 Pipeline-Stage Plot

Figure 9 shows the estimated hardware cost (transistor count), performances (peak and average), and performance/cost ratios for six different designs of SM1 produced by Piper. The pipeline configuration of each design is denoted by C/L, where C and L stand for the elapsed time of instruction and the latency respectively. We chose speed up as the performance metric, assuming the same system clock rate. A straight-line instruction benchmark with conditional branches is used to estimate average-case performance. With proper scheduling and allocation, it is possible to increase the throughput (instruction firing rate) without a significant increase of cost, and even at no extra cost. The design 6/1 has the highest peak-performance per unit cost, while designs 6/1 and 6/2 (which is a 3 stage pipeline with instruction latency of two clocks) have the highest average-performance per unit cost.

An implementation of the SM1 processor has been submitted to MOSIS for fabrication. ADAS has also successfully

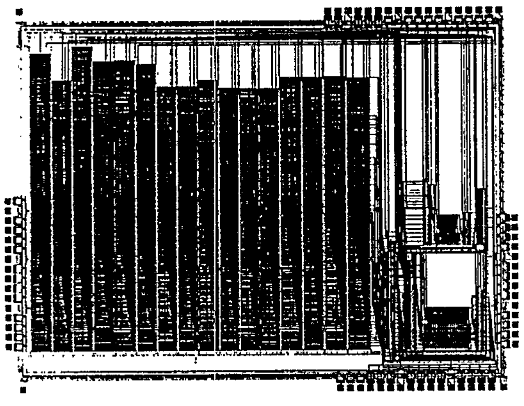

Figure 8: SM3 2 Pipeline-Stage Plot

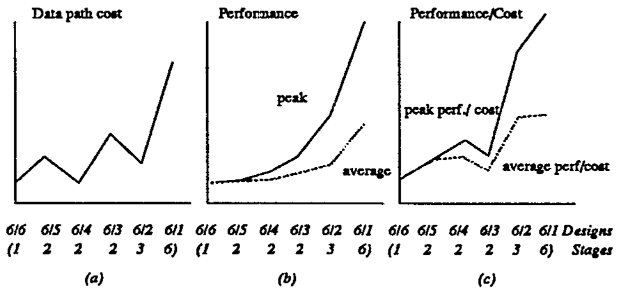

Figure 9: Cost/Performance

implemented the TDY-43, a 250 instruction processor which supports eight addressing modes. From an ISA specification, ADAS produced a gate-level VHDL netlist for implementation by gate array. The design also implements a JTAG standard scan testing protocol. The implementation produced by ADAS requires 34K gates or 380K transistors and has been verified at the gate-level using benchmark-driven test vectors.

12 Conclusion

In this paper, we have presented a full range microprocessor design automation system. It is demonstrated that by giving an instruction set, ADAS can automatically generate layout of a single chip microprocessor. By utilizing the benchmark simulation data, high level synthesis tools provide valuable information to guide the low level synthesis. Future research on ADAS includes further exploration of the interaction between different levels of design tools, capturing design knowledge for the redesign to meet constraints and extending the scope to multi-chip module (MCM) design.

Acknowledgements

We would like to thank Dave Boyer, Bill Bush, Angela Cheng, Rick McGeer, Dylan McNamee, Jonathan Pincus, Wes Reed, and Steve Schoetler for their contributions to this research and Paul A. Suhler for his contributions to this paper.

References

[1] Alvin M. Despain, "The Design System (ASP) of the Aquarius Project," *Proceedings of the Second International Workshop on VLSI Design*, pp. 149-166, Dec. 1988.

[2] Ralph Haygood, "A Prolog Benchmark Suite for Aquarius," *Report No. UCB/CSD 509*, 1989.

[3] Gino Cheng, et al., "A Full-Range Design Automation for Instruction Set Processors," *First International Conference on the Practical Application of PROLOG*, 1992.

[4] Jay R. Southard, "MacPitts: An Approach to Silicon Compilation," *IEEE Computer*, pp. 74-82, Dec. 1983.

[5] Howard Trickey, "Flamel: A High-Level Hardware Compiler," *IEEE Trans. CAD*, pp. 259-269, March 1987.

[6] Mario R. Barbacci, "Instruction Set Processor Specifiations (ISPS): The Notation and Its Applications," *IEEE Trans. Computer*, pp. 24-40, Jan. 1981.

[7] Nohbyung Park, et al., "Sehwa: A Program for Synthesis of Pipelines," *23rd IEEE Design Automation Conference*, pp. 454-460, March 1986.

[8] Michael C. McFarland, et al., "Tutorial on High-Level Synthesis," *25th IEEE Design Automation Conference*, pp. 330-336, Feb. 1988.

[9] Ing-Jer Huang and Alvin Despain, "High Level Sÿnthesis of Pipelined Instruction Set Processors and Back-End Compilers," *29th IEEE Design Automation Conference*, June 1992.

[10] P.M. Kogge, "The Architecture of Pipelined Computers," *McGraw-Hill Book Company*, 1981.

[11] P. Kollaritsch and N. Weste, "A Rule-Based Symbolic Layout Expert," *VLSI Design*, pp. 62-66, Aug. 1984.

[12] B. W. Kernighan, et al., "An efficient heuristic procedure for partitioning graphs," *Bell Sys. Tech. J.*, pp. 291-308, Feb. 1970.

[13] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *19th IEEE Design Automation Conference*, pp. 175-181, 1982

[14] Melvin A. Breuer, "Min-cut Placement, VLSI circuit layout theory and design," *IEEE press*.

[15] Takeshi Yoshimura, et al., "Efficient Algorithms for Channel Routing," *IEEE Trans. on CAD of Integrated Circuit and Systems*, Vol. CAD-1, No. 1, Jan. 1982.

[16] C. P. Hsu, "General River Routing Algorithm," *20th IEEE Design Automation Conference*, pp. 578-583, 1983.

[17] J. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Trans. CAD*, Vol. CAD-4, No. 3, July 1985.

[18] Jonathan D. Pincus and Alvin M. Despain, "Delay Reduction Using Simulated Annealing," *23rd IEEE Design Automation Conference*, pp. 1986.

[19] W. Kao, "Algorithms for Automatic Transistor Sizing in CMOS Digital Circuits," *22nd IEEE Design Automation Conference*, pp. 781-784, 1985.

[20] N. Weste, "Virtual Grid Symbolic Layout," *18th IEEE Design Automation Conference*, pp. 225-233, 1981.

[21] David G. Boyer, "Split Grid Compaction for a Virtual Grid Symbolic Design System," *ICCAD*, pp. 134-137, 1987.

[22] R. Otten, "Automatic Floorplan Design," *19th IEEE Design Automation Conference*, pp. 261-267, 1982.