ARMISS: An Instruction Set Simulator for the ARM Architecture

Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, Ge Yu
Institute of Computer Software and Theory, Northeastern University
mingsong_lv@hotmail.com, dengqx@mail.neu.edu.cn, guannan0609@hotmail.com
ming_xy@163.com, yuge@mail.neu.edu.cn

Abstract

The development efficiency of embedded systems is highly pressured due to the pursuit of short time-to-market of embedded products. In traditional design flow, although software can be developed in parallel with the hardware platform, it can only be tested and verified after the platform is fabricated. ARMISS, an Instruction Set Simulator for the ARM architecture, is developed to enable early software testing and verification. The ARM instruction set, MMU and interrupt handling are emulated in this tool. An instruction caching technique is designed to accelerate the interpretation-based instruction emulation. ARMISS is implemented in the C programming language, thus is highly portable across varies host operating systems for embedded system design.

1. Introduction

Embedded systems are application-centric: typical embedded systems range from highly real-time and dependable machines to small mobile consumer electronic products. For each different product line, both software and hardware are redesigned or revised to meet different requirements on functionality, performance, energy and total cost, which makes the design of embedded systems greatly rely on customized hardware. Additionally, tremendous potential profit leads to fierce market competition, resulting desperate pursuit of short time-to-market. An investigation by Embedded Market Forecasters reported that more than 50% of the investigated projects were finished behind schedule [6]. The tension between customized design and short time-to-market poses great challenge on design efficiency of embedded systems. This situation is even severe with the fast increase of system complexity.

In traditional design flows, systems are partitioned to the software part and the hardware part according to system specifications. Although software can be development in parallel with hardware, the final implementation can only be tested and verified after the hardware platform is prototyped or even fabricated. The lack of real target hardware in the software development process impairs global design efficiency. ARMISS, an Instruction Set Simulator (ISS) for the ARM architecture, is designed to solve this problem.

An ISS [2] is a layer of software residing between applications and operating systems, or between operating systems and real hardware, which enables the upper layer program binary compiled for an Instruction Set Architecture (ISA) to run on the lower layer software/hardware that implements another ISA. The upper and the lower layer ISAs are called source ISA and target ISA respectively. Basically, the simulation is done by statically or dynamically translating each of the instructions of the source ISA into a set of instructions of the target ISA that reproduce the behavior of the source instruction. Figure 1 depicts the two different system architectures.

The position of ARMISS in the system is illustrated by the shaded area in Figure 1(a). A given application compiled for the ARM instruction set is translated by ARMISS into a set of IA-32 instructions that can run on Windows or Linux which runs on an IA-32 microprocessor. The application functions the same as if it runs on a real ARM processor, since ARMISS exactly emulates the functionality of the ARM instruction set. With the support of such an ISS, software designed for an ARM processor can be tested and verified even if the real hardware platform is not ready, and the overall design cycle is shortened. This is the design motivation of ARMISS.

The rest of this paper is organized as follows. Section 2 is dedicated to the emulation of the core ARM instruction set. The simulations of MMU and interrupt handling are discussed in Section 3 and 4 respectively. Section 5 gives the experiment results, and related work is given in Section 6. The paper is concluded in Section 7.

2. Emulation of the ARM Instruction Set

A complete ISA implements the management of processor resource, memory resource, I/O resource and inter-
Figure 1. The position of an ISS in the system

rupts. We dedicate this section to introduce the emulation of processor resources. Two implementation techniques are widely adopted, namely interpretation and binary translation [10] which vary a lot in performance, memory requirements and portability. We will give the reasons of choosing interpretation in our implementation. Since the major drawback of the interpretation technique is low performance, an instruction caching technique is designed to improve the interpretation efficiency.

2.1. Instruction Set Emulation

Instruction set emulation is an important enabling technology for virtualization [14], because the virtual machines must support a program binary compiled for an instruction set that is different from the one implemented by the host processors. Across the whole paper, we call the instruction set to be emulated the source instruction set, and call the real system instruction set the target instruction set. Thus emulation allows a machine implementing the target instruction set to reproduce the behavior of the software compiled to the source instruction set. ARMISS is not part of a virtual machine, and is implemented as a stand-alone hardware simulator.

Notice that the definitions of emulation and simulation are different. We restrict the definition of emulation by instruction set translation that maintains functional correctness of the processor resource. The objective of simulation is to study the process of performing computation. In addition to emulating a program, a simulator may as well model the internal behaviors of a processor or a memory system.

Two major techniques are widely adopted in instruction set emulation: interpretation and binary translation. The software that performs the interpretation is called an interpreter. Before interpretation of a program, the interpreter will construct an image of the source memory and a data structure called source context block that contains all the registers of the source ISA to be emulated. When emulation starts, the interpreter fetches the first instruction from the source binary; then extracted the opcode from the instruction; according to the opcode, the interpreter will dispatch a specific function that emulates the fetched instruction by performing computation and assigning correct values to the source context block and the source memory. The interpreter iterates all the instructions of the source binary, and performs the "fetch-decode-dispatch-execute" procedure for each instruction. The interpretation flow is illustrated in Figure 2(a).

Figure 2. Emulation flow of basic interpretation and binary translation

Binary translation works in a philosophy different from interpretation. The basic idea of binary translation is: the translator first transforms the source binary to the target binary code compiled to the target ISA, and then executes the target code. In binary translation, the generation of target code is optimized by a direct mapping of target ISA registers to source ISA registers. The mapping eliminates the source context block data structure. It is apparent that binary translation is directly implemented in assembly codes due to register mapping. Figure 2(b) illustrates the work flow of binary translation.

2.2. Performance Analysis and Selection of Implementation Technique

Due to different design philosophy, interpretation and binary translation vary a lot in many performance aspects. In this section, we first analyze the performance characteristics of basic interpretation and binary translation, and then give our reasons of choosing interpretation as our implementation technique. The characteristics of the two techniques are listed in Table 1.

The runtime performance of interpretation is slow, since the "fetch-decode-dispatch-execute" step incurs too many runtime decisions and the memory access to the source context block leads to further performance overhead. The runtime performance of binary translation is fast because the translated target code is highly optimized, and the use of register mapping reduces memory access. Since there is no intermediate form in interpretation, the memory usage is low; but for binary translation, the translated target code may consume considerable memory. Interpreters are usually implemented in high level languages such as C, so they
Table 1. A comparison of basic interpretation and binary translation

<table>
<thead>
<tr>
<th>Features</th>
<th>Interpretation</th>
<th>Binary Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Speed</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Memory Usage</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Portability</td>
<td>Good</td>
<td>Poor</td>
</tr>
</tbody>
</table>

are easily ported to different target ISAs, and the overhead is just a re-configuration of the interpreter. Binary translation has a poor portability because assembly coding and register mapping are specific to the target ISA, thus the effort of porting a binary translator from one ISA to another is roughly equivalent to redesign of the translator.

There is an apparent trade-off between runtime performance and memory usage: a faster implementation usually requires larger memory. Consider that we aim at running ARMISS on mainstream desktop computers featured by 2.6-3GHz processors and 1-2 GB main memory, both the runtime overhead and the memory overhead are tolerable. The major consideration in the design of ARMISS is code portability, since the ISS may run on Pentium computers, PowerPC computers such as Mac, or even Sun Sparc workstations. The host computer for system development may vary in instruction set architecture due to the versatility of development platforms available to different developers.

2.3. Implementation of Emulation

The emulation of the ARM instruction set is partitioned into several design steps. First the source context block is modeled by a data structure and different working modes of an ARM processor is supported by the context block. Second, the control flow of the interpreter is constructed, and a two-stage decoding process is designed according to the ARM instruction format. Third, an instruction cache is designed and embedded in the control flow to improve interpretation efficiency.

2.3.1 The Source Context Block for the ARM Instruction Set

ARM based processors are widely adopted in various embedded systems. The ARM architecture supports seven working modes [5], listed in Table 2, to satisfy the special requirements of embedded real-time systems. The latter five modes are called exceptional modes. Switching of working modes can be triggered by either external interrupts or software.

The ARM instruction set defines 37 registers which are all 32-bit wide. There are 31 general purpose registers including a program counter, and 6 program status registers including one CPSR. The lower most 5 bits of CPSR indicates the current working mode of the processor. A data structure called CPUState is implemented to model all the 37 registers and the processor’s working modes. CPUState serves as the source context block of the interpreter, which is illustrated in Figure 3.

```c
struct _CPUState {
    uint32_t regs[16];      // R0-R15 of SYS and USR mode
    uint32_t r13[5];        // R13 of 5 exception modes
    uint32_t r14[5];        // R14 of 5 exception modes
    uint32_t fiq_reg[5];    // R8-R12 of FIQ mode
    uint32_t cpsr;          // CPSR
    uint32_t spsr[5];    // SPSR for 5 exception modes
    uint32_t mode;         // Processor working modes
}
```

Figure 3. CPUState - the source context block

The execution of instructions changes the values hold in the 37 registers, which is reflected by changing the corresponding field of CPUState. Notice that a switching of working mode will result in value changes both in the corresponding segment of cpsr and the value of mode.

2.3.2 Instruction Emulation Work Flow

The main work flow of instruction emulation is a loop of “fetch-decode-dispatch-execute” steps. A data structure is defined to store all the information extracted from the current source instruction. This data structure is accessed by all the four functions in the above step, and the values of the data structure changes when a new instruction is decoded. The definition of the data structure is illustrated in Figure 4.

Notice that when the processor works at different modes, registers used in the assembly code, for example R13, actually point to different physical registers. So we use pointers in this data structure and point them to the right register in CPUState according to the processor’s current working mode. The overall work flow is illustrated in figure 5(a).

Since ARMISS is an instruction set level simulator, it can...
only provide instruction level precision up till now. Behavior of the instruction pipeline is not modeled in ARMISS. We leave pipeline simulation for future work which can enable ARMISS to model the processor with better precision.

typedef struct _instruction {
    uint32_t ins; // opcode
    uint32_t de1, de2; // results of the two-stage decoding
    uint32_t* rd; // point to corresponding registers
    uint32_t* rm; // point to corresponding registers
    uint32_t* rs; // point to corresponding registers
    uint32_t* rn; // point to corresponding registers
    uint32_t imm; // immediate number
    void (*exec)(); // point to the emulation routine
    uint32_t (*test_cond)(); // point to the test function
} Instruction;

Figure 4. Instruction - the data structure to store decoded data

2.3.3 Instruction Decoding

The ARM instruction format [5] is illustrated in Figure 6. The instruction decoding is performed in three steps. First, the highest 4 bits of an instruction, namely the condition, is checked to see if this instruction can be executed. If the condition is false, this instruction will not be further decoded and executed. If the condition is true, the instruction is decoded in two stages. In the first stage 26-27 bits of the instruction are analyzed to decide the type of the instruction. In the second stage, 21-24 bits of the instruction are analyzed to decide the exact opcode for a given type. The results of the two-stage analysis are stored in the ed1 and ed2 field of Instruction respectively. If instruction caching is implemented, cache will be searched immediately after an instruction is fetched, a cache hit will eliminate the following decoding functions.

Figure 5. Interpretation work flow

2.3.4 Instruction Caching

Figure 5(a) shows that each source instruction is sequentially fetched, decoded, dispatched and executed. Experiments show that decoding and execution are the two major performance bottlenecks. Since we aim at code portability, as stated in Section 2.2, there is little possibility to optimize execution performance. So we target at optimizing the performance of instruction decoding. The principle of program locality tells us that some segments of the program are visited iteratively both spatially and temporally. Caching techniques are widely used in computer science, directly implementing the principle of locality, and is also adopted in the design of ARMISS.

The main work flow of Figur 5(a) is revised to Figure 5(b). Whenever an instruction is fetched from the binary, a cache lookup is performed to see if this instruction has been decoded and cached. If cache hit returns, then the decoded information is loaded from the cache; otherwise, the instruction is decoded normally, and the new decoded instruction is put into the cache in case that it may be revisited in the near future.

There are two major issues in the cache organization: mapping policy and cache update policy. Mapping policy tells how to establish a relationship between the instruction and the cache entry. In the design of ARMISS, a table is created for each binary to store the mapping information. The table size is equivalent to the number of instructions in this binary. Each time an instruction is fetched, the program counter uniquely indicates the location of the instruction. According to the program counter, the entry of the mapping table is immediately decided, which trades table size for mapping efficiency. There is an integer column in each table indicating whether the corresponding instruction is cached and the times the instruction is visited. A zero
value indicates no caching, and an integer greater than zero indicate the number of visits to the instruction. Given a fixed cache size, an update policy is required. In the design of ARM ISS, the least used instructions in the cache are replaced by the new instruction.

The cache design introduces memory overhead. If 1K instructions are to be cached, then the total cache size should be 40KB, since an Instruction data structure (Figure 4) takes 40 bytes. The memory overhead is proportional to the number of cached instructions, and this overhead is inevitable.

3. MMU Simulation

Memory Management Units (MMU) are currently commonly adopted in main stream embedded processors. The main functionality of an MMU is to support virtual addresses for programs and provide memory access control. The simulation of MMU in the design of ARM ISS comprises three major functionalities: mapping from virtual address to physical address, memory access control, and manipulation of access failures. The CP15 [5] co-processor in the ARM architecture is the MMU which has ten registers that supports the above three functionalities. A data structure CP15 is defined to store the values of the registers. All the simulations of MMU can be viewed as such processes that modify the CP15 data structure and perform the control rules defined in the ARM architecture according to the values of these registers.

3.1. Address Mapping

Address mapping is the process that transforms a virtual address into a physical address according to the addressing rules defined in the ARM architecture. Page table is the data structure that enables address mapping. Each process maintains a page table that is stored in the main memory and manipulated by CP15. To make it simple, the organization of a page table is like a spreadsheet sheet with each row, called an entry, a mapping of a virtual address to a physical address.

The ARM architecture supports a two-level mapping mechanism. The physical memory is first divided into 1M-size segments which are further divided into pages serving as the basic blocks of memory management. The ARM architecture supports large pages, small pages and tiny pages, with the size of 64K, 4K and 1K respectively. The detailed two-level page architecture supported by the ARM MMU is illustrated in Figure 7, where large pages are assumed to be implemented. The mapping of small pages and tiny pages are similar.

The simulation of address mapping is quite straightforward: for a given virtual address, the corresponding physical address is calculated and the source memory state maintained by the ISS is accessed by the real physical address.

If the access is a write operation, then the value of the corresponding address in the source memory state is modified. Implementation details are omitted here for the purpose of brevity.

![Figure 7. Address mapping mechanism](image)

3.2. Memory Access Control

When an instruction is to access a memory page, the performed operation of the instruction should conform to the access rules indicated in the values of the corresponding registers of CP15. The segments that control memory access are: AP segments defined in the page table descriptors for each memory page, and the S and R segments of register C1. The combination of values of AP, S, and R gives the access rules for system mode and user mode, which is illustrated in Table 3.

<table>
<thead>
<tr>
<th>AP</th>
<th>S</th>
<th>R</th>
<th>Privileged Access</th>
<th>User Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>No Access</td>
<td>No Access</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>Read Only</td>
<td>Read Only</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>Read Only</td>
<td>Read Only</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>01</td>
<td>X</td>
<td>X</td>
<td>Read/Write</td>
<td>No Access</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>X</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>X</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

When incorrect memory accesses occur, such as attempting to write to a read-only memory page, a memory access failure is generated. CPU will be informed as long as an access failure is detected, and the accessing information is stored in some specific registers for further failure manipulation. External devices can also report memory access failure via a pin in the ARM processors. Memory access
failure generates interrupts that can be handled to do post-error analysis and failure recovery.

Internal memory access failure manipulation is simulated in ARMISS, and simulation of failures triggered by external devices is left for future work. Four types of access failures are simulated in ARMISS, namely address alignment failure, address transformation failure, domain control failure and access privilege failure respectively. C5 and C6 registers of CP15 are involved in the manipulation of access failure. The values of C5 and C6 are modified according to the defined format of ARM architecture to log the exact failure. It is possible that access failures resulted by different reasons may occur simultaneously, and ARMISS can cope with such situations. The registers’ values are updated according to priority levels, which have been defined in the ARM architecture.

```c
void enter_exception_swi() {
    CPUState.spsr[1] = CPUState.cpsr;
    switch_mode(SVC);
    IRQ_OFF; CPUState.regs[15] = 0x08;
}

void enter_exception_instabt() {
    CPUState.spsr[2] = CPUState.cpsr;
    switch_mode(ABT);
    IRQ_OFF;
    CPUState.regs[15] = 0x0c;
}
```

Figure 8. Simplified manipulations of software interrupts

4. Interrupt and Exception Handling Simulation

In the ARM architecture, there are three program execution flows:

- Normally a program executes sequentially, and at the end of each instruction the value of PC register will be added by 4 to point to the next instruction;
- An explicit jump instruction may result in jumping to a specified address or the entry of a sub routine;
- When interrupts occurs, the program jumps to the corresponding interrupt handler, and after the execution of the interrupt handler, the program should jump to the next instruction before which the program is interrupted.

The arm interruption system defines seven types of interrupts: Reset, Undefined instruction, Software interrupt, Prefetch abort, Data abort, IRQ, and FIQ. The first four types of interrupts are supported in ARMISS; while the latter three types are triggered by specific processor pins and are not supported in ARMISS. Interrupt priority levels are defined in the ARM architecture, interested readers can refer [5] for further details. The simulation of response to interrupts is performed in the following steps:

- Save current processor state: store the value of the CPSR in SPSR of the corresponding exception mode;
- Set the corresponding bits in the CPSR for further execution;
- Forbid IRQ/FIQ;
- Put the return address to the register lr_mode, and set the value of PC to the corresponding interrupt vector.

The simplified manipulations of software interrupts and prefetch abort are illustrated in Figure 8.

5. Experiment Results

ARMISS is ported to Windows and Linux that run on a desktop computer featured by a Pentium 4 CPU running at 2.8GHz and 1GB main memory. Experiments are performed to check the functional correctness of ARMISS and the performance of improved interpreter that implements the instruction caching technique.

To check whether the simulator functions correctly, a boot loader is ported to run on ARMISS. Since I/O simulation is not implemented in ARMISS, some modifications are made to the boot loader, which redirects data that are outputted via serial port to the console. The runtime screenshot is illustrated in Figure 9. Experiments show that the boot loader successfully boots on ARMISS.

Figure 9. Runtime screen shot of ARMISS

To test the performance of the improved interpreter, we load four common applications of embedded control sys-
tems in ARMISS: FFT_64, FFT_32, Bubblesort and Integral. The numbers of instructions of these four applications are 662899, 682764, 378840, and 569416 respectively. The run time of each application are illustrated in Figure 10. The speedups of the applications range from 4 to 8, with FFT_64 achieving the highest speedup. It demonstrates that the instruction caching technique implemented in the ISA emulator can significantly improve simulation performance.

Figure 10. Run time performance comparison of interpretation with/without caching

6. Related Work

Instruction Set Simulation technique falls into the virtualization technology which is brought forward in 1960’s in IBM System/360 Model 67 [12] and is re-vitalized with the emergence of multi-core platforms [13]. System virtualization can be classified into 3 types: application level virtualization, high-level language virtualization, and operating system virtualization. Instruction set emulation is the enabling technology for virtualization.

MPARM [3] is a project jointly developed by University of Bologna, Stanford University and 14 other universities. MPARM is a multi-processor cycle-accurate architectural simulator. Its purpose is the system-level analysis of design tradeoffs in the usage of different processors, interconnects, memory hierarchies and other devices. MPARM output includes accurate profiling of system performance, execution traces, signal waveforms, and, for many modules, power estimation.

QEMU [8] is an open source processor emulator, which is widely used in system design. QEMU can achieve very high performance since it implements the dynamic binary translation technique. QEMU can emulate x86, PowerPC, ARM and Sparc instruction sets on x86, PowerPC, ARM, Sparc, Alpha and MIPS host processors. QEMU is a full system emulator in which a complete operating system is run in a virtual machine, which requires no modification of the guest operating system.

Bochs [1] is an x86 instruction set emulator which can run on many ISAs including IA-32, PowerPC, Alpha, Sparc and MIPS. Bochs emulates not only the processor but also a whole system with I/O devices such as keyboard, mouse, video card and NIC. Now Bochs can simulate from the old i386 processors to the IA-64 instruction set. MMX and 3DNow co-processing instruction sets are also supported by Bochs. This emulator can run on Windows and almost all the distribution of the Linux operating system.

VMware workstation [4] and Xen [7] are two dominating virtual machines for operating system virtualization. VMware can be installed as an application on a host system and supports full system virtualization for many popular operating systems. Binary translation technique is used in VMware, and the limitation is VMware only supports x86 instruction set. Xen is also a commercial virtualization tool, which is designed using the paravirtualization technology. Paravirtualization can achieve more than 90% percent of the performance of a system running on bare machine. The drawback of paravirtualization is that it requires modification to the guest operating system.

Other virtualization products, such as the Java Virtual Machine [11] and the Microsoft .NET framework [9], are also popular in the software development processes. Language level virtual machines target at providing better portability to the high level language source codes. Interested readers can refer to the corresponding references for further details.

7. Conclusion

In this paper, an instruction set simulator ARMISS is designed that comprises three major parts: the emulation of core ARM instruction set, the simulation of MMU behavior, and the simulation of ARM interrupt manipulation. An instruction caching technique is implemented to improve the performance of interpretation-based emulation of instructions. Experiment results show that the improved interpretation performs at least 4 times faster than basic interpretation. ARMISS is implemented in the C programming Language, so can be easily ported to various operating systems. Adopting ARMISS in the embedded system design can improve the efficiency of software development.

Future work to improve ARMISS includes three major jobs: adding support to the THUMB instruction set; modeling cache behavior in the MMU simulation; adding simulation of I/O devices. These jobs will lead ARMISS to a whole-system simulator. A framework will be further developed to enable the users to customize a system by adding or deleting hardware components in a plug-and-play manner.

References