V–SAT: A Visual Specification and Analysis Tool for System-On-Chip Exploration *

Asheesh Khare akhare@ics.uci.edu
Nicolae Savoiu savoiu@ics.uci.edu
Ashok Halambi ahalambi@ics.uci.edu
Peter Grun pgrun@ics.uci.edu
Nikil Dutt dutt@ics.uci.edu
Alex Nicolau nicolau@ics.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA

Abstract

We describe **V–SAT***, a tool for performing design space exploration of System-On-Chip (SOC) architectures. The key components of V–SAT include EXPRESSION, a language for specification of the architecture, SIMPRESS, a simulator generator for analysis/evaluation of the architecture, and the V–SAT GUI front-end for easy specification and detailed analysis. We give a brief overview of the components (EXPRESSION, SIMPRESS and GUI) and, using an example DLX architecture, demonstrate V–SAT's usefulness in exploration for an embedded SOC codesign flow by specifying and evaluating several modifications to the pipeline structure of the processor. We believe that V–SAT provides a powerful environment, both for early design space exploration, as well as for the detailed design of SOC architectures.*

1 Introduction

Recent advances in System-on-Chip (SOC) technology make it possible to utilize *customizable* embedded processor cores, together with a variety of novel on-chip/off-chip memory hierarchies, allowing customization of SOC architectures for specific embedded applications and tasks. Indeed, current technology projections by the SIA[13] estimate die sizes of 50 million transistors by the year 2001, and 1000 million by 2010, but these are already proving to be conservative. Furthermore, advances in mixed memory/logic fabrication already allow substantial amounts of on-chip DRAM, with the future possibility of seamlessly combining mixed memory types (e.g., SRAM, DRAM, Flash) with logic[8].

These trends clearly present tremendous opportunities for system designers to tune and customize SOC designs for specific applications that have diverse goals such as low power, small footprint code size, hardware/software redundancy, testability, etc. However, shrinking time-to-market cycles, coupled with increasingly short product lifetimes create a critical need for tools and environments that permit the system designer to rapidly evaluate candidate SOC architectures, and also rapidly complete both the hardware and software implementations in parallel. These issues are particularly critical in the design of embedded SOCs that employ reusable soft, firm and hard Intellectual Property (IP) macros. When these IP macros include parameterizable processor cores, an additional challenge arises: the lack of supporting software tools – such as compilers, debuggers, functional-, cycle- and phase-accurate simulators – that permit both early *co-evaluation*, as well as *co-development* of hardware and software. Indeed, several companies are now offering customizable SOC designs using specific instances of parameterizable processor cores (e.g., Tensilica[16], Improv[7]). The obvious next trend is the ability to mix-and-match processor, memory and other IP blocks to build customized SOC designs.

Currently, a major bottleneck for SOC designers using processor cores is the lack of an automated software tool-kit generation environment. Traditionally, these tool-kits were built during the latter stages of system design and thus were not used to aid Design Space Exploration (DSE). However, as the potential systems design space becomes increasingly complex (e.g., through the use of parameterizable IP cores), exploration of the design space becomes virtually impossible (and meaningless) in the absence of powerful tools that aid the designer. This has created a demand for retargetable tool-kits that support rapid DSE. Typically, these tools (e.g., compilers, simulators, debuggers, assemblers etc.) are made retargetable by incorporating a mechanism with which the target machine can be described in a manner easily understood by them. We believe that language-based approaches present the best hope of describing complex systems to these tools. Architecture Description Languages (ADLs) can be used to drive both DSE and automatic toolkit generation. However, there are some drawbacks to this approach:

- Textual descriptions can be tedious and are often non-intuitive for specifying architectures.
- The length and repetitive nature of these descriptions increases the possibility of errors in specification.
- Quantitative and/or qualitative feedback from the tool to the user is not easily accomplished.

In order to combat these short-comings we have developed a graphical tool, called V–SAT, for Visual Specification and Anal-

*This work was partially supported by grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348). **To appear in EUROMICRO (DSD Wkshp) 99.***

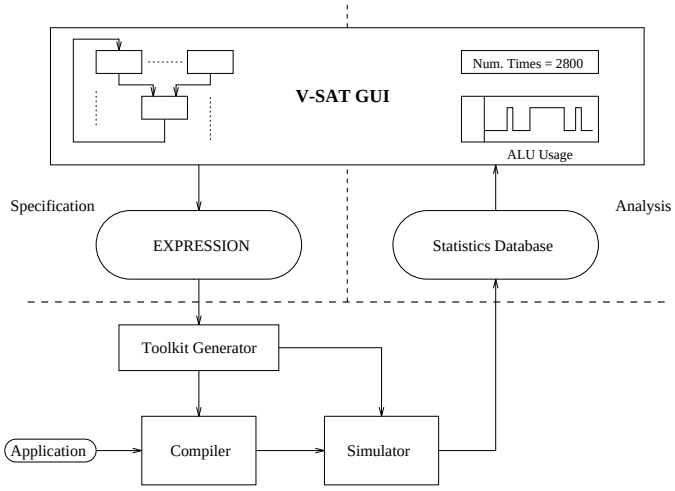

Figure 1. The V-SAT Design Space Exploration Environment

ysis of SOC architectures. As shown in Figure 1, V-SAT includes a graphical user interface (GUI) wherein the user specifies the architecture, a tool-kit generator that generates a compiler and simulator, and a mechanism for collecting and displaying statistics. This GUI-based approach retains the advantages of the ADL approach while improving on its shortcomings. The main advantages of this approach are:

- The graphical interface provides an intuitive way to specify architectures. It also allows the designer to visually identify inconsistencies and/or incompleteness in specification. This results in a drastic reduction in the time spent in specification.
- The graphical interface allows feedback from tools (like compiler, simulator) to be displayed visually and tagged to the architectural components drawn by the user. This is very valuable in a DSE environment.

Our proposed approach also retains the advantages of using ADLs to specify machines to the tools. As shown in Figure 1, the graphical specification of the architecture is converted into a specification in EXPRESSION[5], an ADL designed to support architecture exploration and software toolkit generation.

In this paper, we describe V–SAT: a visual specification and analysis tool for SOC designs. We also describe SIMPRESS, a simulator generator tool that is a part (along with V–SAT) of our DSE environment. In Section 2 we describe related work on ADLs and toolkit generators, and compare them with our approach. Section 3 presents a brief overview of EXPRESSION, an ADL that effectively supports the dual goals of SOC exploration, as well as automatic generation of a high-quality software toolkit for embedded SOC. In Section 4 we describe SIMPRESS, a simulator generator capable of generating an instruction-set simulator, as well as a cycle-accurate structural simulator. Section 5 presents a brief overview of the V–SAT architectural exploration environment. Sections 6 and 7 illustrate ease of design space exploration using V–SAT with the aid of a sample architecture, while Section 8 concludes this paper.

2 Related Work

With the advent of SOC, there has been increasing interest in using ADLs for processor specification and toolkit generation. ADLs can be classified into three categories depending on whether they primarily capture the instruction set (IS), the structure of the processor, or both.

Instruction set centric ADLs (e.g., **nML**[4], **ISDL**[1], **SLED**[12]) characterize the processor by its instruction set architecture. Architecture centric ADLs (e.g., **MIMOLA**[10]) focus on the structural components and connectivity of the processor. They have the advantage that the same description can be used for both synthesis, and toolkit generation. However, due to the very low level of description, compiler generation is made difficult.

More recently, languages which capture the instruction-set and the architecture aspects, as well as a detailed pipeline information (typically described as reservation tables) have emerged (e.g., **FLEXWARE**[2], **MDes**[17], **LISA**[3], **RADL**[14] and **EXPRESSION**[5]). These languages bridge the gap between instruction set centric languages (which are too coarse for structural cycle-accurate simulators), and hardware description languages (which are too detailed to be easily used for compiler generation). A more detailed review of ADLs can be found in [5].

The quality and type of tools generated is closely related to the ADL used. In particular, the generated simulators will reflect the type of information captured in the ADL. Pure instruction-set ADLs allow the generation of instruction set simulators, while structural ADLs generate structural simulators. In order to perform architectural DSE, the simulator has to provide structural feedback to the designer, and guide him in exploring different architectures. Pure instruction set simulators do not provide enough information to drive architectural DSE. Below we present a brief overview of existing toolkit generators (with an emphasis on generating simulators).

Sim-nML[11], an augmented version of nML, has been used for automatic toolkit generation, including instruction set simulator. However, it is not clear if Sim-nML has been used to generate the compiler. SLED provides bit-manipulating code to be used by applications that process machine code, such as compiler back-end and assembler. However, SLED does not contain semantic information needed for simulator generation. LISA has been used primarily to generate a cycle-accurate simulator [3], whose main characteristic is a behavioral model of the pipeline. The FLEXWARE system contains the CODESYN code-generator and the Insulin[15] simulator. The simulator uses a VHDL model of a generic parameterizable machine. The application is translated from the user-defined target instruction set to the instruction set of this generic machine. The **Trimaran**[17] system uses MDes to retarget the compiler back-end. However, it allows only a restricted retargetability of the simulator to the HPL-PD processor family.

Even though some of these previous approaches target ADL-based automatic toolkit generation and design space exploration, not much work has been done in bringing together these elements in a design space exploration environment. Further-

more, previous approaches are restricted to certain classes of processor families and assume a fixed memory/cache organization. However, the advent of reusable IP libraries demands a more general environment that allows a system designer to mix-and-match different types of processors with varying memory/cache organizations. For a wide variety of such processor and memory IP library families, the designer needs to be able to specify and analyze, in an intuitive manner, the interaction between the processor instruction set and architecture, and the application, and explore different points in the design space.

In our approach, we address this problem through a combination of a powerful ADL (EXPRESSION), an automatic simulator generator (SIMPRESS), and a user-friendly graphical environment for exploration (V–SAT). The EXPRESSION ADL captures both the instruction set and architecture information for a design drawn from an IP library. The library typically contains a variety of parameterizable processor cores and customizable memory/cache organizations. SIMPRESS, our simulator generator produces a structural simulator capable of providing detailed structural feedback in terms of utilization, bottle-necks, stalls and hot-spots in the processor architecture. The processor-system description is input using a graphical schematic capture tool, called V–SAT, that outputs an EXPRESSION description which is fed into tool-kit generators (e.g., SIMPRESS) to produce DSE tools. The SIMPRESS generated simulator provides feedback information which is back-annotated to the same V–SAT graphical description. This helps the SOC designer analyze the design in an intuitive manner. As graphical diagrams are easily available for most IP cores, the V–SAT environment can also be used to leverage existing knowledge in order to generate EXPRESSION descriptions.

3 EXPRESSION Overview

As mentioned earlier, our EXPRESSION ADL adopts a mixed approach to specification of processor systems that integrates both the structure (or Architecture Designer's view) and the behavior (or Programmer's view) of the system. A detailed discussion on the benefits of EXPRESSION, on tool-kit generation from EXPRESSION and on using EXPRESSION for design space exploration can be found in [5]. Since in this paper we illustrate some sample architectural exploration scenarios involving changes to the pipeline, we briefly describe how pipelines and datapaths are specified in EXPRESSION.

Pipeline and Datapath Specification in EXPRESSION

The structure (of a processor system) is defined by its components and the connectivity between these components. In EXPRESSION, each component is classified as either a Unit (e.g., pipeline units, functional units), Storage element (e.g., latches, registers, caches, memories), Port (e.g., input, output, inout ports) or Connection (e.g., busses). While connectivity between components can be easily specified in the form of a netlist, extracting high-level information (e.g., pipeline description) from the netlist is a hard task (particularly for complex processors). Therefore, the approach adopted in EXPRESSION is to allow description of the connectivity at an abstract level which captures the required high-level information and also allows generation of the netlist. EXPRESSION provides two high-level constructs which are used to describe partial connectivity between the components: *Pipeline* (used to specify the units which comprise the pipeline stages), and *Data-transfer path* (used to specify the valid unit-to-storage or storage-to-unit data transfers). These two constructs were motivated by the fact that, for most modern processors, all paths (defined as an ordered list of connected components) in the architecture either transfer control (i.e. instructions) or data between components. The transfer of instructions occurs in the pipeline while transfer of data occurs along the data-transfer paths.

The Pipeline and Data-transfer path constructs allow the user to separately specify these paths which can then be coupled together to generate the netlist. This greatly reduces the size of specification and also provides an easy, natural way to specify connectivity for most architectures.

4 SIMPRESS: Simulator Generator

Simulators are critical components of the exploration tool-kit for the system designer. They can be used to perform diverse tasks such as verifying the functionality and/or timing behavior of the system, and generating quantitative measurements (e.g., cache miss ratio) which can be used to aid DSE.

SIMPRESS is a simulator generator which reads in an EXPRESSION description and generates simulators for the target system. In the contex of DSE, SIMPRESS can be used to generate a cycle-accurate, structural simulator. The cycle-accurate, structural simulator aids:

1. Early Design Space Exploration: Early architectural exploration requires evaluation of candidate base-processor and memory/cache organizations. A structural simulator can give comparative numbers on performance and memory traffic behavior to aid the system designer in identifying promising candidate SOC processor-memory organizations.

2. Detailed Architecture Evaluation: The cycle-accurate simulation model provides detailed information that enables the designer (or other tools) to evaluate the architecture in terms of performance, power or other characteristics, and further tune the architecture. An example is the number of instructions issued per cycle (used to determine performance) or the bus transition activity (used to determine power consumption). A structural simulator that can model the execution of the bus is necessary to analyze the effects of, for example, changing the number of buses on performance and power.

3. Interactive Compilation: The simulator can be used to provide guidance to the user of an interactive compilation environment. For example, performance measures before and after a compiler transformation can be used to determine its effectiveness.

In addition to the simulator, SIMPRESS generates the hooks needed to connect the simulator to the V–SAT GUI. It also incorporates statistics collection functionality into the simulator. Below, we mention some of the salient features of SIMPRESS and present a brief overview of the simulator generation mechanism.

As mentioned earlier, SIMPRESS generates the simulator from an EXPRESSION description. In order to facilitate automatic generation, many of SIMPRESS's features are directly based on the features of EXPRESSION. Some of them are:

- Hierarchy: The components in EXPRESSION are described in a hierarchical manner. SIMPRESS incorporates the same hierarchy in the generated simulator.
- Levels of abstraction: EXPRESSION supports different levels of abstraction in specification of the architecture. SIMPRESS supports all the abstraction levels.
- Classification of components: Like in EXPRESSION, all components generated by SIMPRESS can be classified as either Units, Storages, Ports or Connections.

The architecture model employed by SIMPRESS is generic enough to be able to incorporate a wide variety of processor/memory systems. SIMPRESS assumes a two phase clock cycle model with reads (from storage) occurring in the first phase and writes (to storage) occurring in the second phase. Furthermore, units function as active components which perform computation and request data from (or send to) other components. Storages are passive components which send data to (or receive from) active components.[1] Ports and connections are used to link units to storages and vice-versa.

The structure of a system can be viewed as a set of objects that communicate with each other via well defined interfaces. Thus, an object-oriented language like C++ is very well-suited for describing a simulatable specification of the system and we have chosen C++ for this reason. SIMPRESS reads in the EXPRESSION description and generates the structural model in C++. Other benefits of using C++ include encapsulation of information local to the components, hierarchical description of components, and ease of extensibility. Below we present a brief overview of the organization of SIMPRESS.

SIMPRESS Organization

SIMPRESS is organized as a collection of libraries (that can be classified as one of four types) as shown in Figure 2:

1. **Base Components Library:** This library is the "core" of SIMPRESS. It defines each of the four component types in EXPRESSION (unit, storage, port, connection) as a class in the library. All other component types are derived from one of these classes. These four base classes define the public interface which governs the interaction between various components.
2. **Target Components Library:** This library contains a description of each component class (derived from the base component classes) that is target specific (e.g., alu, busses, etc). This library captures the functionality of the individual components that comprise the processor system and is used to capture soft, firm and hard IP libraries.
3. **Connectivity Library:** This library defines the connectivity as a netlist of components. Each component is an instantiation of a class defined in the Target Components Library.
4. **Simulator Functionality Library:** This library defines functions (e.g., single-step, run, break, etc) that can be used by the environment (e.g., GUI or compiler) to perform simulations and to control the simulator.

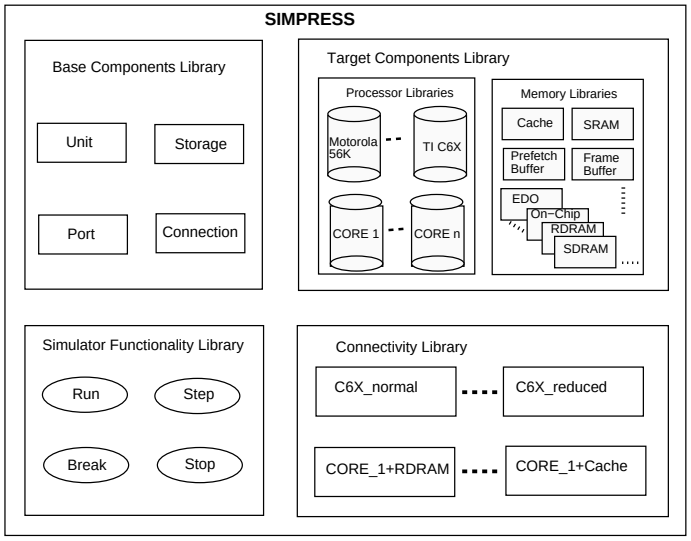

Figure 2. An organizational view of SIMPRESS

The Base Components and Simulator Functionality Library are target independent and hence do not vary from simulator to simulator. During generation of the simulator these libraries are linked with the relevant target components and connectivity libraries to form the simulator executable. The connectivity library is automatically generated from EXPRESSION. The target-specific components can either be built from the list of pre-defined components provided by SIMPRESS or can be explicitly written if necessary. In order to aid the designer, SIMPRESS provides a wide variety of parameterized components. Examples include, but are not limited to:

- Parameterized storages (e.g., Latches, Registers, Memory, Register-File, Cache).
- Operators (e.g., add, sub, mult, div, etc).
- Ports, Buses, Connections.
- A standardized pipeline controller (for stalling/flushing the pipeline).
- A standardized data-hazard detector

[1] Note: Storage elements that are embedded with computational power (e.g., IRAM) are modeled as a composition of active units and passive storage elements.

Besides these libraries, SIMPRESS also generates the hooks necessary to control the simulator from the GUI. An important feature of the simulator generated by SIMPRESS is its ability to collect various useful statistical information which enables the designer to evaluate the system. SIMPRESS allows the user to choose from an extensive set of statistics which are then incorporated into the simulator. These can be selected and pegged to any component (of interest to the designer) through the GUI. SIMPRESS then automatically generates the code necessary to keep track of the statistic. It does this by creating collector agents that shadow the execution of the components. SIMPRESS provides two classes of statistics:

1. Cumulative Statistics: These are statistics that accumulate various types of information over the course of the simulation. Examples include cycle count, resource usage, stall count (due to hazards), etc.

2. Tracking Statistics: These are statistics that track the variation (over time) of cumulative statistics. Examples include resource utilization over a period of time, bus transition activity over a period of time, etc.

Additional statistics can be easily specified and incorporated into SIMPRESS.

5 V-SAT Graphical User Interface

The V-SAT GUI (see Figure 1) was designed to aid the system designer perform Design Space Exploration. User-directed DSE usually occurs as an iteration over two stages:

1. **Specification Stage** wherein the designer describes a particular design configuration, and,

2. **Analysis Stage** wherein the designer (or tool) evaluates the specified design for various target goals.

The process is terminated when the desired objectives are met. Decreasing the time spent in the specification and analysis stages greatly reduces the time spent in DSE. A further benefit is that the designer can iterate over a larger design space to potentially determine a more favorable design configuration. The V-SAT DSE environment is designed to make specifying the architecture fast and intuitive, and to provide a detailed analysis of the architecture.

The V-SAT GUI was developed to provide the designer with an intuitive, flexible interface to the DSE tools. Corresponding to the stages in DSE, the GUI, too, has two modes of operation (described below). During the specification mode of usage, the user inputs an architectural diagram. During the simulation phase, the GUI switches into analysis mode for simulation progress and statistics display.

In either mode of operation the GUI is comprised of two parts: the *canvas* and the *palette*. The canvas (see Figure 3) is the window which holds the architecture description as a graph diagram. Each node (shown as a rectangle) in the architecture diagram represents a component of the architecture: compound unit, processing unit, storage unit, port, or latch. Component

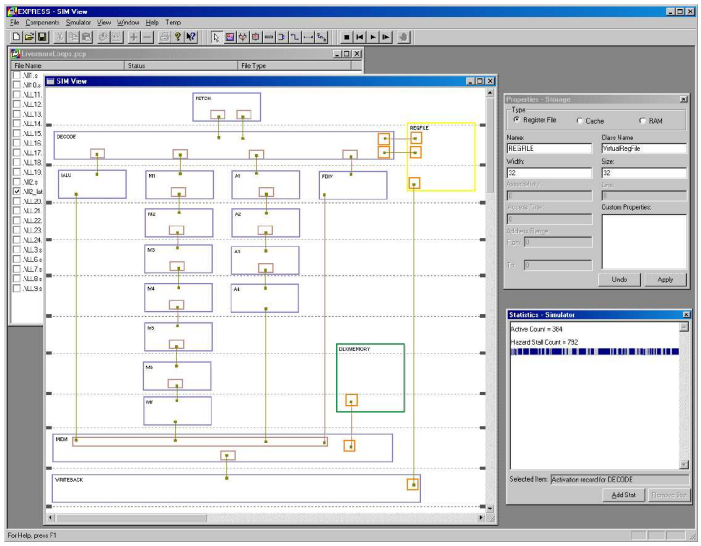

Figure 3. The V-SAT graphical interface. The window that contains the architecture diagram and the statistics is called the canvas, and the toolbar region is called the palette

attributes (e.g., unit capacity, operations supported by a unit, bit width of a storage component, etc.), that must be specified to generate a meaningful EXPRESSION specification, are input using property dialog boxes. The edges between the nodes model connections. After these components have been added to the diagram the pipeline and data-transfer paths can be defined. The pipeline stages are represented as regions of the diagram separated by dotted horizontal lines. Data-transfer paths are specified by simply selecting the components involved (in the order in which they occur during the data transfer). All diagram components have default appearances but a user can easily override that when deriving new components from the SIMPRESS base component classes.

The GUI allows full control over the architecture layout. The user can then accurately copy an existing architecture's layout (i.e. from a data handbook) and thus have a familiar view of the architecture. Future versions will also include an automatic layout tool designed to generate a customizable first-cut layout.

The second part of the GUI, the palette (see Figure 3), contains all the menus and toolbars that provide access to objects (like units, connections, etc.) used to build the architecture diagram. It also provides the necessary controls for generating the EXPRESSION architecture description (needed to drive the simulator generator) and controlling the simulation process.

The main difference between the two GUI modes of operation is in how the architecture diagram is used. In the specification mode, the diagram (i.e. components, component properties, and layout) can be modified to describe the desired architecture. In the analysis mode, changes to the diagram are disabled and the GUI uses the diagram to display simulation progress. It also displays statistics collected by the simulator to facilitate quick evaluation of the design. It is this tight integration between the architecture description and simulator GUI that makes it easy for the system designer to cycle through the

DSE stages and perform a rapid evaluation of the architecture.

6 DLX: An example architecture

We illustrate the use of the V–SAT environment for Design Space Exploration with the aid of an example processor. We chose the DLX processor[6] because it is well understood, has been studied extensively, and is simple enough to demonstrate some interesting features of DSE. Figure 4 presents the (simplified) structure of the DLX. The primary functional units are: an integer unit (INT unit) which performs integer operations and address computation for memory accesses, a 7-stage multiplier (M1 – M7), a 4-stage floating point adder (A1 – A4), and a divider (DIV). While the DLX is a relatively simple RISC processor, it does contain some interesting features that are also found in today's processors. Primary among them are:

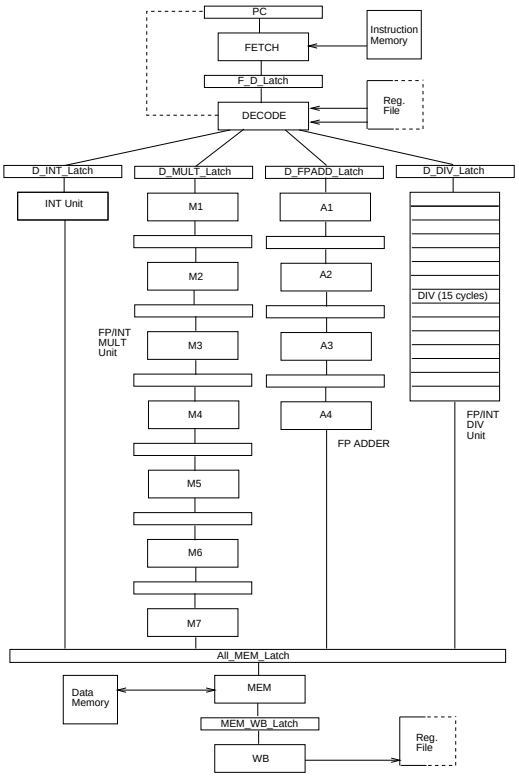

Figure 4. Structural View of DLX

- Register-based architecture: Loads and Stores are the only instructions that access data memory (in the MEM stage).
- Complex-pipeline: The DLX architecture has a fragmented pipeline with multiple execution paths.
- Multi-stage units: The multiply and floating-point add units are pipelined multi-stage units while the divider is a non-pipelined multi-cycle unit.

The DLX also has a moderately complex control unit that ensures that there are no structural and/or data hazards when instructions are issued/executed. The DLX architecture was described using the V–SAT GUI and an EXPRESSION description was generated.

For this paper, we focus on the structure of DLX, describing the pipeline and data-transfer paths in particular. The complete description of the DLX processor in EXPRESSION is approximately 280 lines and can be viewed at *http://www.ics.uci.edu/~ahalambi/EXPRESS/ADL/DLX/main.html*

7 Design Space Exploration

SOC designs have various design goals. These goals include minimal cost, maximal performance, low power, high reliability, etc. Often times, these goals conflict with each other. Design Space Exploration allows the SOC designer to make tradeoffs between these goals and arrive at an "optimal" design. Typically, the SOC designer would like to explore changes to the architecture or the instruction-set of the processor-memory system. Common examples of such changes include, but are not limited to:

- *Changing the pipeline structure.* E.g., increasing (or decreasing) the number of stages to increase (or decrease) the clock frequency, adding forwarding paths to reduce pipeline stalls.
- *Changing the data path structure*. E.g., changing slow units to fast units in order to increase performance, changing connectivity between units and storage elements (like register files) in order to decrease power consumption.
- *Increasing parallelism*. E.g., adding more functional units that can execute in parallel in order to increase performance.
- *Changing the instruction-set*. E.g., adding new operations which can be exploited by particular applications (like multiply-accumulate for DSP).
- *Changing the memory components*. E.g., changing the size of the register file, changing the associativity of the cache, etc.
- *Changing the memory hierarchy*. E.g., adding a cache between the processor and off-chip memory, changing the on-chip memory hierachy etc.

The specify-simulate-analyze methodology of Design Space Exploration works well in the SOC domain because most SOC designs are targeted to a class of applications (e.g., multimedia applications, or networking applications). Since the system designer knows the application(s) up-front, he can fine-tune the architecture by simulating it with the representative application(s) and analyzing the results of the simulation.

To test the usefulness of V–SAT in architecture exploration and to highlight its ease of use, we performed some experiments on the DLX processor. While V–SAT supports comprehensive design space exploration, for purposes of illustration in this paper, we experimented with changing the pipeline structure of the DLX and evaluated those changes on a set of benchmark programs (other experimental results are detailed in [9]). The changes to the pipeline are:

1. Adding a forwarding path from All_MEM_Latch to A1.
2. Adding a forwarding path MEM_WB_Latch to INT unit.
3. Adding both (1) and (2).
4. Adding (1) and a forwarding path from All_MEM_Latch to INT Unit.
5. Adding (1) and a forwarding path from MEM_WB_Latch to A1.

To show the effectiveness of the V–SAT environment, and to demonstrate its ease of use in DSE, we tested the architectural changes to DLX (mentioned above) on a representative sample of benchmarks. In this paper we present results of DSE using 10 benchmarks. The first six (LL3, LL4, LL7, LL9, LL14, LL19) are from the Livermoore loop test-suite and the other four (Laplace, Linear, Lowpass Wavelet) are multimedia kernels. Detailed experimental results on other benchmark programs can be obtained from [9].

We first analyze the effects of adding forwarding paths (changes 1-5 above) to the DLX pipeline. Figure 5 presents the results of DSE using the forwarding options mentioned above.

Forwarding paths bypass some stages in the pipeline in order to feed the result of one stage to an instruction in a previous stage that needs the result. This helps to eliminate pipeline stalls due to read-after-write (RAW) data hazards. This is most useful in DSP-type applications which perform sequential transformations on data. An example is the multiply-accumulate, where the result of a multiplication is needed by the addition that immediately follows it.

The first bar (in Figure 5) represents the performance improvement (as compared to the architecture without forwarding paths) due to the forwarding path to the floating-point adder only. As can be seen, applications (such as Lowpass, Wavelet) that contain multiply-accumulate instruction pairs benefit the most from this forwarding path.

The second bar (in Figure 5) represents the performance improvement due to the forwarding path to the integer unit. The source of the forwarding path is the output of the memory stage because this allows us to exploit instruction sequences which contain loads followed by integer computation. As can be seen, applications (such as LL19, Laplace, Linear) which perform a lot of integer computation benefit from this path.

The third bar (in Figure 5) represents the performance improvement due to a combination of both the previous forwarding paths. As can be seen, most applications benefit from the combination.

The fourth bar (in Figure 5) represents the performance improvement due to adding a forwarding path from the All_MEM_Latch to the integer unit along with the first forwarding path. As can be seen, the additional path does not really improve the performance over the first path. This is because most of the multiply-accumulate type of operations are floating-point (in the benchmarks we chose) and hence the additional forwarding path is rarely used.

The fifth bar (in Figure 5) represents the performance improvement due to adding a forwarding path from the output of the MEM stage to the floating-point adder along with the first forwarding path. As can be seen, the additional path does not really improve the performance over the first path. This is because most floating-point operation sequences follow the load-multiply-add-store order. An additional path from the stage which performs the load to the stage which performs the addition does not have any benefit because of the intervening multiplication (which is performed on a different unit).

In conclusion, while a further analysis of the cost of adding forwarding paths is necessary to determine their usefulness, it can be seen that adding forwarding paths which exploit the features of the application (e.g., forwarding paths 1 and 2) is beneficial. In this manner, the system designer can exploit the nature of the application (multimedia in our example benchmarks) to intelligently explore the design space and fine-tune the architecture.

Further, using the GUI, adding forwarding paths was a matter of just making the connections between the source (latch) and sink (unit) components. This is a very intuitive way of specifying changes to the architecture and simplifies the task of the SOC designer. The usefulness (of V–SAT) is further demonstrated by the fact that during the analysis phase, the statistics collected by the simulator were very helpful in determining where to place the forwarding paths. Simulation of the baseline DLX architecture (without forwarding paths) yielded the result that, on an average, the pipeline was stalled (due to a RAW hazard) 53% of the time. This, combined with the statistics which indicated that the integer unit and the floating-point adder were the most used units, enabled us to decide on the proper placement of forwarding paths.

We also conducted other DSE experiments, the full results of which are not presented here for lack of space. We give a short summary of these results. We experimented with changing the number of stages in the floating point execution units. We tried changing the floating point multiplier from 7 stages to 5 and the floating point adder from 4 stages to 3 to simulate the use of faster units. The performance improvement by changing the multiplier was not significant since there are relatively few multiplications in the examples. Changing the adder stages gave a 6% improvement. The effect of reducing the number of stages as above and including the forwarding paths from the last experiment that gave the best improvement viz, All_MEM_Latch to A1 and MEM_WB_Latch to INT unit, gave an average improvement of 32% over all the benchmarks we considered. Detailed results and analysis of these experiments can be seen in [9].

8 Summary

System-on-a-Chip (SOC) technology, coupled with the rapidly increasing availability of soft and hard IP libraries, enables the system designer to develop highly customized embedded systems that can meet demanding performance, power, cost and size constraints. However, there is an urgent need for an environment that will allow the system designer to rapidly specify and evaluate design alternatives. In this paper we presented V–SAT, a new environment for DSE that leverages on existing ADL technology. The graphical interface allows the user to

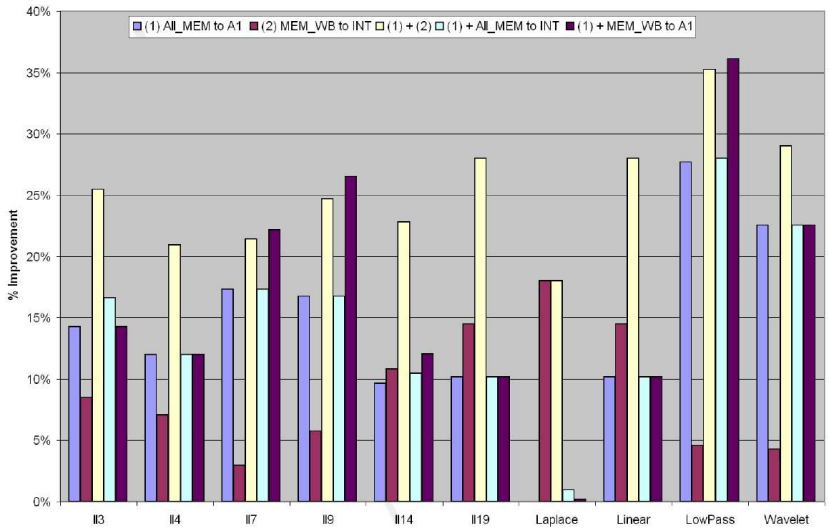

Figure 5. Speedups obtained by adding forwarding paths to the DLX pipeline.

specify the architecture in a visual form. V–SAT then generates an EXPRESSION ADL description which can be used to automatically generate tools such as compilers and simulators. We also described SIMPRESS, a simulator generator that can be used to generate a detailed, structural simulator. V–SAT allows the user to collect detailed statistics (such as stall cycles, resource usage, etc.) which are back-annotated to the GUI to help the user analyze the design and modify the critical portions. To illustrate utility of the V–SAT environment, we performed DSE on the pipeline of the DLX processor. This is only an indicator of the power of the V–SAT environment in aiding the SOC designer perform architecture exploration. Our on-going work involves performing comprehensive DSE on more complex architectures (like the Texas Instruments C6X processor system). Future work includes providing tools that analyze the detailed statistics generated by the simulator and the specification of the system in order to offer useful suggestions to the system designer. We believe that such tools will be useful in the context of Systems-on-Chip that are domain specific (e.g. multimedia) and whose applications exhibit similar characteristics.

9 Acknowledgements

This research was supported in part through grants from NSF (MIP–9708067) and ONR (N00014-93-1-1348). We would like to acknowledge and thank Vijay Ganesh and Weiyu Tang for their contributions to the SIMPRESS project.

References

[1] G. Hadjiyiannis et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[2] P. Paulin et al. FlexWare: A flexible firmware development environment for embedded systems. In *Proc. Dagstuhl Code Generation Workshop*, 1994.

[3] V. Zivojnovic et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[4] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, March 1999.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, 1996.

[7] Improv Incorporated. *http://www.improvsys.com*.

[8] D. Keitel-Schulz and N. Wehn. Issues in embedded DRAM development and applications. In *Proc. ISSS*, December 1998.

[9] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. Technical report, University of California, Irvine, 1999.

[10] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[11] V. Rajesh and Rajat Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, January 1999.

[12] N. Ramsey and M. Fernandez. Specifying representations of machine instructions. In *Proc. ACM TOPLAS*, May 1997.

[13] Semiconductor Industry Association. *National technology roadmap for semiconductors: Technology needs*, 1998.

[14] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, December 1998.

[15] S. Sutarwala, P. G. Paulin, and Y. Kumar. Insulin: An instruction set simulation environment. In *Proc. CHDL*, April 1993.

[16] Tensilica Incorporated. *http://www.tensilica.com*.

[17] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.