# A Source-level Dynamic Analysis Methodology and Tool for High-level Synthesis

Chih-Tung Chen and Kayhan Küçükçakar

Unified Design System Laboratory
Motorola, Inc.
{chen,kayhan}@adtaz.sps.mot.com

## Abstract

*This paper presents a novel source-level dynamic analysis methodology and tool for High-Level Synthesis (HLS). It not only for the first time enables HLS to offer source-level design debugging on the 'synthesized' RTL designs, but also allows the designer to analyze their dynamic characteristics, such as resource utilization, power consumption, etc., at the algorithmic (source) level. This technology has been proven in the industry as the critical element for successfully designing a microcontroller with 300+ instructions with Matisse, an interactive HLS system. Additionally, we demonstrate the use of this technology for architectural power optimization.*

## 1. Introduction

After close to two decades of research, commercial High-level Synthesis (HLS) tools have been introduced recently and have been in some production use. A tutorial on HLS methodology and past research can be found in [7] and [10]. As discussed in [8], the question is not whether high-level synthesis will be successful or not but how widely it will be used in the industry. As pointed out by Bergamaschi in [1], the discussion in [8] and our own study in Motorola, there are some common problems with traditional HLS systems which still hinder designers from embracing HLS. Two specific problems among them are the controllability/predictability and simulation/analysis of the synthesized RTL implementations. First of all, designers demand the controls of the HLS process in order to generate the desired and predictable results. The predictability is found to be crucial for subsequent design tasks including RTL/logic synthesis, data-path module generation, floorplanning, layout, test, and verification. Secondly, designers need an 'intuitive' way to analyze and verify the output of HLS. For example, the cycle-by-cycle behavior of the synthesized design is often different from the one exhibited by the simulation of the input description. Furthermore, the designer needs to understand the internal details, not just the I/O behavior, of the synthesized design so that he/she can change and annotate the subsequent HLS iterations in order to get the desired results.

To accelerate the industrial adoption of HLS, we have developed Matisse, a versatile architectural design tool, which answer many challenges raised in the above studies and discussions [6]. It is beyond the scope of this paper to describe the differences of Matisse from traditional HLS.

This paper presents a novel source-level dynamic execution analysis methodology incorporated in Matisse which enables the designers to analyze the dynamic characteristics of HLS-generated designs in the most intuitive manner: *the original behavioral description*. A portion of this methodology is analogous to the source-level debugging of a compiled software code, where the programmer debugs the execution of the compiled machine code while viewing the original source code. In this analogy, HLS, the input description, and the simulation of the synthesized design correspond to the software compiler, the software source code, and the execution of the compiled machine code, respectively. Additionally, our methodology allows the designers to analyze the design's dynamic characteristics from the source-level execution point of view such as cycle-based resource utilization and power consumption. Thus, the designer can correlate the synthesis results with the HLS input description and constraints, and then, if necessary, adjusts the subsequent HLS iteration to optimize the design's dynamic characteristics more effectively.

Note that our source-level analysis methodology is different from the typical HDL (Hardware Description Language) debugging tools which are built on top of the HDL simulators. These tools allow the designers to debug their HDL code being simulated by the simulator. Applying these tools in the HLS context, the designer is limited to analyze the simulation of the input behavioral description and the RTL implementation **separately** and **independently**. On the contrary, our methodology bridges over this gap by analyzing the RTL simulation from the cycle-accurate execution point of view of the input behavior descrip-

tion.

To our best knowledge, the only related work in the hardware design technology is the finite-state machine or flowchart design/analysis tools, such as *VisualHDL* from Summit Design Inc., *DesignBook* from Escalade, *Express* from i-Logix and *speedCHART* from Speed. This type of tools offer visual feedback and controls through state-transition diagrams or flow charts while simulating the HDL code generated for RTL/logic synthesis. The technique for the above FSM/flowchart analysis relies on the fact that there is a clear correlation between the specification and the generated implementation (HDL code); e.g., state registers. However, HLS doesn't offer such straightforward correlation for source-level analysis of HLS designs.

The remainder of this paper is organized as follows. Section 2 gives an overview of what it takes to implement a source-level execution analysis methodology for HLS. In Section 3, we describe the concept of design annotation during HLS and the techniques for enabling the source-level analysis. Section 4 presents the source-level analysis environment in Matisse and illustrates various analysis capabilities offered by this environment. In Section 5, we discuss the applications of our methodology including a microcontroller design and an architectural power optimization experiment. Finally, we conclude this paper in Section 6.

## 2. Source-level execution analysis methodology

As discussed earlier, the focus of this work is to develop a source-level execution analysis methodology for HLS. The methodology should not only serve as the symbolic debugger for the RTL designs generated by HLS, but also support the analysis of the dynamic characteristics such as cycle-based resource utilization and power consumption. There were several questions raised at the early stage of this work. Is the methodology feasible for HLS? Also, can we leverage existing techniques used for software debugging and FSM/flowchart analysis?

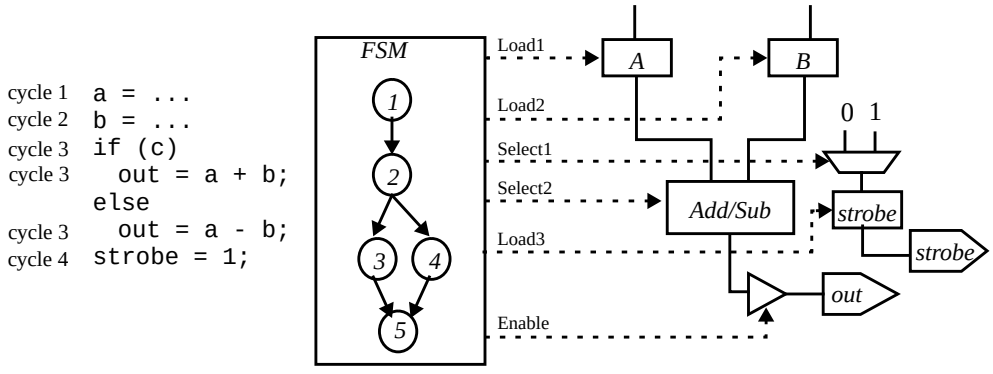
### 2.1. Feasibility

To understand why it is feasible for HLS to offer the source-level execution analysis, recall that HLS is a process of mapping an algorithmic description of a design behavior into a Register-Transfer Level (RTL) implementation. The HLS tasks include scheduling, resource allocation and sharing, interconnect creation, and finally controller (FSM) generation. Figure 1 shows an example of the mapping performed by HLS. It can be seen that there exists a mapping from the input description to the states or state transitions of the controller (scheduling) and to RTL components and interconnect (binding). Such mapping information can be used to correlate the RTL simulation results back to the input description (the source). Briefly speaking, if we know the current state of the controller (FSM) or the state transition being taken in a particular cycle during RTL simulation, the reverse mapping of the scheduling and binding should tell us at this cycle which portion of the input description is executed and which components and interconnect are utilized. Thus, cycle-based source-level execution and resource utilization become feasible.

### 2.2. Mathematical model

In this section, we first describe the essence of the HLS design execution, meaning the simulation of the RTL implementation. Then, we define the reverse mapping of the scheduling and derive the source-level execution from the above information. Finally, some extensions of the model will be addressed.

For the sake of simpler explanation, assume the HLS design contains a single FSM as the controller. If the generated controller is *state-based* (e.g., a Moore-type FSM)




*a. Specification* *b. RTL Implementation*

**Figure 1: High-level synthesis process**

with states $S = \{s_1, ..., s_A\}$, the execution of the HLS design, therefore, can be characterized by an ordered list of states $DX = \{dx_1, ..., dx_B\}$ such that $dx_b \in S$ and $(dx_b, dx_{b+1})$ is a valid state transition. Note that $dx_b$ represents the current state of the controller at the $b$th cycle of the HLS design execution.

Let the input description be a list of statements $I = \{i_1, ..., i_C\}$. The scheduling task of HLS can be represented by a one-to-many mapping $\lambda: I \rightarrow S^S$ such that $\lambda(i_x)$ is a subset of $S$ and denotes the states in which statement $i_x$ is to be executed. Thus, the reverse mapping of $\lambda$ becomes $\lambda^{-1}: S \rightarrow I^I$ where $\lambda^{-1}(s_y) = \{i_x \mid s_y \in \lambda(i_x)\}$.

Formally, the cycle-based source-level execution can be represented by an ordered list $SX = \{sx_1, ..., sx_D\}$ such that $sx_d$ is a subset of statements $S$ which executed at the $d$th cycle of the HLS design execution. It can be seen easily now that the source-level execution $SX$ can be derived from $DX$ and $\lambda^{-1}$ as $\{\lambda^{-1}(dx_1), ..., \lambda^{-1}(dx_D)\}$ since $sx_d$ is equal to $\lambda^{-1}(dx_d)$.

On the contrary, if the generated controller is state-transition-based (e.g., a Mealy-type FSM) with state transitions $T = \{t_1, ..., t_E\}$, the execution of the HLS design must be characterized by an ordered list of state transitions $DT = \{dt_1, ..., dt_F\}$ such that $dt_f \in T$ and $(dt_f, dt_{f+1})$ is connected through a valid state. The HLS scheduling, in this situation, is a one-to-many mapping $\lambda: I \rightarrow T^T$ such that $\lambda(i_x)$ is a subset of $T$ in which statement $i_x$ is to be executed. Additionally, the reverse mapping $\lambda^{-1}$ becomes $\lambda^{-1}: T \rightarrow I^I$ where $\lambda^{-1}(t_y) = \{i_x \mid t_y \in \lambda(i_x)\}$. Similarly, the source-level execution $SX$ can be derived from $DT$ and $\lambda^{-1}$ as $\{\lambda^{-1}(dt_1), ..., \lambda^{-1}(dt_D)\}$ since $sx_d$ is equal to $\lambda^{-1}(dt_d)$.

The model above can be easily extended to support the cycle-based resource utilization. Briefly speaking, the cycle-based resource utilization is a dynamic analysis of data-path resource utilization throughout the run (simulation) time of the HLS design. The data-path resource can be a data-path component or an interconnect. The advantage of analyzing cycle-based resource utilization is that it can accurately portray the actual resource utilization of designs with complex conditional resource sharing and data-dependent loops.

Like source-level execution, the cycle-based resource utilization can be derived from the HLS design execution and the reverse mapping of the HLS binding. For example, assume the HLS design is state-based as described earlier and comprises of a set of resources $R = \{r_1, ..., r_G\}$. The HLS binding can be transformed to a one-to-many mapping $\gamma: R \rightarrow S^S$ such that $\gamma(r_x)$ is a subset of $S$ in which resource $r_x$ is to be used, and its reverse mapping becomes $\gamma^{-1}: S \rightarrow R^R$ where $\gamma^{-1}(s_y) = \{r_x \mid s_y \in \gamma(r_x)\}$. Finally, let the cycle-based resource utilization be an ordered list $U = \{u_1, ..., u_H\}$ such that $u_h$ is a subset of $R$ which are utilized at the $h$th cycle of the HLS design execution. Thus, $U$ can be derived as $\{\gamma^{-1}(dx_1), ..., \gamma^{-1}(dx_H)\}$ since $u_h$ is equal to $\gamma^{-1}(dx_h)$.

Although the model described above assumes a single-FSM controller, it can also be extended to support those HLS systems which generate multi-FSM controllers such as Hercules/Hebe [5] and Matisse [6]. For example, assume the FSMs are all state-based and use the same clock. Let the FSMs be $M = \{m_1, ..., m_I\}$. The HLS design execution can be characterized by using multiple $DX$ lists, one for each FSM as described earlier. The source-level execution $SX$ can then be derived from $\lambda^{-1}$ and the $DX$ lists with $sx_d$ as $\lambda^{-1}(dx_d^1) \cup ... \cup \lambda^{-1}(dx_d^I)$.

Another extension of our model is to use operations and values instead of statements for the source-level analysis. This extension would allow the user to better analyze intra-statement events. This can be accomplished simply by replacing the statement list $I$ with a list containing individual operations and values.

### 2.3. Analysis flow

As described in Section 2.2, our source-level execution analysis methodology is enabled by two key ingredients: the reverse mapping and the state (or state-transition) history of the underlying controller which captures the essence of the HLS design execution. In this section, we will discuss what it takes to incorporate such a methodology into an HLS system and the associated analysis flow.

Theoretically, an HLS design can be simulated "as is" and its state-based execution history still can be obtained through the simulator by monitoring the state registers of the controller. However, the transition-based execution history cannot be captured by such monitoring. For example, assuming there are two states $s_1$ and $s_2$ with two conditional transitions from $s_1$ to $s_2$ depending on the value of an input *cond*. Knowing that the HLS design is in state $s_2$ is not sufficient to tell which transition the design took to reach $s_2$. The problem is further complicated by the potential logic glitches, such as those on *cond*, which may cause specious state transitions in the middle of a cycle.

Therefore, we introduce the concept of *design annotation* and *analyzable HLS designs*. The design annotation is a process of embedding one or more daemons into the design during HLS. The embedded daemons when simulated will use specific techniques to collect dynamic data of interest such as state transitions. Moreover, the embedded daemons will not affect the functional behavior of the design during simulation. The HLS design with the embedded daemons is called *source-level analyzable*. The design annotation will be further discussed in Section 3.

The second crucial task in our methodology is to generate the reverse mapping of scheduling and binding. This involves reprocessing the scheduling and binding decisions and storing the information tabularly with states or

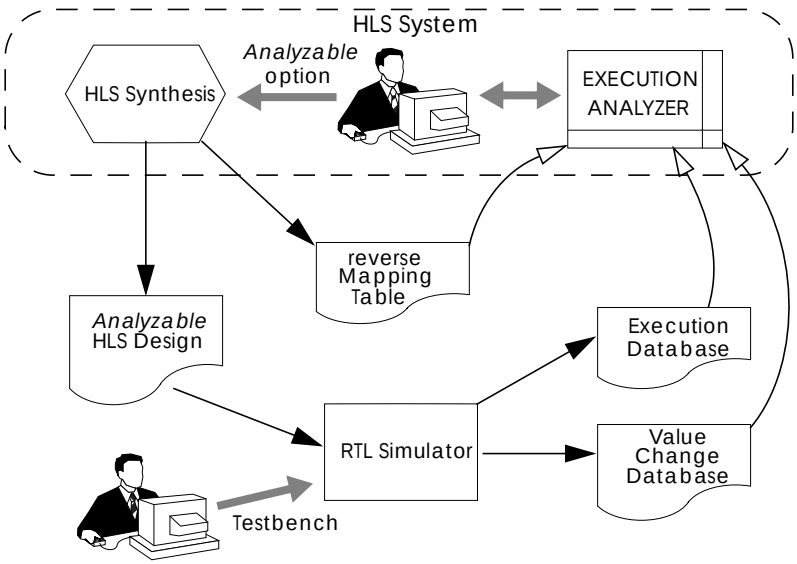



**Figure 2: Source-level execution analysis flow**

state transitions as the indices. This task, however, is best carried out during the HLS controller generation where the complete scheduling and binding information is available and the states as well as the state-transitions are being generated.

As shown in Figure 2, a typical source-level execution analysis flow begins with synthesizing an *analyzable* HLS design containing embedded daemons and generating one or more tables containing the reverse mapping of scheduling and binding.

Following that, the analyzable HLS design is simulated, and an execution database is generated by the embedded daemons along with a value change database.

The execution database stores the execution history; e.g. a list of 2-tuple (*t*, *st*) where *t* is the simulation time and *st* is an identifier of a particular state transition. The value change database contains the value history of one or more signals in the design. This value change database enables our methodology to offer both behavioral and structural signal analysis along with the source-level execution analysis. Although the generation of the above databases is needed for post-simulation analysis and time-backward analysis, the real-time source-level analysis is also possible by using the simulator's application programming interface (API) to generate and access the records in real time.

The last step in the analysis flow is to invoke a source-level execution analysis tool similar to a software symbolic debugger for the actual analysis work. The details of such source-level execution analysis tool will be described in Section 4.

## 3. Design annotation

Design annotation is a process of embedding one or more daemons into the design during HLS for collecting dynamic data of interest using specific techniques. Currently, we identified two types of design annotation: controller annotation for capturing the state-transition based execution history and data-path annotation for calculating cycle-based power consumption.

Note that the design annotation should be performed such that the analyzable HLS design doesn't incur any overhead in the downstream synthesis tools as compared to the original HLS design. In our case, we hide the embedded daemons from logic synthesis by enclosing them with directives that turn logic synthesis off and on.

### 3.1. Controller annotation

As discussed in Section 2.3, the transition-based execution history cannot be captured by simply recording the controller's state registers. With potential logic glitches, it is also difficult to derive the actual transition history by continuously monitoring the state-transition logic of the controller. Since *Matisse*, our targeted HLS system, is state-transition based, we annotated the controller with one daemon per FSM. Each daemon comprises of a state-transition identification (*STI*) logic and an *STI* register, and is connected with a typical Mealy-type FSM as shown in Figure 3. Like other Mealy-type output functions, the *STI* logic is a function of FSM's current state *CS* and the input *I*, and computes the next state-transition identification *NSTI*. The *STI* register is analogous to the FSM state register. Functionally speaking, the *STI* register represents edge-triggered flip-flops driven by the FSM clock signal.

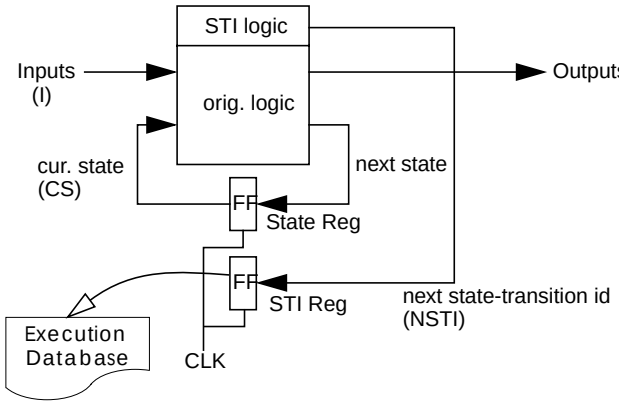



**Figure 3: STI annotation**

Also shown in Figure 3 is an arc connecting the *STI* register output to the execution database. It points out the real role of the *STI* register, which is to store the identifier of the actual state transition committed by the FSM into the execution database along with the simulation time at the end of every clock cycle. Thus, the cycle-based state-transition history can be faithfully recorded under the presence of logic glitches.

We found that it is more efficient to generate the reverse mapping tables of scheduling and binding during the controller annotation. Recall that these reverse mapping tables for the state-transition based HLS designs correlate each state transition with a set of input statements being executed or a set of resource being utilized. By using the state transition identifiers generated during the controller annotation as the key field for the reverse mapping tables, the reverse mapping for source-level execution and cycle-based utilization can be done in linear time.

### 3.2. Data-path annotation

The data-path annotation for cycle-based power analysis is a process to embed a daemon (*SD*) to an individual data-path component or net for gathering signal switching statistics periodically. As shown in Figure 4, *SD* is driven

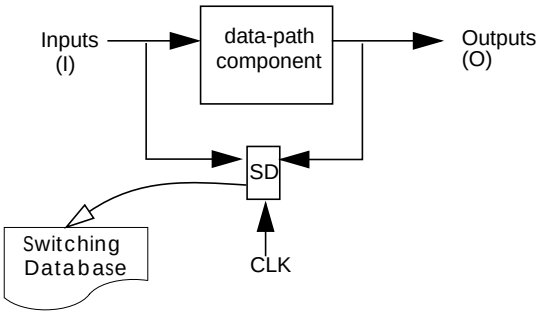



**Figure 4: Data-path annotation**

by a system or component-specific clock *CLK*. Similar to the *STI* daemon, *SD* stores the peripheral switching statistics of the component into the switching database along with the simulation time at the end of every cycle. Thus, cycle-based power consumption can be computed by the switching statistics in each cycle and the power characterization models of individual components available from RTL power analysis tools such as [11] and [9].

## 4. Source-level analysis environment

Figure 5 shows the source-level analysis environment developed in Matisse using the methodology described earlier. Currently, it offers source-level execution analysis, dynamic resource utilization analysis and dynamic profiling. We would like to remind that although our source-level analysis environment seemingly resembles a typical HDL debugging tool, the underlying techniques are fundamentally different. More importantly, traditional HDL debugging tools cannot be used for source-level analysis of synthesized designs.

### 4.1. Source-level execution analysis

As shown in Figure 5, the source-level execution analyzer (*EA*) consists of a control panel, a source window, and a data display window. The control panel allows the designer to perform typical debugging operations such as *start*, *stop at*, *next*, *continue* and *data display*. Due to the fact that the execution history is available in the execution database, *EA* also offers backward stepping (*prev*) and backward continuation to move the simulation time back to a prior clock cycle. The source window is for listing the source code of the HLS design (currently an algorithmic subset of Verilog HDL) and highlighting those statement being executed at the current cycle. It also gives the designer the access to individual statements for setting the stop points and the access to variables and operations for displaying values. Finally, the data display window contains a list of data display entries which are updated automatically whenever the simulation time changes.

The basic core of *EA* can be described in pseudo code as following.

```
1. import(execution_database, reverse_mapping_tables);
2. while (not quit) begin
3.    request_or_compute(next_cycle);
4.    query_execution_database(next_cycle, assoc_stis);
5.    exec_stmts = ∅; used_rsrcs = ∅;
6.    foreach sti in assoc_stis begin
7.          exec_stmts |= schedule_reverse_mapping(sti);
8.          used_rsrcs |= binding_reverse_mapping(sti);
9 .   end
10.   display_time(next_cycle);
11.   source_window_highlight(exec_stmts);
12.   structure_window_highlight(used_rsrcs);
13. end
```

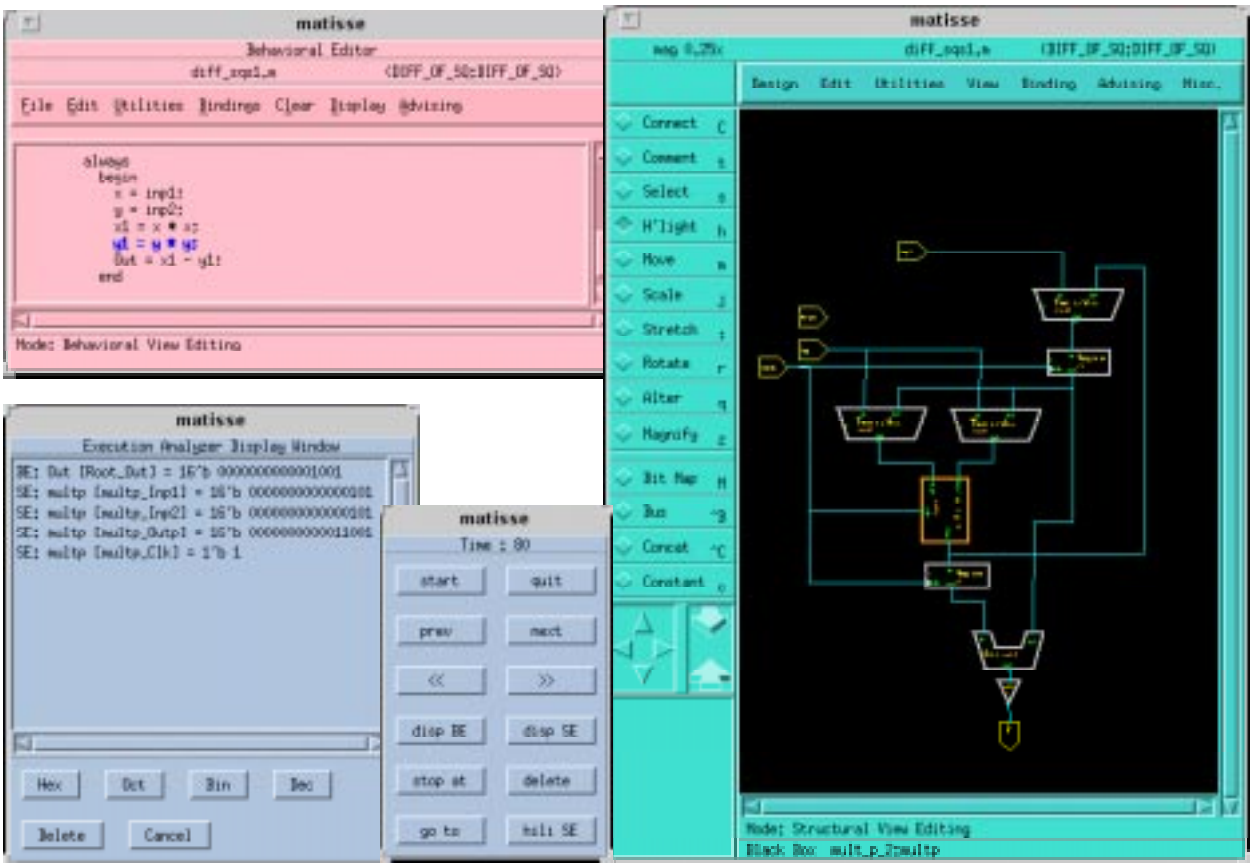



**Figure 5: Source-level analysis environment**

Note that *sti* stands for state-transition identifier and '|=' is a union-assign operation.

### 4.2. Dynamic resource utilization analysis

Also shown in Figure 5 is a structure window for cycle-based resource utilization analysis. The structure window of *EA* displays the data-path schematics and highlights the active (utilized) resources at the current cycle in a similar fashion to *SEE-SAW* [2]. The designer can also select a particular net or component through the structure window for value display.

By using the dynamic resource utilization analysis, the designer can visualize 'where' and 'when' a portion of the data path is idle under the actual operational environment. Then the schedule and/or bindings can be adjusted to improve the overall dynamic utilization, or power saving techniques such as signal gating can be applied to reduce the power consumption.

### 4.3. Cycle-based power analysis

The cycle-based power analysis can be performed by integrating the component-level power models provided by [9] with the data-path annotation technique described in Section 3.2. The intuitive functionality is to provide the designer so-called *power display* similar to the existing *data display* with the same granularity such as per net, per component or even per operation basis. The power display entries are to be shown in the display window and updated whenever the simulation time change.

### 4.4. Dynamic profiling analysis

Another related analysis available is the dynamic profiling analysis. The dynamic profiling analysis reports the execution frequency of individual statements in the input description and the usage frequency of individual resources in the data path under a scenario of the HLS design simulation. The profiling data is computed in a similar way as described in Section 2.2 for cycle-based source-level analysis using the execution database and the reverse mapping tables except that the computation is accumulated over the entire execution history.

## 5. Applications of source-level analysis

Matisse equipped with the source-level execution analysis presented in this paper has been used in several industrial designs for evaluation or production. The designs included hardwired DSP, encryption/decryption engine,

bus interface, control-dominated peripheral, and a microcontroller. The source-level execution analysis was proven to be an invaluable tool to the designers especially for the designs involving considerably interactive and/or incremental design. In this section, we will summarize two notable applications of the source-level analysis.

### 5.1. Design of an industrial microcontroller

Matisse has been used to redesign a commercial 8-bit microcontroller with 300+ instructions. During this redesign project, it became inevitable that the designers had to schedule the majority of the micro-instructions and mapping them onto a well-defined data-path architecture interactively in order to ensure the compatibility and the efficiency. Thus, they devised an incremental design flow comprising of adding new instructions, interactive micro-instruction scheduling, mapping onto the existing data path, generating the RTL implementation, simulating and debugging. Without using the source-level execution analyzer presented in this paper, the designers will be forced to verify the newly added instructions by analyzing the RTL simulation directly, which was shown to be an overwhelmingly time-consuming task. The designers estimated that our source-level execution analyzer provided 3-4X reduction in debugging time than traditional RTL analysis.

### 5.2. Architectural power optimization

We performed an experiment of a G.721 ADPCM predictor [3]. We explored low-power architecture in an iterative design flow including architectural design using Matisse, hot point analysis using an RTL power estimator [9], cycle-based resource utilization analysis and profiling as presented in this paper. By identifying the hot points and their utilization patterns, we were able to obtained 4.4X power saving in one person week through power reduction techniques like signal gating, pipelined multipliers, register files, and pipelined scheduling. The results were then verified by an accurate gate-level power estimator [4].

## 6. Conclusion

As mentioned in Section 1, HLS needs to improve the controllability/predictability and simulation/analysis of the synthesized RTL implementations in order to increase its adoption in the industry. We have presented a unique source-level analysis methodology and tool for HLS to address these problems. It not only for the first time enables HLS to offer an intuitive design debugging capability, but also allows the designer to perform dynamic analysis of resource utilization, power consumption, etc. at the source level. This technology has been shown to be the critical element for effectively applying the HLS methodology on the industrial designs.

## 7. References


[1] R.A. Bergamaschi, “Productivity Issues in High-level Design: Are Tools Solving the Real Problems?”, *In Proceedings of the Design Automation Conference*, pages 674-677, June 1995.

[2] R. Blackburn, D.E. Thomas and P. Koenig, “CORAL II: Linking Behavior and Structure in an IC Design System”, *In Proceedings of Design Automation Conference*, pages 529-535, June 1988.

[3] C.-T. Chen and K. Kucukcakar, “An Architectural Power Optimization Case Study using High-level Synthesis”, To appear in *Proceedings of International Conference on Computer Design*, October, 1997.

[4] B. George, G. Yeap, M.G. Wloka, S.C. Tyler and D. Gossain, “Power Analysis for Semi-Custom Design”, *In Proceedings of Custom Integrated Circuits Conference*, May 1994, pp. 249--252.

[5] D. Ku and G. De Micheli, *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer Academic Publishers, 1992.

[6] *Matisse User Manual*, Motorola, Inc., 1995.

[7] M.C. McFarland, A.C. Parker and R. Camposano, “The High-Level Synthesis of Digital Systems”, *In Proceedings of the IEEE*, 78(2):301-318, February 1990.

[8] Panel: Real Requirements of High-level Synthesis, *International Symposium on System Synthesis*, 1996.

[9] *Taos User Manual*, Motorola, Inc.,1996.

[10] R.A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

[11] *WattWatcher/Architect*, Sente, Inc., 1996.