

# An Efficient and Extendable Modeling Approach for VLIW DSP Processors

Naser Sedaghati-Mokhtari, Mahdi Nazm-Bojnordi, Abbas Hormati,  
and Sied Mehdi Fakhraie

Silicon Intelligence and VLSI Signal Processing Laboratory  
School of ECE, University of Tehran, Tehran 14395-515, Iran  
n.sedaghati@ece.ut.ac.ir, m.bojnordi@ece.ut.ac.ir,  
abbas@excelicon.com, fakhraie@ut.ac.ir

**Abstract.** This paper presents an efficient and extendable modeling approach for DSP processors with VLIW architecture. The proposed approach is designed for sequential implementation platforms such as C++ programming language. It employs specific pipeline modeling technique called reverse calling. As a sample implementation, a DSP processor model is designed based on Texas Instruments (TI) C62xx architecture. The processor model handles pipeline resources (memories and register files) during concurrent accesses by updating method. To verify the functionality of the model, a cycle-accurate simulation environment is developed using C++ programming language. In this simulator, a DSP-specific data type, called DSPDT, is designed and implemented for bit-accurate implementation of signal processing operations. The simulation environment utilizes a simple assertion-based verification technique with messages using three levels of severities: Alert, Warning, and Error. The simulator is functionally validated by practical DSP benchmarks such as IIR filters, correlation, FFT blocks and also the G.729a speech codec for single and multiple speech channels.

**Keywords:** DSP Processors, VLIW Architecture, Cycle-Accurate Simulator.

## 1 Introduction

Simulation tools are essential aides to both designers and researchers in computer architecture, due to their ability to study and validate new designs without the cost of actually building the hardware. They also provide for a good platform on which researchers can explore a wide range of design choices, which might be practically not feasible. Simulation tools also allow one to study the combined interaction of all the architectural features, before anything is built and can bring out potential bugs that might not have otherwise been detected [1], [2].

In this paper, we present an efficient and extendable processor model for VLIW DSP architectures. We verify the modeling approach by Texas Instruments (TI) C62xx DSP processor. The processor model employs a specific pipeline modeling technique called reverse calling. We present a cycle-accurate simulation environment for the processor model. We validate the simulator by standard DSP benchmarks such as IIR filters, correlation, FFT blocks and G.729a speech codec.

The rest of the paper will discuss related work in this area of research, modeling issues and approaches, a brief overview on the target processor architecture, environmental considerations, verification and validation strategies, experiments and concluding words.

## 2 Modeling Approach

Multi-issue VLIW processors usually followed the regular pipeline structure. In contrast with the Superscalar architectures, the pipeline of the VLIW processors is usually free of any conflicting backward paths (i.e., forwarding unit) [3]. Using this benefit, we propose a modeling technique for VLIW DSP architectures (including pipeline structure, register file and memory, and processor core) in sequential platforms (i.e. C++ programming language).

Before describing the modeling approach, we present two concepts in this area.

### 2.1 Single vs. Multiple Stage Units

According to the timing and access methodology, the processor pipeline has two kinds of units: single-stage and multi-stage. Majority of the units in a VLIW processor model are single-stage in which the overall function of that unit is started, continued, and terminated in the same pipeline stage. Example of these units is decoding stage(s). In each cycle, each single-stage unit receives requests only from preceding neighbor unit in the processor pipeline. In other words, the input and output interfaces of a unit in stage  $N$  are with stages  $N-1$  and  $N+1$ , respectively. Another group of units, multi-stage units, are that influenced by the pipeline in more than one stages in each execution cycle. Generally, the entire pipeline shared resources such as register file, memory and I/O interfaces are multi-stage units.

### 2.2 Single vs. Multiple Cycle Units

For stage-based categorization, we consider simulation cycle which is demonstrated spatial distribution of VLIW instruction packets. Changing the point of view from spatial to temporal distribution of instruction packets, the processor units divided into two categories: Single-cycle and multi-cycle. Dispatch is the only multi-cycle unit which is continued operating for a single instruction packet to one cycle or more. Other simulator units belong to single-cycle category.

Accordingly, multi-cycle objects produce new handling issues such as pipeline exceptions and interfaces. On the other hand, multi-stage objects require special considerations for concurrent accesses. These issues are addressed by message passing strategy and updating mechanism.

### 2.3 Message Passing Strategy

Implementing the precise pipeline structure imposes a message passing strategy. In this simplest form, all of the output variables of each pipeline stage objects are considered as message variables. The pipeline stage objects communicate together through these variables. Each object has a *run()* method which is called when the

corresponding operation is required. For maintaining the pipeline structure, we call the *run()* method of all pipeline objects in the reverse order. Thus, we called this strategy as Reverse Calling. This technique is used to prevent undesirable transmissions of message variables through pipeline stages.

## 2.4 Updating Mechanism

To handle multiple access requests, each of the shared resources (multi-stage objects) has a method called *update()*. At the execution time, some components send their requests to the target multi-stage objects. At the end of simulation cycle, and when all requests are received and gathered, the target object updates its contents. This way, validations and access limitations are checked during the updating step.

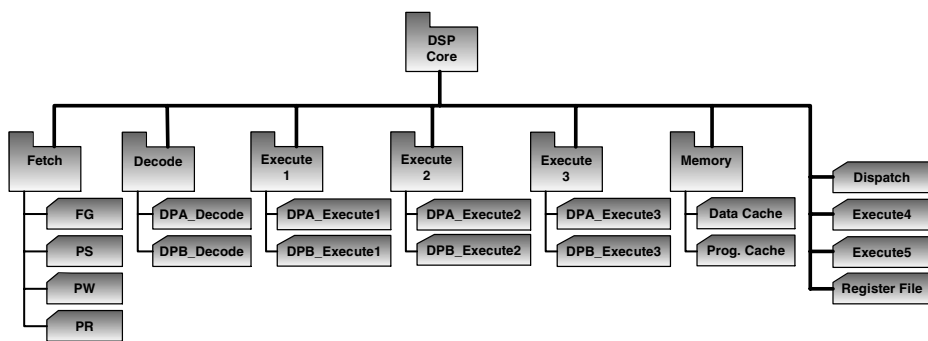


Fig. 1. Class Hierarchy of the processor model

## 3 Processor Model

We exemplify the modeling approach by Texas Instruments (TI) C62xx DSP processor [4], [5]. It is a fixed-point device employed the VelociTI™ architecture. The core of the CPU consists of 32 general purpose registers of 32-bit lengths, which can be combined to store 16 long values of 40-bit lengths. The core can execute up to eight instructions every cycle, one each for the eight functional units it has. The functional units are divided into two similar sets of basic functional units. Each set can access its own one half of the register file directly and the other half can be accessed using cross paths. The two halves are named 'A' and 'B'. More architecture details are available at [4], [5].

The exemplified model consists of several numbers of classes which are familiar to the hardware modeling concepts. The hierarchical design for classes makes the implementation process easier and less complicated. Each block of a pipeline stage is considered as an independent class with its own methods and attributes. The main class hierarchy is demonstrated in Fig 1.

### 3.1 Pipeline Model

For modeling the real pipelining operation, we employed the clocking concept of the real processor hardware. At the clock edge, each stage component is fired. This

caused the run method of each component to be called at the time of activation. The delay elements (registers) have not been modeled explicitly in the simulator; they are considered implicitly embedded by reverse calling and updating mechanisms. The update mechanism for shared resources is performed at the end of each simulation cycle.

Connecting all the stages from Fetch to Execute in a forward direction may cause propagation of unexpected faulty values into the pipeline stages. Avoiding this problem, all the pipeline stages are executed in the reverse direction, called reverse calling technique. Using this strategy, at the beginning of each simulation cycle, each stage will have its registered values from the previous stage. These values are actually updated at the previous simulation cycle (by calling the *run()* methods of pipeline objects) and correctly utilized in the current cycle.

### 3.2 Register File Model

General register file model consists of two internal parts (A and B) which are instantiated from the same class, i.e. *RegFilePart*. Considering real reading and writing constraints, a controlling mechanism is employed to optimize port assignment to the functional units and prevent any unpredictable conditions. Each register file part (A and B) has 10 read ports and 6 write ports. One of the read ports is dedicated to cross read operations. For practical implementation, and according to the decoding information, a multiplexing mechanism is employed to direct the read registers from the limited read ports to the desired source operands of the requesting functional units.

For the write operation, where the number of writes is greater than the number of available ports, we multiplex the operations and assign them to the available write ports. In this way, when all the functional units perform their write accesses to the register files (add their write request address to the write address queue), and therefore create a write transaction, the requested accesses are processed.

Limited numbers of selected writes will be directed to the available write ports. In this way, after the termination of all executions in a simulation cycle, the register file is updated. This means that all the pending write requests are committed at the end of each simulation cycle. For the case of multiple assignments to the same port, multiplexing with suitable priority mechanisms are considered to make the result of all the register file accesses fully predictable.

It should be noted that, from the assembler point of view, it is impossible for a register file to be accessed (read or written) more than the number of available ports. It is also impossible to be written more than one to a single write port in a single cycle.

### 3.3 Memory Model

Two kinds of memories are designed for the environment: Program and Data memories. All the memories' parameters are selected according to the TI's reference model. The Fetch stages (PG, PS, PW, and PR) are interfaced to the Program memory while D units are interfaced to the Data memory. Memory models provide all the real memory and caching behaviors. Therefore, one can easily use the proposed environment for analysis and evaluation of the DSP applications according to its reported run-time statistics.

## 4 Processor Simulator

For implementing the model and verifying its functionality, we develop a cycle-accurate simulation environment. The simulator is intended to simulate the real behavior and function of the target VLIW processor. In order to execute real bit-accurate DSP operations of various bit widths, the simulator employs a specific type of data as a C++ class which is called DSPDT.

### 4.1 Implementation

At the center of the simulator, there is a specific data type designed and developed for accurate DSP simulation. The data type, called *DSPDT*, is developed to model all the DSP operations practically. The simulator blocks are working with signals which are only inherited from this data type. The *DSPDT* methods and attributes make available an accurate, bit-true data type for modeling and simulation of DSP operations in C++. Also, this data type provides capability to monitor objects of the simulation environment based on real DSP behaviors, such as saturation.

### 4.2 The Simulator Architecture

The simulator model, in addition to datapath and controller, consists of memory, register file, and statistical reporting and monitoring (SRMU) units, as shown in Fig 2-a.

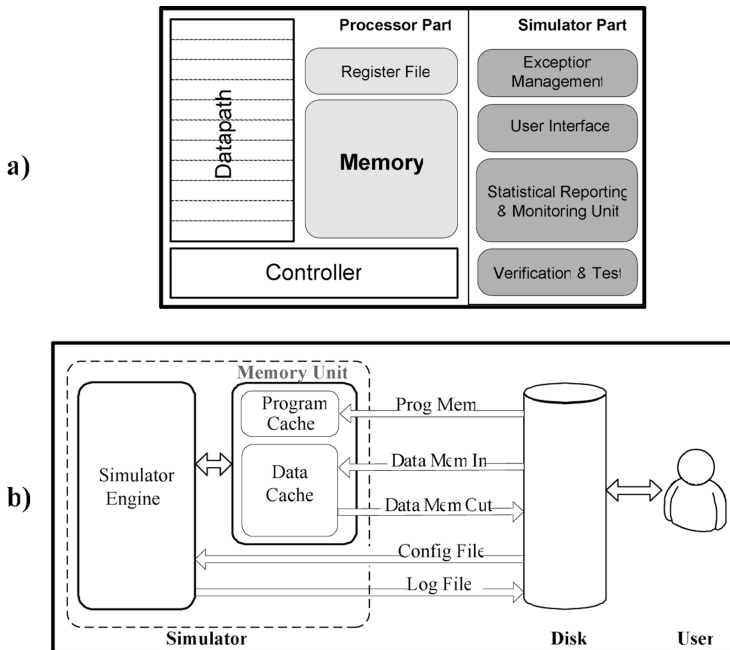


Fig. 2. a) Simulator architecture and b) Interaction between user and simulator

The SRMU is responsible for gathering all the user-requested information from the simulation. This unit is interfaced to standard file system to store the monitoring and statistical reports in the desired files. Employing such a file-based interfacing mechanism, user is able to access and control the simulator easily. Fig 2-b shows the possible relation between user and the simulation environment.

Four consequent operations construct the main simulation process. At the first step, the simulator provides a copy of all the pipeline controlling and state variables. The pipeline objects utilize these copies at each cycle. At the second step, the simulator call the *run()* methods of the stage objects. This step includes synchronization and providing message variables. Updating shared resources through update mechanism is the third step of simulation round. Finally, by calling the SRMU, simulator reports the statistics gathered during the monitoring of the model.

## 5 Experiments

We have integrated our model using all the classes with their methods and attributes. The developed model is compiled using GNU C++ Compiler. The testing strategy is based on TI's CCStudio toolset. We have applied tests to both models: ours and CCStudio. For validation, DSP microbenchmarks such as IIR filters, Correlation and FFT block, and so on are utilized. Running these microbenchmarks on the proposed simulator is feasible by the cycle count listed in Table 1 Table .

For IIR filters,  $n$  determines number of cycles required for processing of each input sample. For FFT,  $n$  is the size of the FFT block. In autocorrelation,  $n$  and  $m$  determine number and length of correlations, respectively. We note that mentioned delay clock cycles are considered with counting no cache miss cycle. Executions of the mentioned benchmarks are validated by Code Composer Studio toolset [8] and verified by the developed bit-true models in MATLAB.

**Table 1.** Execution of real DSP microbenchmarks

$\mu$ benchmark	Clock Cycle
4th order IIR	$10 \times n + 6$
6th order IIR	$15 \times n + 6$
Radix-4 FFT	$\text{Log}_4(n) \times (10 \times n/4 + 29) + 36 + n/4$
Autocorrelation	$n \times m/2 + 31 + (n/2-1)$

For simulation of the voice processing applications, we employ G.729a [6] standard reference code which is developed and released in C language. We modify the code for real DSP execution. Table 2 demonstrates the obtained results of G.729a speech codec for 10ms (one frame) single-channel voice data. Although the code is not obviously well optimized, but it satisfies the simulation requirements for application evaluation.

**Table 2.** Execution of single channel G729a

Measure	Instruction Count	Simulation Cycles
Encoder	213526	81565
Decoder	90541	34823

Regarding multi-channel simulation of G.729a speech codec, we modify the reference ITU code to support reentrancy capability. The obtained results demonstrate that, considering the real processor implementation, the model supports up to 10 real-time voice channels while theoretically the architecture should be able to process more real-time voice channels [7]. This limitation is imposed because of the inefficiency of the application code, many frequent memory accesses and context-switching overheads during multichannel operations. The performance and also implementation complexity of the proposed approach, in comparison with the previously presented models (C6XSin [8] and SimpleDSP [10]) approve the achieved performance gain.

## 6 Conclusion

In this paper, we have presented an efficient and extendable modeling approach for DSP architectures with VLIW architectures. For this purpose, a cycle-accurate simulation environment based on reverse calling technique is designed and presented. The environment is verified by developing processor model of the Texas Instruments TMS320C6211 (C62xx) DSP processor using C++ programming language. A DSP-specific data type, called DSPDT, is designed and utilized for real implementation of practical processor environment. An assertion-based verification is employed to verify the code and provide messaging and monitoring facilities in different levels. The simulator, and accordingly the modeling approach, is functionally validated using DSP benchmarks employing such as IIR filters, autocorrelations, FFT blocks, and G.729a speech codec.

**Acknowledgments.** Naser Sedaghati-Mokhtari wishes to express his gratitude to Iran Telecommunication Research Center (ITRC) for financial support of his research.

## References

1. Rosenblum, M., Herrod, S.A., Witchel, E., Gupta, A.: Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology* 3, 34–43 (1995)
2. Bechem, C., Combs, J., Utamaphethai, N., Black, B., Shawn Blanton, R.D., Shen, J.P.: An Integrated Functional Performance Simulator. *IEEE MICRO* 19, 26–35 (1999)
3. Patterson, D.A., Hennessy, J.L.: *Computer Architecture: A Quantitative Approach*, 2nd edn. Morgan Kaufmann, Menlo Park (1996)
4. Turley, J., Hakkarainen, H.: TI's New C6x DSP Screams at 1,600 MIPS. *Microprocessor Report* 11, 1–4 (1997)
5. TMS320C6000 CPU Instruction Set Reference Guide. Texas Instruments Literature Number SPRU189C (1999)
6. ITU-T Draft Recommendation G.729. Coding of Speech at 8Kbps Using the Conjugate Structure Algebraic Code Excited Linear – Prediction (CSACELP) (1996)
7. TI SPRA564B. G.729/A Speech Coder: Multichannel TMS320C62x Implementation (February 2000), <http://www.ti.com>

8. Texas Instruments. Code Composer Studio IDE Version 2.2 (August. 2003)  
<http://www.ti.com/tmwccs>
9. Ringenberg, J., Oehmke, D., Austin, T., Mudge, T.: SimpleDSP: A Fast and Flexible DSP Processor Model. In: Workshop on Media and Streaming Processors (MSP5) in IEEE/ACM MICRO-36, San Diego (2003)
10. Barbieri, I., Bariani, M., Raggio, M.: A VLIW architecture simulator innovative approach for HW-SW co-design. In: IEEE International Conference on Multimedia and Expo (ICME), New York, pp. 1375–1378 (2000)