ARMSim: An Instruction-Set Simulator for the ARM processor

Alpa Shah [ajs248@cs.columbia.edu] - Columbia University

Abstract

A hardware simulator is a piece of software that emulates specific hardware devices, enabling execution of software that is written and compiled for those devices, on alternate systems. This paper describes a simulator for the ARM processor, which is widely used in embedded devices like PDAs, cellular phones etc. ARMSim is a lightweight ISA (Instruction Set Architecture) level simulator and a trace generator too. It has some optimizations at the decoder level to improve performance.

1. Introduction

Simulator or Virtual machine technology is an integral part of many computing systems today. This technology is incredibly useful since it makes it possible for users and analyzers to test and execute software well before the actual hardware is available, or in absence of the actual hardware. Some benefits of using simulators are:

- Simulators are flexible and thus new features or components can easily be added.
- It is possible to model anything, including something that might not be possible to do on the hardware.
- It allows stress testing of programs like operating systems by simulating complex interrupt and exception conditions.
- Since simulators are built in software, they are more deterministic. The deterministic behavior of simulators makes programs execution reproducible, and thus helps in locating problems.

Primary applications of simulators consist of computer architecture studies and performance tuning of compiled software, and the compilation process itself. Various types of simulators exist, each addressing different aspects like clock cycle rate, modeling the microprocessor chip logic, modeling the program execution environment, etc. This paper concentrates on an instruction set simulator [ISS] implementation for the ARM processor[1], the ARMSim system. Hereafter, the machines that are simulated will be referred to as *target machines*, and the system on which the simulator is actually run is referred to as the *host machine*.

The remainder of the paper is organized as follows: Section 2 discusses different simulation strategies, and then discusses ISS strategies in detail. Existing related work on simulators is discussed in Section 3. It also describes the ideas and design principles behind ARMSim. The structure and architecture of ARMSim is presented in Section 4. Section 5 describes the realization of the design, and Section 6 discusses some possible extensions and concludes the paper.

2. Simulation Strategies

The best simulation method depends on the application of the simulation results. This section outlines several simulation strategies and their applications.

- ***Architectural level Simulation:***
  Logic designers build *Architectural simulators* to express and test new designs. These allow emulation of the different parts of a processor, using either the simple core, or the core and the data caches and other components. These are generally not intended for executing target system binaries on an alternate platform, but rather to allow research into the modification of the internal datapaths of the processor.

- ***Direct Execution:***
  Target machine binaries can be executed natively on the simulator host processor by encasing the program in an environment that makes it execute as though it were on the simulated system. This technique requires that either the host system have the same instruction set as the target, or that the program be recompiled for the host architecture. Instructions that cannot execute directly

on the host are replaced with procedure calls to simulator code. This method is also known as Dynamic Recompilation [*Dynarecs*]. Native execution of the recompiled code leads to a much faster execution of the simulated software, but they have lengthy context switching, i.e. when the host processor has to switch to target processes. This can make the simulation slower. Most direct execution simulators statically inspect and translate all instructions before simulation begins.

- ***Threaded Code[13]:***
  This is a simulation technique where each op-code in the target machine instruction set is mapped to the address of some (lower level) code in the simulator system, to perform the appropriate operation. This can be implemented efficiently in machine code on most processors by simply performing an indirect jump to the address, which is the next instruction. This method does not suffer from lengthy context switching.

- ***Instruction set simulators***:
  Instruction set simulators [ISS] execute target machine programs by simulating the effects of each instruction on a target machine, one instruction at a time. Instruction set simulators are attractive for their flexibility: they can in principle, model any computer, gather any statistics, and run any program that the target architecture would run. They easily serve as backend systems for traditional debuggers as well as architecture design tools such as cache simulators. A lot of temporal debuggers have recently started using ISS. An ISS can dispatch instructions by fetching from a simulated memory, isolating the operation code fields, and also branching, based on the values of these fields. Once dispatched, reading and manipulation of variables that represent the target system's state are used to simulate the instruction's semantics.

  When CPU architects design a new machine they typically write an instruction-level simulator to test their ideas. These simulators are written to test concepts and processor design tradeoffs; flexibility is important and speed is not of primary importance. They often gather statistics, and this constrains and slows down the simulator. ISS are also referred as "complete system instruction set simulator," as well as "completer system simulator." Furthermore, complete system simulators can be designed to be fully deterministic. They effectively address two major problems in real-time analysis: viz. difficulty in reproducing the effects of program execution, and time distortion resulting from intrusion. Since simulators are implemented fully in software, they can be fully tailored for specific uses. ISS may not address issues like the delays from devices such as disk subsystems, and the memory hierarchy, including the memory management unit, caches and data-buses. Timing accuracy in ISS is of less importance than predictability preciseness, and determinism. An artificial system has few unpredictable factors. Thus, a simulated system starting execution in a known state will always proceed along the same path. This is useful for experiments and debugging purposes. A user can detect misbehaving blocks of code that take more than expected time, and thus restart the process to examine the offending blocks in greater detail and thus trace the execution.

  Several techniques can improve the performance of Instruction set simulators. Instead of decoding the operation fields each time an instruction is executed, the instruction is translated once into a form that is faster to execute. This idea has been used in a variety of simulators for a number of applications. Several efficient memory models have also been proposed.

3. Existing Work

There has been a lot of research work on software simulation of processors. These generally employ the above-mentioned strategies in addition to some new methods. This section describes some popular simulators for different architectures, and also specific ones for ARM. The ARMSim system is compared with existing ARM-specific simulators.

- *SimOS[9]*: SimOS takes a promising approach that handles both user and kernel code by dynamically translating target instructions into short sequences of host-native code. Unlike most direct execution simulators, SimOS keeps only a small part of the target machine's state in host registers and so is able to multiplex between target processors quickly.
- *Shade[2]*: Shade simulates SPARC processor by using dynamic compilation.
- *Talisman[8]:* Talisman is a simulator for multicomputers based on the threaded-code strategy.
- *Mint[10]*: Mint is a simulator for MIPS R3000.

The following are existing simulators developed for the ARM processor, using some of the above-mentioned techniques:

- *ARMPhetmaine[12]*: this simulator is based on the direct compilation technique.
- *tARMac*: also based on the same method.
- *SWARM[11]*: initially designed as an ARM module to plug into the SimOS system developed at Stanford University. It is now is an independent ARM simulator. It implements the core and a small amount of internal co-processors at a basic level, and provides full support for registers, memory caches, and the external memory hierarchy.
- *SimARM[15]*: an instruction set simulator [ISS]. ARMulator[14] is another ISS with a slight variation: it ensures identical cycle-count for instructions. This means that instructions take the same number of cycles to execute as if run on real ARM hardware. This is important for precise simulation since compilers can optimize code that takes advantage of the cycle-counts of specific instructions.

3.1 The ARMSim Approach:

ARMSim is an instruction level simulator for the ARM processor that allows tracing of program execution. ARMSim simulates the entire ISA except for the non-standardized co-processor instructions that vary widely from system to system depending on the actual co-processor module available on the chip. ARMSim is used to execute ARM binaries on a different target machine, and gather some statistics based on the execution of the binaries. It executes one instruction at a time and updates the processor state accordingly.

4. Structure of ARMSim

ARMSim is a simulator of the 32-bit ARM RISC processor[1]. ARMSim simulates the entire instruction set except for those requiring use of the co-processor unit. The co-processor is not a functional unit standard across all ARM processors, and it was not possible to come up with a common subset of operations that all co-processors would support. Figure (a) illustrates the processor of the ARM as simulated by ARMSim. The following are specific functional features of ARMSim:

- *System binaries:* ARMSim takes in binaries for ARM processor for simulation, instead of assembly code. Thus, code compiled for the target processor can be directly executed.
- *Binary data representation:* ARMSim treats data as being stored in the *big-endian* representation.
- *Determinism:* The program behavior must be repeating for testing purposes. Being completely software-simulated, ARMSim guarantees that the outcome of program behavior is deterministic.
- *Low startup:* Being an instruction level simulator, the system has no initial startup time, and no preprocessing is required as in the case of direct compilations or threaded-code systems.
- *Extensible:* The implementation of the system is based on a modular design, and this makes it possible to extend the simulator in order to support other modules like I/O devices, and complex memory hierarchies, if needed.
- *Statistics:* ARMSim has been designed to support tracing the execution of a program and gather statistics. ARMSim can be used to collect information like frequency of data memory accesses, the

number of branches that are taken, and other analyses on program behavior based on instruction profiling.

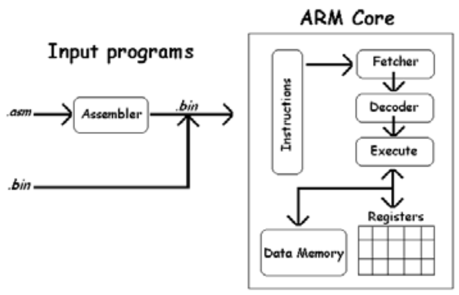

Fig. (a)

4.1 Modeling Processor

ARMSim is a *behavioral modeling* of the ARM processor. This aspect made possible numerous abstractions from the underlying hardware, e.g. simplified memory replacing the complete memory hierarchy subsystem, unification of all the individual functional units in the CPU, etc. The ARM processor elements simulated in ARMSim are program and data memory, the status register, CPSR, the ALU, barrel shifter, conditional execution of instructions, general purpose registers. The state of the system is defined by the values in the registers and memory.

4.2 Modeling Basic Instruction execution

ARMSim simulates program execution by iterating through a cycle of instruction fetching, decoding and execution.

4.2.1 *Fetch*

Just like in any microprocessor[6], ARMSim internally maintains a program counter to determine the next instruction to be fetched from the instruction memory. This program counter is updated automatically upon each instruction fetch, and can also be explicitly updated by instructions.

4.2.2 *Decode*

The class of each fetched instruction is first identified, and then, the different fields such as the opcode, operands, of the instruction are decoded according to the type of the instruction. ARMSim achieves some optimization based on the decoding step – each instruction needs to be decoded only once, regardless of the number of times it is fetched for execution. This technique is called *decode-once-execute-many-times*.

4.2.3 *Execute*

The execute phase of the iteration involves computation of memory addresses (to get the operands), if necessary, and the actual operation of the current instruction. This operation updates the state of the system as determined by the instruction type.

4.3 Memory model

ARMSim does not explicitly model the memory hierarchy of the real ARM processor. Being an instruction-level simulator, ARMSim can avoid having to accurately emulate movement of data between the memory, the caches and the processor, and instead can represent all memory-accesses as being uniform. Instruction memory is stored as a linear array of 21-bit values, as shown in figure (b).

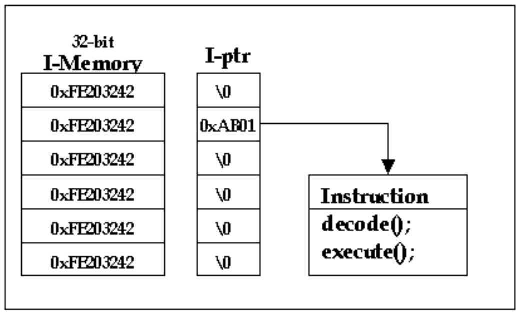

Fig. (b)

4.3.1 *Data Memory*

The ARM supports data transfer in 3 different sizes: byte, halfword, and word between the registers and memory. ARMSim implements the transfer of memory data of all the sizes while maintaining the *endian*-ness and *signed*-ness of the data.

4.3.2 *Program Memory*

ARMSim does not simulate the ARM processor in the 16-bit *THUMB* mode. This has the effect that all instructions are stored at word-aligned instructions, and all instructions fetches operate as 32-bit data transfers.

4.4 Processor State modeling

Of the various modes of operations possible on the ARM, the most interesting one is the *user* mode. Most of the running time of programs is spent executing instructions in this mode. ARMSim maintains the exact set of on-chip registers as are visible to instructions executing in the *user* mode on the ARM processor. Out of these registers, the *CPSR* register is updated based on the result of instruction execution, while the data in the 16 general purpose registers can be explicitly set by the instructions.

4.5 Statistics gathering

ARMSim allows tracing through the execution of the programs. Some of the information that ARMSim can record are:

- Instruction address, effective address
- Decoded opcode value
- Number of branches taken
- Number of memory references

5. Realization of the ARMSim system

In this section of the paper, I describe the various system design decisions I have made while building ARMSim. An ISS is by nature much slower at interpreting programs than running the program natively on the intended hardware, or even dynamic recompiling simulators that can optimize the interpretation by actually translating certain parts of the target system program into native code for the host system that the simulator itself is running on. However, this sort of translation decreases the amount of control on the simulation – such a thing would prevent any extensive form of behavioral analysis of execution. For example, it would be impossible to count the number of times a certain block of code in the target program has been executed, especially if the original target code is significantly different from its translated version. Moreover, with this sort of simulation, it is hard to maintain uniformity of conditions among different host system architectures. ARMSim is a fully interpretative simulator, and it incorporates some optimizations that help speed up the overall execution time of target programs portably across different host systems.

5.1. ARMCore: a unified CPU: ALU, barrel shifter

ARMSim interprets the execution of all target system instructions thus obviating the need for separate functional units.

5.2. ARMCore: CPSR and general purpose registers

The 17 general purpose and CPSR registers visible in the *user* mode in the ARM processor are implemented in ARMSim.

5.3. Instruction identification, decoding

ARMSim identifies the type of each fetched instruction by performing a bitwise comparison of the 32-bit instruction with a predefined set of masks. It then creates an instance of a subclass of the *Instruction* class that will automatically decode the remaining different fields of the instruction such as the operands, immediate values, and condition codes.

5.4. Decode-once-execute-many-times

Compiler-generated executable programs, in general, contain many blocks of code that are iterated numerous times. ARMSim attempts to utilize this fact to achieve an optimization in the simulation – since all 32-bit instructions are decoded and executed in the form of an *Instruction* object, ARMSim will create a single instance for all iterations of the instruction at each memory address. This allows the simulator to avoid having to re-create new instances for the instructions in each iteration of the loop. Instead, after executing the instruction at the given memory location the first time, ARMSim will preserve this instance and reuse it for any future execution of that instruction. Also, ARMSim carries out all processor-state-independent decoding only for the first time that the instruction is executed – these are generally field extractions such as source/destination register identification, and immediate operands evaluation.

5.5. Realization of the memory system abstraction

ARMSim treats the memory system as being uniform, with constant access time, regardless of the address of the memory data being accessed. Instructions are stored at word-aligned addresses. Since executing instructions could conceivably request byte-data transfers at non-aligned addresses, ARMSim permits memory data transfers at non-word-aligned memory locations.

6. Conclusions, Future Extensions

ARMSim is a simulator that is well suited to simulate the overall behavior of execution of programs that are intended for execution on an ARM system. Some important uses for ARMSim would be: comparisons of the quality of compiled code as produced by different compilers and/or compiler options. The modular nature of the implementation of ARMSim makes it easy to:

- Can more closely represent the memory hierarchy without having to rewrite major parts of the system – this can then be used to monitor memory access patterns in test programs such as temporal and spatial locality, size of memory requirement, etc.
- Supplement the system with newer definitions of instructions.
- Incorporate clock cycles-per-instruction [CPI] for each class of instruction; possibly include some mechanism for adjusting the CPI for instructions that cause a cache-miss.
- Provide an "architectural" level garbage collector that will intelligently maintain the instruction pointers for those instructions that are in frequent use, e.g. in a loop, and clear out the rest. A sophisticated version would even perform prefetching and create appropriate instruction pointers before the instructions are actually required.

7. References

[1] ARM7TDMI ARM processors User's Manual, Advanced Risc Machines Ltd.

[2] R. Cmelik, D. Keppel [1994]: "Shade: A Fast Instruction Set Simulator for Execution Profiling," *Proceedings of SIGMETRICS*, ACM, Nashville, TN, 128-137.

[3] R. C. Bedichek: "Some efficient architecture simulation techniques". In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990,Washington, DC, USA*, pages 53–64, Berkeley, CA, USA, Jan. 1990. USENIX.

[4] P. Magnusson and B. Werner: "Efficient memory simulation in SimICS". In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[5] P. S. Magnusson: "Efficient instruction cache simulation and execution profiling with a threaded-code interpreter". In *Proceedings of Winter Simulation Conference 97*, 1997.

[6] John Hennessy and David Patterson. Computer Organization and Design: The Hardware-Software Interface (Appendix A, by James R. Larus), Morgan Kaufman, 1993.

[7] Emmett Witchel, Mendel Rosenblum. *Embra: Fast and Flexible Machine Simulation*. Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Philadelphia, 1996.

[8] Bedicheck, R. 1995. Talisman: Fast and accurate multicomputer simulation. In Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (May), 14±24.

[9] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACMTOMACS*, 1997.

[10] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory. In Proceedings of MASCOTS, pages 201-207, January 1994.

[11] SWARM 0.44 Documentation Michael Dales Department of Computing Science, University of Glasgow,Scotland

[12] The Design of ARMphetamine 2 Julian Brown, University of Bristol

[13] James R. Bell Threaded code. CACM, 16(6). June 1973

[14] The ARMulator; Document number: ARM DAI 0032; Issued 1999; http://www.arm.com/

[15] SimARM; Green Hills Software Inc. http://www.ghs.com/