Efficient DSP simulation for Multimedia Applications on VLIW architectures

IVANO BARBIERI, MASSIMO BARIANI, ALBERTO CABITTO, MARCO RAGGIO
Department of Biophysical and Electronic Engineering
University of Genoa
Via Opera Pia 11A
ITALY

Abstract: - This document describes a Simulation Environment for benchmarking a Multimedia application on VLIW DSP and for testing architectural modifications to improve performance (in terms of execution time, required memory, power consumption). Instruction set, pipeline and cache memory simulations flexibility allows using this environment on a large number of Applications and VLIW architectures. A case study has been implemented in order to validate the described approach.

Key Words: VLIW, Simulation, ILP, Hw - Sw Co-design, DSP, Multimedia, Development Tools.

1 Introduction
The term VLIW (Very Long Instruction Word) [1] is referred to a project philosophy where to the RISC design is added the ILP concept (Instruction Level Parallelism) [2]. Considering ILP there are two possible viewpoints: the ILP available in a region of code and ILP achievable in the given architecture (Hw-ILP) [3]. Given the Hw-ILP, the programmer tries to write code exploiting the available ILP as much as possible. On the other hand, the architecture designer analyzes the typical application code to produce the same Hw-ILP. The Hw-ILP is realized as following:

- Several functional unit can execute at the same time
- Multiple copies of functional units accessing different registers file
- Functional unit with latency longer than one cycle can be pipelined

Multimedia and in general digital signal processing applications have typically large available ILP [4] and data access specific requirements: repetitive numeric calculation, attention to numeric fidelity, high memory bandwidth, real-time processing [5]. Even thought a numbers of general-purpose processors with DSP extensions suitable for DSP task are nowadays available, the use of a DSP processor is still advantageous especially for cost-performance rate and power consumption [6] [7].

The purpose of the simulation environment described in this document is to allow the designer to find the best matching between available and achievable ILP through the interactions between code-development and architecture-parameters tuning tools. An example of this interaction is the possibility to modify the instruction set. In a high-level language development it means to re-compile the code with the new instruction set. In an assembly development it allows to directly fit the available instruction to the application. Instruction set modifications represent architecture modifications in terms of data path (connections between registers, integrated co-processor arithmetical/floatig unit modifications) [8].

The environment scope is the VLIW architecture simulation in general. The simulator flexibility has been validated on some state of the art VLIW-DSP processors [9][10][11].

2 The Simulation Environment
The simulator provides several functionalities to help the programmer in validating source code and to reconfigure the target architecture.

2.1 Debugging Tools
Two main features of this kind are available in the simulation environment: the possibility to stop the simulation in a predefined point (break point) and view the status of the simulated processor, and the step-by-step execution.

The programmer can insert breakpoints directly from the assembly code or from the high-level source code. When simulation stop on a break point, it is possible to view the contents of general-purpose registers and memory locations the pipeline status of the simulated target architecture, and simulation statistics such as number of machine cycle elapsed from the start of execution, data and instruction cache miss rate, resource exploitation and ILP rate [11].

In the step-by-step execution mode it is possible to check the simulator status each machine-cycle: Each cycle is possible to display the complete pipeline status. Therefore the user is able to monitor the simulator status every machine cycle: it is possible to view the contents of general-purpose registers and memory locations the pipeline status of the simulated target architecture, and simulation statistics such as number of machine cycle elapsed from the start of execution, data and instruction cache miss rate, resource exploitation and ILP rate [11].

In the step-by-step execution mode it is possible to check the simulator status each machine-cycle: Each cycle is possible to display the complete pipeline status. Therefore the user is able to monitor the simulator status every machine cycle: it is possible to view the contents of general-purpose registers and memory locations the pipeline status of the simulated target architecture, and simulation statistics such as number of machine cycle elapsed from the start of execution, data and instruction cache miss rate, resource exploitation and ILP rate [11].
2.2 Instruction Set Dynamical Generation

The simulation is also a Hardware-Software co-design tool for embedded systems. The purpose is to provide a development chain from application selection and implementation down to test and profiling on various targets processor/architecture through simulation. The simulator instruction set flexibility is the main feature to allows a fast way to preliminary test architectural design evaluations [3][12]. A language to describe general VLIW architecture has been identified. From a user point of view the description of each instruction must contain: mnemonic name, instruction class, type of operand(s) and destination, latency, operation code and the expression that define the instruction. If the instruction uses a control register, the description will contain the field to specify the used register (or parts of the register). Other characteristics can be specified depending on the instruction class. Instructions are divided into three different classes. Each class identifies a specific instruction type: memory operation, arithmetic-logical operation, branches. The description language allows the user to completely define instruction’s behavior through the expression field. In the expression field, it is possible specify in a C-like notation, all the relations between operands resulting as destination value.

The user-defined instruction set is taken as input of the Instruction Set Dynamical Generator (ISDG) parser. The parser analyze the description and produces an intermediate instructions representation used to produce the Instruction Set modules.

2.3 Cache reconfiguration

Compilers for VLIW architectures incorporate techniques to exploit ILP. ILP techniques are less effective in removing memory stall time than CPU time, making the memory system a greater bottleneck in ILP-based systems respect to RISC-CISC compilers. Moreover, external cache misses can take hundreds of processor cycles, mainly for the growing gap between processor and memory speeds [13][14]. Memory performance should be carefully evaluated, especially in multimedia applications where large amounts of data are involved. The cache miss rate is a key parameter in application’s optimization. A parametric cache simulation module has been realized to measure the performance of a target application on different cache models. User can modify cache size and line size, and select one in a set of cache update strategy.

3 The VLIW Simulator Structure

The simulation system is composed by a set of re-usable sub-blocks. The first module is the CODE_LOADER which loads and parses COFF(Common Object File Format) or ELF (Executable and Linkable Format) files locating the executable file's sections and other simulation required information. The next module in the simulator structure is the memory model (MEMORY). This module provide a set of procedure to simulate how the VLIW processor write/read memory. Other simulator block are Processor Pipeline (PIPELINE), Instruction Decoding and Instruction Execution. The Registry files and the External devices interface represent the simulation shell around the simulation core.

3.1 Memory Model

The memory model description and implementation of the simulator environment support the read/write operation performed by the processor, following typical DSP-VLIW memory operational mode (e.g. 32 bit WORD, 16 bit HALF WORD and BYTE addressing).

The memory simulation purpose is to allow other module: Instruction Decoding and Executing, peripheral devices (program loader, etc.) to send addresses and to read and write data without allocating a huge amount of the platform resources of memory (Host Processor). The simulator uses a two-step indexed mechanism to allocate/access needed memory. Lets consider a 32 bit address bus. The 32-bit address space is partitioned in 64 KB memory pages. The address 16 most significant bits are used as PageTable index and the less significant as offset. Each time the memory receives an address the 16 most significant bits are checked to verify if the page containing the referred address has already been allocated. If the page isn’t been allocated there are two cases:

- The operation in progress is a memory write: the page is allocated and then data are written. The operation is a memory read: it is generated a memory fault error, because the program is trying to read a memory locations on a never used/allocated memory page.

3.2 Instruction Simulation

The instruction simulation is composed by the fetch decoding and execution phases. From the simulation point of view, the number of cycles required in this phases are a configuration parameter. What actually happen is a data transfer of the fetch packet (FP) pointed by the Progran Counter (PC) from the memory to the first pipeline entry.

3.2.1 The Instruction Decoding

The Instruction decoding is divided in two phases: Dispatch (DP) and Decode(DC).

During the dispatch phase the instruction is associated with one between a number of “functional categories”. This phase is also used to calculate the Instruction Table index. The Decode routine is able to extract instruction fields selecting the right syntax thanks to the DP instruction-categories association. instruction set modularity, re-usability and easy modifiability have been achieved through indexed function call and dynamical instruction set generation (see chap. 2.2) Instruction execution has been simulated using ar
3.3 The Register File Management

The simulation is based on the progressive pipeline status updating taking into account the data coherence in memory locations and in the register file. To support data coherence two Register files have been used: one for the current Register File status and the other one for the following. Each time an instruction is executed its operands are loaded from the current register file and results are stored in the following. This mechanism guarantees data coherence through the pipeline flow one temporal step after another.

3.4 Pipeline Simulation

The pipeline status is represented in the simulation as a three-dimensional space \((\text{phase}, \text{operation}, \text{time})\). With \text{operation} is meant the instruction’s position in the FP, \text{phase} is the pipeline’s phase and \text{time} is the given time stamp. The entire simulation process is based on the fetch packet flow through each phase of the pipeline. The basic element of the pipeline status description is the \text{Operation} element, described by the following fields:

- \text{opunit}: This field contains the instruction-categories extracted in the DP phase.
- \text{interf}: Is the Common Interface for the described instruction.
- \text{instr}: The n-bit instruction as it is in the program memory (.text section).
- \text{index}: The instruction index to address the instruction in the Instruction Table.
- \text{addr}: The address of the instruction in the program memory
- \text{Latency}: The number of Execution cycles used to store results in the register file.

A two-dimensional array of \text{Operation} elements is the Pipeline Status \((\text{Pipeline \_Status})\), where the first index is the pipeline phase and the second one is the position of a certain instruction in the fetch-packet. The simulation process is based on two arrays like the one described above, to represent the current and the following pipeline statuses.

![Pipeline Status array](image_url)

Fig 1 The Pipeline \_Status array

The basic Pipeline \_Status array element is a pointer to an Operation element. The first raw contains the last-fetch Instruction packet.

The Pipeline \_Status progression is based on the following idea: instructions in a given pipeline phase in the current pipeline status are processed and transferred to the next pipeline phase in the following pipeline status. How the instructions are processed depends on the instruction type and on the phase they are. At each machine cycle the pipeline status is processed, it means that for each pipeline phase actions are performed depending on the instruction type and on the pipeline phase; then instructions are moved to the next pipeline phase.

After all the phases have been updated the current pipeline status is turned into the following.

4 Simulator Performance on a case study: TMS320C6x

The described environment has been designed to simulate a wide range of VLIW or VLIW-like architecture. A comparison with a state of the art simulation tool to evaluate performance in terms of instruction-simulation time is reported in the following tables. The target architecture is the TMS320C6x. The simulator compared is the Fast Instruction Simulator from TI (TIFS). Two different benchmarks have been used: H.263+ [15] coder and G.723.1 [16][17] encoder and decoder, both implemented in C.

The platform used for the test is a Pentium II 450 MHz, 128 MB RAM, Windows NT 4.0. OS.

In the following tables tests results about simulation time and number of cycles are shown. For the video test the complete encoding of 5 QCIF frame of the standard sequence “Foreman” has been performed. For the audio test 5 audio frames have been coded and decoded. The Register files and the memory of the two simulators matched exactly at the end of tests.

<table>
<thead>
<tr>
<th>Simulation Time [sec]</th>
<th>N. of measured Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIFS</td>
<td>516</td>
</tr>
<tr>
<td>C6XSIM</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>20414689</td>
</tr>
<tr>
<td></td>
<td>20414725</td>
</tr>
</tbody>
</table>

Table 1. H.263+ Encoder Test results
Table 2. G.723 CoDec Test results

<table>
<thead>
<tr>
<th>Simulation Time [sec]</th>
<th>N. of measured Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIFS</td>
<td>60</td>
</tr>
<tr>
<td>C6XSIM</td>
<td>27</td>
</tr>
</tbody>
</table>

It should be noted the performance improvement of the three-dimensional pipeline status approach. The large number of I/O operation in both tests decreases performance. For Computational Intensive with few I/O tasks the speed-up factor is about 10 times.

5 Conclusions
The simulation environment described in this paper allows flexibility in VLIW architectures simulation and good performance at the same time.
The main purpose is to select, for a given multimedia application, the right VLIW architecture (instruction set, cache configuration, pipeline deep) reaching the best match between available and achievable ILP.

References


[10] TI, TMS320C64x Technical Overview, September 2000


[16] ITU-T Recommendation G.723.1, “Dual rate speech coder for multimedia communication transmitting at 5.3 and 6.3 kbit/s”, October 1995