HERCULES - A System for High-Level Synthesis

Giovanni De Micheli David C. Ku
Computer Systems Laboratory
Stanford University

Abstract

This paper presents an approach to high-level synthesis of VLSI processors and systems. Synthesis consists of two phases: behavioral synthesis, which involves implementation-independent representations, and structural synthesis, that relates to the transformation of a behavior into an implementation. We describe HERCULES, a system for high-level synthesis developed at Stanford University. In particular, we address the hardware description problem, behavioral synthesis and optimization using a method called the *reference stack*, and the mapping of behavior onto a structure. We present a model for control based on sequencing graphs that supports multiple threads of execution flow, allowing varying degree of parallelism in the resulting hardware. Results are then presented for three examples: MC6502, Intel8251 and FRISC, a 16-bit microprocessor.

1 Introduction

The goal of this project is to provide a system for the computer-aided design and synthesis of digital systems, more formally called *high-level synthesis*. High-level synthesis transforms a behavioral specification of hardware in terms of hardware description language (HDL) descriptions into a structural interconnection of hardware units that may be mapped effectively to a VLSI implementation.

High-level synthesis systems have been the object of extensive investigation both in the academic environment [7,8,11,16,13,14,15] and in industry [1,5]. Hercules is developed in the frame of the Stanford OLYMPUS synthesis project, which consists of three major components: *high-level, logic*, and *physical synthesis*, as depicted in Figure 1. Synthesis can be seen as a pipeline through each component of the design system. Each component performs transformations on a given representation, the result of which is a new representation that may be processed by subsequent stages. Feedback in the form of estimates on area and timing is provided by the logic and physical synthesis tools to effectively guide optimizations during high-level and logic synthesis. High-level synthesis is pivotal in determining how effectively the spectrum of tradeoffs between performance and area can be explored.

This paper describes an approach to high-level synthesis, focusing primarily on the modeling of hardware behavior, the transformations that preserve the functionality without structural implications, and the mapping of behavior into a structure. Section 2 gives an overview to the Hercules system. Section 3 presents the language we used for hardware description. Section 4 describes the abstraction and optimizations used by behavioral synthesis. In particular, we present a method called the *Reference Stack* that is used to simplify both behavioral optimizations as well as subsequent structural mapping. Section 5 describes structural synthesis, the transformation of behavior into a graphical intermediate form, and the mapping of the intermediate form onto a structure. Finally, we present the results of applying this methodology to several benchmark examples: Motorola 6502 microprocessor, Intel 8251 UART, and FRISC, a simple example proposed in [14].

2 System Overview

High-level transformations may be classified as either *independent* or *dependent* on the structural implementation. In the first case, the set of transformations is called **behavioral synthesis** since it operates on the behavior of the circuit. The latter is called **structural synthesis**, and is a transformation of a hardware behavior into a hardware structure.

At each level of synthesis, there is a corresponding intermediate representation that serves as the abstraction on which optimizing transformations may be carried out. For behavioral synthesis, the abstraction is a recursively defined *parse tree* that supports local scoping of variables and hierarchical interconnection. For structural synthesis, the abstraction is a sequencing graph with multiple threads of execution flow.

Behavioral synthesis involves optimization techniques similar to those used in optimizing compilers, such as dead code elimination, variable unfolding, and constant unfolding. In addition, it includes hardware-oriented optimizations such as procedure in-line expansion, loop unrolling, and meta-variable evaluation. We present a method in Section 4.2 called the *Reference Stack* that is used to efficiently resolve variable unfolding, constant unfolding, conditional assignments, and multiple assignments in a one pass transformation on the parse tree representation. In addition to the objectives stated above, the reference stack may also provide information to structural synthesis which will minimize the number of registers needed to implement the functionality of the input description, as well as eliminate the control steps needed to implement assignments to variables.

Once behavioral synthesis is completed, **structural synthesis** maps the optimized parse tree into a sequencing graph model called the *sequencing intermediate form*, or SIF for short. Note that at this point, the resulting SIF may be immediately transformed into an implementation. However, the result may not be acceptable from a performance and/or area standpoint. This is not surprising, since area and delay models have not been accounted for up to this point. This leads to a major emphasis of Hercules - *iterative refinement* of the structural model, guided by the estimates on area and timing provided by logic synthesis. This iterative refinement approach involving both high-level and logic synthesis in a tightly interacting loop is an innovative approach, and is selected for the following reasons.

- We want to resolve a cyclic condition on the sequence of the synthesis steps. A good estimate of area/delay can be done after logic synthesis. However, logic synthesis can be carried out only after the functional blocks have been identified. A good partition into functional blocks in turn requires the knowledge of the system cycle time, which depends on the module delays.
- Logic synthesis algorithms [4,17] have shown to be powerful means to reduce the complexity of the logic circuits. We have developed a model for logic synthesis that allows the representation of our modules (which are arbitrary sequential circuits) and we exploit logic optimization algorithms early in the synthesis process.
- We feel that some implementation decisions should be made by comparing views of the circuit in different domains, such as the behavioral, structural and logic.

CH2540-3/88/0000/0483$01.00 © 1988 IEEE

For these reasons, we feel that structural synthesis and logic synthesis should be tightly coupled. In our approach, we synthesize first the data path and control under the assumption that combinational logic takes zero delay and that no timing constraints (other than sequencing) are imposed on the structure. Logic synthesis techniques [4] are then applied to this initial structure to evaluate the area and delay figures of merit. This information, together with external constraint information, are used to derive a more refined *control structure* and/or modify the existing structure of the system.

The circuit representation is stored in a logic intermediate form developed at Stanford, called SLIF (*Structural/Logic Intermediate Form*). SLIF supports hierarchical structure of blocks, netlist interconnections, complex components such as latch and tristate elements, as well as the standard logic operations.

3 Behavioral Description

The goal of high-level description languages is to make possible the mapping of the *behavioral* characteristics of a system into its *physical* counterpart. For software systems the physical counterpart is the executable object code, whereas for hardware synthesis systems it is the physical layout of the chip.

We envision the entry level description of a system as consisting of two sets of specifications: i) a *behavioral description* that describes the function of the system to be realized without committing to an implementation, and ii) a set of *constraints* on the design. The constraints are related to the technology being used and to the interfacing of the system with the outside environment. The second set of specifications, as considered by [3] allows the designer to specify upper and lower bounds on the time difference between events corresponding to signal transitions.

There has been a great deal of debate on the style of behavioral language best suited for hardware design. The result is a proliferation of hardware languages that are too simulation oriented and hence overly baroque for synthesis (e.g. VHDL). The approach we undertook is to use as a basis the **C** language, adding minor extensions in order to tailor it for hardware description. The motivations for choosing C over the other possibilities are *simplicity* and *familiarity*. We wish to provide the basic mechanisms to describe general hardware functionality, without being burdened by timing and resource constraints. We feel design constraints are inherently different from the functional description and hence should be separately specified. Moreover, another advantage for choosing C is the linkage to an existing functional simulator, THOR, that is based on C models [2].

3.1 Extending C for Hardware

Efficient hardware description requires several concepts (and corresponding language constructs) that the C language lacks. Most notable are efficient synchronization and communication between different concurrent programs and the modeling of concurrent programs.

The extensions, both in concepts and in constructs, to the C language are incorporated into a derivative of the C language called *HardwareC*. They are described below.

1. **Processes** - HardwareC provides for the definition of hardware *processes*. A system can be modeled as a set of concurrent processes, where the modes of interaction are through the particular style of interprocess communication (IPC). This paradigm is appropriate for hardware since hardware modules are resources that are allocated, and which continuously operate on a time varying set of inputs. Upon completion they automatically restart in order to operate on the next set of inputs.

   The concept of concurrent processes is very powerful both in software and in hardware. In both domains, it allows the designer (1) the ability to specify parallelism between interacting modules at a high-level, and (2) the ability to isolate communication points between one process and another. The latter is accomplished through the use of IPC.

   Automatic synthesis is restricted at present to being within process boundaries. This is acceptable since the designer in most cases has a reason for partitioning the problem into processes in the first place. It does not, however, preclude the possibility of process-level partitioning as in an architectural exploration tool.

   Processes may be used by the designer to architect *pipelining* or other design styles at the behavioral level, which greatly enhances the power of high-level hardware description.

2. **Interprocess Communication** - It is instructive to note, and hopefully learn, from the analogies of design in the software domain. In software systems, there are two paradigms for interprocess communication: *shared memory* and *message passing*. Each approach has its advantages and limitations. For example, in communication through shared memory, the performance advantage is offset by an increase in the complexity of the resulting program. Likewise, the conceptual elegance of message passing solves both synchronization and communication in systems, but may result in unacceptable performance penalties if it is used without restraint [6].

   For hardware design, we would like the flexibility of using the approach that is best fitted for a particular problem. For large volume data transfers, the preferred method is to use shared memory, using message passing purely for synchronization and limited transfer of information [6].

   HardwareC offers both approaches. First, it allows for shared memory communication through the use of *parameters* to routines. Second, it allows a synchronous *send-receive* message passing scheme with fixed-size messages. The size of a message represents the number of bits that is communicated between the processes, and may be specified by the designer in the description. Synchronous message passing provides an simple yet powerful approach to inter-process synchronization and limited information transfer without incurring the cost of message buffering.

   An example of the use of IPC in hardware design is the Intel 8251 UART description, shown in Figure 2. The figure is an outline of the interprocess interaction between four independent processes: *Main*, *SyncRcvr*, *AsyncRcvr*, and *Xmit*. *Main* accepts commands from the microprocessor interface and sends or receives appropriate information to and from the receiver and transmitter processes. *SyncRcvr* (*AsyncRcvr*) receives the synchronous (asynchronous) serial data and sends it to the *Main* process. *Xmit* receives the data to be written from *Main* and serially outputs the data. Synchronization is achieved since both processes must be ready simultaneously for a message transaction to occur.

3. **Memory** - HardwareC allows for the definition of memory modules. Memory is a logical entity that is, at this stage of description, independent of the style of implementation. It can be an architected register, a register file, or a general purpose memory. Each entry in the memory module is of fixed size, and a particular entry may be accessed via an index, not necessarily known at compile time. An example for a RAM is

   ```
   declare memory RAM[AddrSize][DataSize];
   ```

   RAM is a memory module with AddrSize address bits and DataSize data bits. Access to memory is not arbitrated by the system. However, Hercules will guarantee that within one process, memory accesses will be made exclusive in time. Therefore, if the memory is used by only one process no arbitration is needed.

4. **Parameter Classes** - Parameters to routines are categorized into three types: *in*, *out*, and *inout*, depending on whether they are only referenced, only modified, or either referenced or modified. Inout parameters are bidirectional wires. The access protocol to this bidirectional line is left to the task of the designer, not the system.

5. **Architected Registers** - At times the designer would like to explicitly associate a given local variable with a register. This situation may arise for instance if the variable is an internal status register or some register that has particular significance for testing or simulation. HardwareC provides the capability to declare *architected registers* within a procedure. An architected register is declared as

```
register ProgramStatus[SIZE];
```

and will be implemented as a register in data-path synthesis. Since it implies structure, architected registers are not strictly behavioral constructs. However, they are provided to accommodate designers that may describe at a lower (structural) level.

6. **Operators** - A variety of combinational operations are provided by HardwareC. These include Boolean operations (and, or, not, xor), comparisons (equal, not equal, >, <, etc.), and arithmetic operations (+, -). Only Boolean operations will be synthesized in the data-path using logic equations. The non-Boolean operators are linked in from a library of functional units. The use of library units simplifies hardware description, and increases the flexibility of *resource scheduling and allocation*. For example, if a particular unit does not satisfy the design constraints, then another unit with the same functionality but with different area/timing specifications may be used instead.

7. **Variable Types** - HardwareC allows two types of variables. A *Boolean* typed variable is either a scalar, or a one-dimensional bit vector of a given size. An *integer* typed variable is a meta-variable; and is never synthesized in hardware. Meta-variables are used by the designer to simplify the description, and are resolved at compile time.

4 Behavioral Synthesis

The term *behavioral synthesis* applies to an abstraction and the optimizations that are carried out on this abstraction that are independent of its structure. For example, operator folding would not be considered as part of behavioral synthesis since it affects the allocation of resources in the resulting implementation.

The abstraction we have chosen is a recursively defined *parse tree*. Each leaf node in the parse tree is a *simple* statement, such as memory access, assignment, message passing, or binary and unary operations. Each internal node in the parse tree represents a *structured programming construct*, such as blocks, `while` statements, `switch` statements, etc. There are several features of the parse tree that not only simplify optimizations applied at the behavioral level, but also simplify the mapping to a structural intermediate representation. The advantages are given below.

1. *Hierarchical Description* - The parse tree allows hierarchical blocks to be described. This has an impact on later structural mapping.

2. *Local Scoping of Variables* - By grouping together variables with their scope of definition, the parse tree simplifies several behavioral optimizations that would otherwise be extremely difficult, such as loop unrolling and in-line expansion of procedures where duplication of part or all of the parse tree is necessary.

3. *Elegant Algorithms* - Because of its recursive structure, the parse tree allows elegant algorithms to be written.

4. *Multi-Pass Behavioral Synthesis* - The parse tree as an intermediate representation permits a multiple pass system. For example, the reference stack algorithm depends on knowing a priori the variables that are modified in the body of a loop. However, in a one pass system, it is impossible to obtain this information without manual specification. By having a multiple pass system, we can obtain the modified variable information in one pass, then invoke the reference stack algorithm on another pass. Each behavioral optimization constitutes as a pass through the parse tree.

4.1 Behavioral Optimizations

The set of behavioral transformations that are performed is described below.

- *Loop Unrolling*. `For` loops with fixed bounds are unrolled. This will avoid introducing registers to hold results of loop computation, at the expense of extra hardware. However, the hardware may be folded by structural synthesis.

- *Procedure In-Line Expansion*. Replaces a call to a routine with the contents of the routine. This is carried out selectively, and allows the designer to *flatten* the call hierarchy.

- *Meta-Variable Evaluation*. Meta variables are replaced by corresponding value.

- *Flatten Block Hierarchy*. Removes unnecessary block nestings in the parse tree.

- *Reduce Constant Conditionals*. Constant conditions are either swept away, or replaced by the branch of the conditional corresponding to the constant value. For example, `if (1) stmt1; else stmt2;` will reduce to `stmt1`.

- *Variable Unfolding*. Using the Reference Stack.

- *Constant Unfolding*. Using the Reference Stack.

- *Dead Code Elimination*. Dead code means operations that will not affect the output of the corresponding routine (i.e. parameters of the routine). There are two types of dead code. First, dead code can result from the *program description*. This occurs if code is written where it can not be accessed, such as after a return statement. Second, dead code can result from *unused variable references*. For example, if operations are performed with the results assigned to internal variables, then they can be effectively removed since the program behavior at the program boundary has not been affected. This type of elimination can be recursively backtracked to remove all unnecessary operations.

4.2 Reference Stack

The objective of the reference stack is to efficiently resolve *variable unfolding*, *constant unfolding*, and *resolution of conditional assignments* and *multiple assignments* in an one-pass transformation on the parse tree representation. The information generated by the reference stack to reduce the control steps that are traditionally needed to implement assignments to local variables. While the reference stack algorithm is fully detailed in [9], we present here the basic principles of the algorithm.

When a program references a particular variable at a different locations in the code, it may reference different *values* of that variable, depending on whether assignments were made between the references. The *value* of a variable in a program is defined to be the data most recently assigned to that variable. The reference stack keeps a stack for every variable in the program, where the top of stack represents the most recent value at that particular stage of translation. For example,

$a = b + c;$		$T1 = b + c;$
		$a = T1;$
$d = a;$	$\Longrightarrow$	$d = T1;$
$a = a + 1;$		$T2 = T1 + 1;$
		$a = T2;$
$e = a;$		$e = T2;$

$T1$ and $T2$ are temporary variables used to hold the results of a given operation, and are automatically generated by the parser. In the example, it is clear that whenever an assignment is made to a given variable, subsequent references to that variable results in referencing the *value*, and not the *variable*. The same simplification can be done

for constant unfolding, where the value of a variable may be a constant value instead of a variable.

The problem is complicated by the fact that we may have conditional assignments to a variable. A canonical example is

$a = 1;$
if (*condition*)
    $a = b;$
$x = a;$

What is x? Ideally, x should be the output of a multiplexer of two values based on the result of *condition*. Therefore, a Boolean equation describing the value of x is $x = (condition \wedge b) \vee (\overline{condition} \wedge 1)$. This is exactly what the reference stack does.

The reference stack is used in the one-pass transformation as follows.

- Initially, each variable in the routine will initialize its corresponding reference stack (one for each bit) with a certain value. For input and inout parameters, the value is itself; otherwise, the value is 0 or some error value.

- If an assignment is made to a given variable, the top of the reference stack will be modified to the assigned value, so that subsequent references may access the most current *value* of the variable.

- When a conditional branch is encountered, a new entry is pushed onto the reference stacks of all the variables for each different case of the conditional. The new entries are initialized to be the previous top of stack. Assignments in a given branch of the conditional will affect the corresponding top of reference stack only. References to variables will access the most recent value assigned to it within the same branch.

  Upon completion of the conditional, we will create a *virtual* multiplexer variable whenever a variable has been modified within a branch of the conditional. The mux is virtual since if it is not subsequently referenced, it will be removed.

- When `while` loop is encountered, a new entry is pushed onto the reference stack for both the loop exit (0) and loop enter (1) conditions. For the loop enter condition, initialize the top of stacks for all variables that are modified in the body of the loop to themselves; otherwise, initialize to the previous top of stack.

  Upon exit from the loop, for each variable that is modified, record the *initial* value before entering the loop, and the *final* value after exiting from the loop. This information is used by structural synthesis [9].

5 Structural Synthesis

Structural synthesis is the transformation of the behavioral representation, described by one or more parse trees, into a structure consisting of an interconnection of modules. Since the modules are arbitrary combinational or sequential circuits, we have chosen to merge together both the data path and the corresponding control into a single module. Combined synthesis has been used for example by [5]. At present, the target of synthesis is a *synchronous* digital system, although the model is applicable to any implementation.

Our abstraction for structural synthesis is a sequencing graph model called the sequencing intermediate form, or SIF for short. It supports *multiple threads of execution flow* through the graph, which deviates from the existing approaches where only a single thread of control is possible through the hardware [17].

The vertices of the graph are the operations to be performed, and the edges represent the predecessor/successor relations between the vertices, subject to data dependency restrictions that may exist between them. There are several advantages to using the sequencing graph as the abstraction for structural synthesis. They include the following.

1. *Multiple Thread of Execution Flow* allows the synthesis system to explore the design space corresponding to sequential verses parallel implementations.

2. *Varying Degree of Parallelism* is achieved simply by modifying the predecessor-successor relations between the vertices. It is crucial for structural synthesis, since to fully explore the tradeoffs between area and performance the synthesis system needs to examine the effects of varying the degree of parallelism on the resulting implementation.

3. *Abstraction for Synthesis.* The concept of sequencing graph abstracts nicely the interdependencies between the operations, and provides an elegant foundation for later control/timing optimizations.

4. *Extension to Accommodate Timing Constraints.* Sequencing graph model may be extended to take into consideration timing constraints between the operations (vertices). This may be represented by associating the timing constraint to each edge. This issue is currently under investigation.

Structural synthesis consists of two tasks. The first task is ***data path synthesis***, which directly maps the behavioral intermediate form onto a structural interconnection. The second task is ***control synthesis***, which determines how and when data traverses through the data path. The output of structural synthesis is a set of logic equations with delay information that describe the control and portions of the data path (excluding library elements, such as adders and multipliers). This is stored in the SLIF format.

5.1 Data path Synthesis

Data path synthesis maps operations to a set of logic equations. In particular, it performs the following tasks.

- ***Boolean operations*** - Boolean operations such as AND, OR, XOR, and NOT are described by the appropriate logic equations.

- ***Register allocation*** - a variable may either be implemented as a wire or a register. Under our structural model, a local variable is mapped to register *only* if it is either explicitly declared to be an architected register, or it is "used before set" in the body of a data dependent loop. The "used before set" semantic implies storage of computation for the next iteration of the loop.

- ***Interface to procedure call*** - procedure calls may have arguments that return a value (out parameter), or it may require arguments that both input and return a value (inout parameter). The data path needs to provide for the interfacing and information transfer between the caller and the called routines.

- ***Links to library modules*** - library modules implement the combinational operators such as addition and subtraction, and must be instantiated and linked with the logic equations that describe the data path.

- ***Connection to Parameters*** - accesses to inout parameters (bidirectional) are controlled by tristate elements.

5.2 Control Synthesis

Control Synthesis consists of two tasks. The first is ***sequencing control*** which is responsible for preserving the sequencing behavior from the SIF. At this level, several simplifying assumptions are made. Specifically, combinational logic blocks are assumed to have zero execution delay, and we ignore external protocol constraints. The goal of sequencing control is to capture the behavior of the structure in a minimal number of states and transitions between the states. The approach used by Hercules is a ***ripple through*** control model.

The second task is the ***constraint control*** which deals with the constraints imposed by real hardware systems. This is done after logic

synthesis since it relies on the delay and area information. Specifically, combinational logic blocks are evaluated for delay and area estimates [4]. This information is fed back to guide subsequent design optimizations during structural synthesis. Along with these estimates, external protocol specifications are also imposed on the structure. The goal of constraint control synthesis is to find an optimal *cycle time* based on both design and timing constraints.

5.3 Control Model

We model hardware control by a control graph embedded into SIF. The vertices of the control graph are categorized into two types: *state* and *stateless* vertices. State vertices are those operations that require at least one cycle for execution in our hardware model, and include assignment to parameters, memory access, message passing primitives, calls to non-combinational routines, and while loop condition evaluation. Stateless vertices are those that do not necessarily require one cycle for execution, and include switch condition evaluation, block start and block end, joins from conditionals, and no-op vertices.

The flow of execution through the graph can viewed as an execution *wavefront* that ripples through the graph. Each vertex upon completion will activate its successors. For stateless vertices, no time is needed for execution in the sequencing control model, and control is immediately branched off to the successors.

5.4 Implementation of Control

The control is implemented as an interconnection of Moore type finite state machines, one for *each* state vertex of the control graph. Each of the finite state machines may be designed either in terms of *level-sensitive* or *edge-triggered* registers.

A finite state machine interacts with other finite state machines via three signals - *enable*, *done*, and *restart*. The *enable* input indicates the start of execution, and is a conjunction of all the *done* signals from its predecessors. The *done* output indicates the completion of execution, and the *restart* input resets the entire control graph. For the case of a procedure call, the *restart* signal is equivalent to the *enable* signal issued by the calling vertex. For the case of a process, it is issued as an external signal.

The finite state machine consists of three internal states: RESET, ACTIVE, and CONTINUE. The operation corresponding to the vertex is executing only in the ACTIVE state. Upon completion of execution, there is a transition from the ACTIVE to the CONTINUE state in which the *done* signal is asserted. A transition out of the CONTINUE state to the RESET state takes place when the *restart* signal is asserted, which resets all finite state machines implementing the control graph. For stateless vertices, the *enable* signal is defined similar to state vertices, with the *done* signal equivalent to the corresponding *enable*. For further details on the implementation, we refer the interested readers to [9].

6 Implementation and Results

Hercules is a system for high-level synthesis of digital hardware. It takes as input a *behavioral description* in the form of one or more HardwareC programs and generates an *logic implementation* in the form of one or more SLIF files. There are over 22,000 lines of C code in the implementation.

Hercules has been used to synthesize several benchmarks examples, including two proposed for the ACM High Level Synthesis Workshop held at Rosario in January, 1988: the MC6502 microprocessor, Intel 8251 UART chip, and FRISC, a simple example described in [14]. The results are tabulated for each benchmark in Figure 3. The values given for the number of registers, temporary variables, and multiplexers are in 1-bit quantities. That is, a 16-bit register will give a count of 16 registers. The temporary variables serve as *nets* connecting the functional modules together. The table shows the total gate count in the CMOS3 library, excluding components such as registers, tristates, and adder/subtractor. Running times are in the range of 1 minute for the FRISC example to 5 minutes for the MC6502 example on a VAX 11/780.

7 Conclusion and Future Work

We have presented a system for high-level synthesis of digital systems. The high-level synthesis is broken down into behavioral and structural synthesis. While the latter is largely responsible for exploring the cost-speed tradeoffs in a design, the first is important in extracting the behavioral characteristics from the high-level descriptions. A new method called the reference stack is introduced that performs many of the behavioral level optimizations in a one pass transformation, and aids later mapping of variables to registers. We also propose a control model based on sequencing graphs that supports multiple threads of control.

Future work includes developing iterative techniques to improve the structure being generated, extracting area and delay information from logic synthesis and applying them to structural synthesis. Control synthesis that takes into consideration the user-defined constraints is also necessary in achieving reasonable results. Furthermore, work is currently underway to link Hercules to the THOR simulator, which can provide feedback on the correctness of the description.

8 Acknowledgment

The Hercules project is supported by NSF Contract MIP-8710748 and by the Stanford Center for Integrated Systems seed fund No. 172C062.

References

[1] J. Bhasker. *An Optimizer for Hardware Synthesis*, Scientific Honeyweller, vol 7, no 3, 23-34.

[2] L. Soule and Tom Blank. *Statistics for Parallelism and Abstraction level in Digital Simulation*, Proceedings of DAC Conference, Miami Beach, July 1987, p 588-591.

[3] Gaetano Boriello and Randy H. Katz. *Synthesis and Optimization of Interface Transducer Logic* Proceedings of ICCAD Conference, Santa Clara, 1987.

[4] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang. *MIS: A Multiple-Level Logic Optimization System*, IEEE Transactions on CAD, vol CAD-6, No. 6, November 87, pp. 1062-1081.

[5] R. Camposano. *Structural Synthesis in the Yorktown Silicon Compiler*, Proceedings VLSI 87 Conference, Vancouver, August 87.

[6] D. R. Cheriton and W. Zwaenepoel. *Distributed process groups in the V kernel*, ACM Transactions on Computer Systems, 3(2), May 1985.

[7] Forrest Brewer and Dan Gajski. *Knowledge based Control in Micro-Architecture Designs* , Proceedings of Design Automation Conference, Las Vegas, 1987, pp 203-209.

[8] Louis J. Hafer and Alice C. Parker. *Automated Synthesis of Digital Hardware*, IEEE Transactions on Computers, vol c-31, no 2, Feb 1982.

[9] David C. Ku and Giovanni De Micheli. *HERCULES - Algorithms for High Level Synthesis*, Stanford CIS Technical Report, 1988.

[10] M. C. McFarland. *The Value Trace: A Data Base for Automated Digital Design*, Report DRC-01-4-80, December 1978.

[11] J. A. Nestor and D. E. Thomas. *Behavioral Synthesis with Interfaces*, ICCAD '86, Santa Clara, November 1986, 112-115.

[12] W. Rosenstiel, R. Camposano. *Synthesizing Circuits from Behavioral Level Specifications*, 7th International Symposium on Computer Hardware Description Languages and their Applications, Tokio, August 1985.

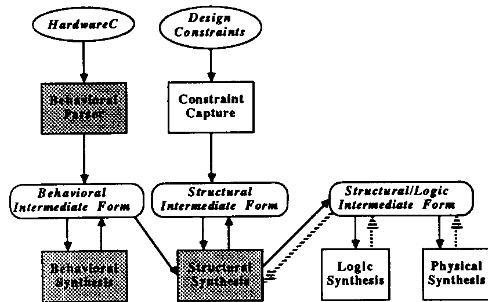

Figure 1: OLYMPUS Design System block diagram - Hercules consists of the shaded boxes, dashed arrows represent work in progress

[13] Chia-Jeng Tseng and Daniel P. Siewiorek. *Automated Synthesis of Data Paths in Digital Systems*, IEEE Transaction on Computer-Aided Design, vol CAD-5, no 3, July 1986.

[14] J. R. Southard. *Macpitts: An approach to silicon compilation*, Computer, vol 16, pp 74-82, December 1983.

[15] D. E. Thomas. *Automatic Data path Synthesis* Advances in CAD for VLSI in *Design Methodologies*, vol. 6, S. Goto (ed.), North Holland, 1986, pages 401-439.

[16] H. Trickey. *Flamel: A High-Level Hardware Compiler*, IEEE Transactions on CAD, CAD-6, No. 2 (March 1987), 259-269.

[17] R. Brayton, R. Camposano, G. DeMicheli, R. Otten and J. van Eijndhoven. *The Yorktown Silicon Compiler System*, in Silicon Compilation, D. Gajsky (ed.), Addison Wesley, 1988.

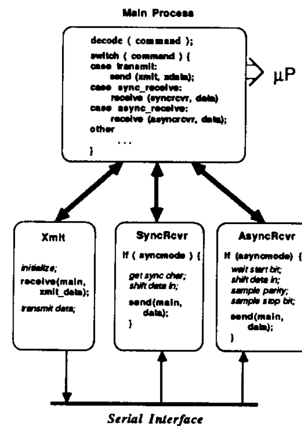

Figure 2: Outline of I8251 process interaction

FRISC	Lines of Code	Times Called	Instances Needed	# bit wires	# bit register	# gates unoptmz	# gates optimz	# gates CMOS3
risc	86	1	1	631	112	497	185	220
read	9	11	2	17	16	73	21	23
write	7	5	1	17	16	76	23	24

FRISC	Lines of Code	Times Called	Instances Needed	# bit wires	# bit register	# gates unoptmz	# gates optimz	# gates CMOS3
main	68	1	1	13	8	210	105	132
xmit	100	1	1	38	11	200	92	107
async	59	1	1	19	11	130	80	87
sync	42	1	1	14	11	86	50	57
hunt_mode	35	1	1	32	4	84	52	66

FRISC	Lines of Code	Times Called	Instances Needed	# bit wires	# bit register	# gates unoptmz	# gates optimz	# gates CMOS3
mc6502	80	1	1	143	46	228	115	140
group0	195	1	1	591	33	625	245	283
group1	80	1	1	224	32	221	120	128
group2	120	1	1	342	33	385	185	205
abs	12	11	2	83	0	77	35	40
immed	6	5	2	17	0	57	23	31
indx	14	1	1	107	0	111	70	82
indy	13	1	1	107	0	111	70	82
pull	12	7	3	11	0	103	60	72
push	11	10	3	12	0	102	60	73
read	8	31	6	6	8	62	33	42
write	7	8	2	3	0	36	27	33
setnz	6	21	2	1	0	17	9	10
zp	8	15	2	58	0	75	32	39

Figure 3: Statistics for Benchmarks - Wires and Registers given as one-bit quantities