AN ABSTRACT OF THE THESIS OF

Balaji Megarajan for the degree of Master of Science in

Electrical and Computer Engineering presented on April 6, 2004.

Title:Enhancing and Profiling the AE32000 Cycle Accurate Embedded Processor Simulator

Redacted for Privacy

Abstract approved: _____

Ben Lee

The AE32000 processor core, developed by Advanced Digital Chips Inc., Korea, is used primarily in the embedded processing environment. The AE32000 simulator models this embedded processor core having high code density. An enhanced simulator was developed to study the performance of the present Instruction Set Architecture after comparison with the Simplescalar ARM simulator. ARM is among the most widely used processor cores for embedded applications and so was chosen for this comparison. Code density of the AE32000 is very high because of its shorter instruction length. This results in a smaller footprint inside the memory. But the longer instruction length of the ARM proves better when it comes to performance. The LERI(Load Extension Register Immediate) unit of the AE32000 has a special role before instructions that need long immediate values during execution.

Enhancing and Profiling the AE32000 Cycle Accurate Embedded Processor Simulator

by

Balaji Megarajan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented April 6, 2004
Commencement June 2004

Master of Science thesis of Balaji Megarajan presented on April 6, 2004

APPROVED:

Redacted for Privacy

Major Professor, Electrical and Computer Engineering

Redacted for Privacy

Associate Director of the School of Electrical Engineering and Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request. Redacted for Privacy

Balaji Megarajan, Author

ACKNOWLEDGMENTS

I would like to sincerely appericiate the help from my family, that has helped me reach where I am now. Without their moral and financial support, I would not have been writing this.

I would like to thank and greatly appereciate Dr. Ben Lee, my major advisor for all the patience and time he invested for making this thesis successful. I am grateful to Dr. Lee, for his continuous encouragement and support throughout my masters program. His availability to answer my questions during all times of the day was extremely helpful. I also like to thank Dr. Alexandre Tenca, Dr. Roger Traylor and Dr. William Hetherington for serving on my defense committee.

I also want to thank John Mark, Savi and Kim Hyun Gyu for answering so many of my questions. It is always friends that make you remember every place that you have been. I thank, my friends Saket, Harish, Krishna, Siva, Thirumal, Weetit, Chen, Haisa, Hwang, Bajaj, Mahesh and Bhai for giving me the many good times I had in Corvallis and also many other friends.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

ENHANCING AND PROFILING THE AE32000 CYCLE ACCURATE EMBEDDED PROCESSOR SIMULATOR

1. INTRODUCTION

1.1. Background

Today an average American household could have around 40 embedded processors [1], together with 5 to 10 in each computer and another dozen inside every car. The same household would have only one or two microprocessors inside their PC's. We can find embedded processors inside entertainment centers, washing machines, cars, microwaves, remote controls, cell phones etc. As we start to demand more from these every day appliances and other accessories there is a grater need for better products and more research in this field.

A microprocessor would be just one component among many in a system containing RAM, EEPROM, analog and digital I/O, etc., whereas the embedded microprocessor would form the core of a micro-controller and all the above features can be found inside the micro-controller. The micro-controllers are easier to program, but the instruction set of a micro-controller would not be as powerful as a processor because of space and cost constraints. It is the application that would decide on whether to use a microprocessor or an embedded processor. A micro-controller can be used in applications that do not require large computation capabilities.

1.2. Motivation

The motivation for this work was the prototyping of the AE32000 core which will help in better understanding the important aspects of the core before the actual die is made. The aim is to understand and suggest improvements to the AE32000 instruction set architecture(ISA). This ISA was designed with the code size reduction as its main focus. By enhancing this ISA we hope to find wider applications for the AE32000 architecture.

One of the important constraints towards the use of an embedded microprocessor based system is the size of the memory. The memory requirements will affect both the size of the die and also the power spent to read, write and maintain the memory. The size of memory would also affect the latency. The trend is towards increasing the size of the ISA for new embedded processors. The problem with a 32-bit RISC core would be density of the code inside memory. The AE32000 [2] focuses on this very problem of code density. The approach used is similar to the ARM Thumb [3] and the MIPS 16 [4]. One of issues with a shorter instruction length is the length of the immediate operands that can be used inside the instruction. The AE32000 approaches this by using an independent (LERI)Load Extension Register Immediate unit which will help in the number of accesses to the memory.

The following chapters will describe the method used with the results being presented at the end. Chapter 2 has the Literature Review, Chapter 3 will describe the simulation models and the enhancements to the AE32000 core. Chapter 4 has the results, finally followed by the Chapter 5 containing the Conclusion and Future Work sections.

2. LITERATURE REVIEW

Embedded Microprocessors are finding ever wider applications, each needing more power and performance at the lowest possible cost. Memory size reduction is one of the main areas of research for the simple reason that when we build an embedded system the memory is on chip with the processor core and the memory size would directly affect the die size and cost. Also the number of accesses to memory would influence power consumption. Our research mainly revolves around this important issue of memory size reduction.

Lee, Beckett and Applebe [5] addressed the issue of memory size and performance in the embedded microprocessor systems. They had designed a new architecture called the Extendable Instruction Set Computer (EISC). It has a fixed 16-bit instruction length with short length offset and immediate fields. They found that the performance of the microprocessor was not only limited by the speed difference between the processor and the memory system but also by the physical properties of the bus connecting the CPU and the memory system. In their design they had an extension flag (the e-flag) which could extend the offset and the immediate values to 32 bits. The authors found that loads and stores occur frequently, but mostly use short length offset addressing, and that frequency of small sized constants is high. In their tests they found that the density of the EISC was 66% higher than the MIPS R3000 for certain benchmark programs. Their code was also found to be 5% to 15% smaller then ARM-7TDMI and the frequency of load and store instructions were about 15% lesser.

Bird and Mudge [9] claimed that, as the bandwidth to the instruction cache can be a limiting factor, code density is the critical factor for embedded systems.

Once the code is generated they analyses the instruction stream for often reused sequences of instructions. These patterns of multiple instructions were then mapped into single byte opcodes and thus they were able to achieve compression of multiple, multi-byte instructions into a single byte. They found that many instruction sequences were different only in the register numbers in the arithmetic instructions and operand offsets for the load and store operations. They checked for such redundant sequences starting from a label and ending at the branch instruction for all instruction sequences inside the code. A comparison was made between their design and PowerPC processors and they suggested that by incorporating 1K ROM in the CPU a programs code size can be reduced by 45% to 60%.

Kwon, Parker and Lee [10] introduced a new architecture TOE (Two Operand Execution) which overcame the performance degradation due to compressed instruction set by explicitly specifying the eligibility for parallel execution. They argued that in the ARM Thumb and MIPS16 the shorter instructions were carefully selected subsets of standard 32-bit instructions sufficient to encode the program to much smaller sizes. They claim up to 70% reduction in code size in certain cases. But the drawback was a 40% increase in the number of instructions which could somewhat offset the advantages gained by shorter instructions. Their TOE architecture could execute the 32-bit instructions and the 16-bit instructions after decompression. With a 1-bit instruction field they were able to execute 2 instructions in parallel, provided they were independent and the functional units were available. As the parallel instructions were specified by the compiler no additional hardware was needed. The shortage of one bit inside the instructions was offset by dividing the registers into multiple groups, and the access to each group would depend on

the type of the instruction. On comparison to the Thumb architecture the authors reported a 36% decrease in the code size and that it was possible to execute 33.4% instructions in parallel.

Wolfe and Chanin [7] delved into the issue of code compression at the instruction cache level. They had found that the difference in the code sizes between RISC and CISC was making designers reluctant to use RISC in the embedded systems. Typically in such a system the cost of the instruction memory can be a major contributing factor towards the overall costs and also result in an increase in power consumption. The advantages from RISC, like highly tuned pipelines, ease of decoding the fixed length instructions, and highly optimizing compilers would be offset because of these aforementioned drawbacks. So they came up with the Compressed Code RISC Processor or CCRP having a special code expanding instruction cache. The object code is compressed at the host level and loaded on to the instruction memory and at run time the decompression is carried out by the instruction cache refill engine. The execution core would thus see them as standard RISC instructions. They only had to implement a new cache design for their verification. The authors were able to verify an increase in performance for some memory implementations, which reduced the memory bandwidth.

Suresh, Najjar, Vahid, Villarreal and Stitt [11] presented a profiling tool that is mainly dedicated for loop profiling. Their observation was that loop execution constituted the most executed segment of many programs and so they can be used for hardware partitioning. One tool presented was an instruction set simulator and the other was based on compile-time instrumentation of gcc. Their emphasis was that tools seeking to optimize the performance and/or energy consumption of em-

bedded software should focus on finding the critical loop code. The particular piece could be recompiled, synthesized with customized instructions or customized memory hierarchy and hardware/software partitioning. They suggest that instruction level profiling could be tuned to provide useful information about the percentage of time and resources spent inside different loops. They concluded that in their studies the contribution of the first 2-4 loops of embedded applications was nearly 90% and the contribution of the first 6 loops in the Spec benchmark contributed to nearly 55% of the execution time.

Lefurgy, Bird, Chen and Mudge [6] have developed a post-compilation analyzer that examines a program and replaces common sequences with a single instruction code word. The microprocessor would execute the compressed instruction sequences by fetching these code words from memory and then expanding them back to the original sequence of instructions in the decode stage of the pipeline and then issuing them to the execution stage. Reduction of the program size also reduces the instruction cache misses thereby further improving the performance. The authors found that there was a high degree of redundancy in the encoding of instructions and a small number of instruction encodings are highly reused in many programs. They compiled the SPEC CINT95 benchmarks [SPEC95] for PowerPC with GCC 2.7.2 with -O2 optimization and found that less than 20% of instructions in the benchmarks have bit pattern encodings that were used only once. Another observation was that in the go benchmark 1accounted for 30% of the program size, and 10of the instructions. All sequences of instructions that are frequently repeated are replaced by a single codeword and the encoded sequence are kept in a dictionary. The fi-

nal compressed program would consist of codewords and uncompressed instructions interspersed.

Chakrapani, Korkmaz, Mooney, Palem, Wong [12] have worked on compiler optimizations to reduce the power consumption of embedded processor systems. they have quantitatively characterized the limits of a compilers capabilities. They state that after the design of the computing element, micro-architecture techniques, software runtime system and lastly compiler optimizations. They suggest scheduling register access, proper code selection and minimization of power consuming instructions from the architecture.

Panda, Dutt and Nicolau [13] have presented a technique to organize scalar and array variables inside embedded code with the objective of improving data cache performance. They have clustered variables to minimize compulsory cache misses and for solving memory assignment problems to minimize conflict cache misses. They use a method to reduce the compulsory and conflict misses in data cache that account for 50% of all cache misses.

3. THE SIMULATORS AND THEIR WORKING

3.1. Description of the AE32000 Core

The AE32000 simulator was developed to accurately model its ISA and is used to execute different benchmarks to better understand the working of this model. The AE32000 simulator originally had very basic modules for loading the instructions and data memories a simple memory layout. The original simulator did not have any syscall support, so none of the programs with syscalls could execute till the end. In the following subsections the execution core and the enhanced loader, memory and syscall modules are described.

The core of the AE32000 has a 32-bit ALU, 32-bit barrel shifter, and a 32x32-bit parallel multiplier. The core has a 5-stage pipeline scheme as shown in figure 3.1. It has the register file with 16 32-bit registers which are connected to the buses ABUS, BBUS and CBUS. Out of these, ABUS and BBUS supply the operands to the EX unit and CBUS writes the results back to the register file or to one of the four special purpose registers. The core also has a prefetch queue which can hold up to 8 instructions including the instruction inside the Instruction Register (IR). The LERI instructions inside the queue are monitored and processed by a separate LERI folding unit.

3.1.1. The LERI Folding Unit

The AE32000 core can fetch up to two instructions at a time from the instruction memory and place them on to the circular queue. At the same time a

one bit value is set and pushed into the Instruction Queue to indicate whether the corresponding instruction is a LERI. For this purpose we have 1-bit of extra space added in to all the locations on the queue. After this check the other instructions are sent to the decoder unit. From the decoder the instruction heads to the execution, memory operation and write-back stages. The LERI instruction folding unit updates the 32-bit Extension register(ER) which then provides the expanded immediate operand. The additional LERI unit can process up to three successive LERI instructions without any performance loss, even if the next instruction in the queue requires this operand. The LERI unit checks the LERI flags and if that is set then the unit would prepare the value in a 32-bit format and give it to the corresponding instruction and sets the ER(Extension Register) and the IER(Immediate Extension Register) registers.

3.2. Simulator Core

The AE32000 simulator is based on a typical 5-stage pipeline architecture, as seen in figure 3.1 having the Instruction Fetch(IF), Instruction Decode (ID), Execution (EX), Memory Access(MEM) and the Write Back(WB) stages. These stages are described here:

3.2.1. Instruction Fetch

The instruction fetch module, as the name suggests fetches instructions from the instruction memory. We have a Program Counter(PC) register that indicates

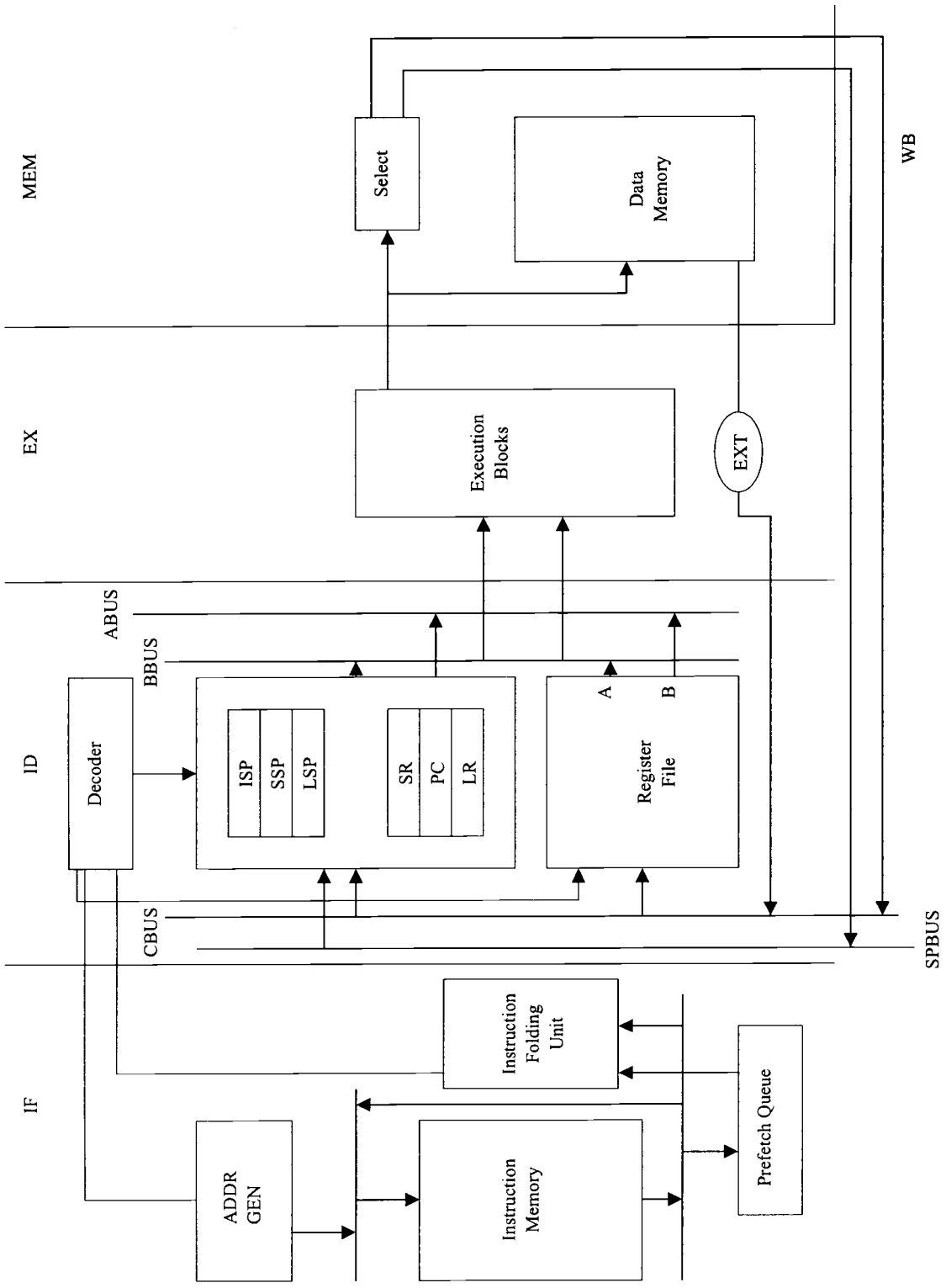

Figure 3.1. Basic structure of the AE32000 core.

the location of the instruction inside memory that has to be fetched. The addresses of memory locations are all 32-bits wide. At the start of IF stage the branch flag is checked, if it is set then the PC is updated from the Branch Target Address(BTA) register otherwise the next instruction is fetched. The PC value is checked to verify if this address is on a word boundary. If it is on a word boundary then we can fetch two 16-bit instructions from that location or in the other case we fetch only one instruction. If there is a successful BTB hit, then the instructions are fetched from this new address location and the instruction queue is cleared before it is loaded from the new location.The branch flag is set only inside the POP and JUMP instructions. Finally the instruction queue is updated inside this stage.

3.2.2. Instruction Decode

This stage determines the type of the instruction coming in from the IF stage. The categorization of instructions also takes place inside this stage. The important LERI folding unit also comes inside this stage. The instruction is decoded by shifting and comparing the opcode bits. Hazard detection unit is present inside this stage to decide if stalls have to be inserted into the pipeline. If instruction decoded is a register JUMP instruction and the previous instruction was a LOAD then a flag is set, if data hazard is predicted. The value of the branch flag and the branch target buffer flag are checked to calculate the new address in case of a hit. The various statistics for LERI count, branches, branch taken, total instruction count, etc. are updated here.

3.2.3. Execution, Memory Access, Write Back

These three stages are straight-forward. The execution stage handles the ALU operations, the Memory Access does the different loads and stores between the register file and the data memory. The Write Back stage then finally puts the data back into the register file.

3.2.4. Pipe Update

This module updates the pipeline after every cycle. This stage basically moves the instructions through the pipeline while checking for hazards. If the hazard flags are set then a bubble is inserted into the pipeline. The forwarding unit is also present here. There are two forwarding paths, one from the ALU output in the EX stage to the ALU input in the ID stage and the other from the memory data register in the MEM stage to the ALU input in the ID stage. A check is done to verify if there are any dependencies between the source registers of the instruction in the ID stage and the destination register of either the instruction in the ID or MEM stages. The forwarding can be done to either of the input source registers in the EX stage. Also the registers are written during the first half of the clock cycle and are read during the second half of the clock cycle to remove any data hazards during the WB stage.

3.2.5. Simulation steps in the AE32000

The simulator first initializes the register file, and then memory spaces are allocated for instruction, data, and initialized segments. The next step is to load the program into the simulated space and to fetch any initialized data into the data memory(dmem). Later the base and bounds locations for the instruction and data memory spaces are initialized. Next the core is reset during, which the register file, instruction queue, the pipeline registers and the various statistic gathering modules are reset. Also the endianness of the simulator is defined during the initialization of the program counter. The core executes for as many number of cycles as defined by the user with ability to stop the pipeline and gather statistics at any stage. Inside the core execution the five stages execute in the reverse order. This is done to update the register contents in time for use inside the subsequent instructions. Inside the pipe update the forwarding module is implemented which checks if there would be register values that need forwarding between cycles. After that the WB and the MEM stages are updated followed by checking of the pipeline hazard stall value which can insert a NOP into the pipeline. Also the simulator can be stopped at any cycle to check the current imem and dmem contents and also the istream stats.

The AE32000's loader was able to load only simple binary executables sequentially into the memory. The loader did not perform any checks on the header information for the contents or the size of the various sections inside the binary file. Also there was no support for syscall execution inside the simulator. Once the instructions are moved to the instruction memory the rest of the space was allocated for the dmem. The stack is assigned space at the bottom of the dmem and it grows

to lower addresses from this location. The memory space limits are specified in a script file. The limited space was strained for bigger programs and the heap and stack would collide. The limits for the memory space is changed inside the script file. The various enhancements to the AE32000 simulator are described in the following sections.

3.3. Loader Module

The Simplescalar-PISA has a fully tested loader and memory modules. This loader accepts binaries of only coff formats. If the bin format binaries have to be loaded in to the Simplescalar-PISA memory module then the loader would have had to be changed to accept this new binary format. The AE32000 compiler can compile a program and produce binaries of both elf and bin formats. So the loader can be modified to accept the elf binaries and some changes to the memory have to be made. But Simplescalar-ARM simulator also from Simplescalar has an elf loader. To add the SS-ARM loader and memory, only the file format and data type compatibility issues had to be resolved.

The code to call the loader module from inside the simulator is:

```
sim_load_prog(char *fname,                  /* program to load */
              int argc, char **argv,        /* program arguments */
              char **envp)                  /* program environment */
{
/* load program text and data, set up environment, memory, and regs */
ld_load_prog(fname, argc, argv, envp, &regs, mem, TRUE);
}
```

The different variables in the above piece of code are fname (name of the binary file to open), argc and argv (number of command line arguments and the pointer to the array containing the command line variables) and envp(describing the various environment variables). After initializing the loader it checks the format of the binary executable. The SS-ARM loader was modified to store the argv array at the start of the imem starting from location 0x0. The original cross compiler for AE32000 does not support passing of command line arguments directly as it was designed for an embedded processor environment. The disassembled files for the compile benchmark programs have instructions that direct the simulator to read the imem for argv variables. Also the argc value have to be hard-coded into every program that will be run on this simulator, also because of the assumptions at the time of compiler design . The figure 3.2 shows the old and new instruction memory on executing the dhrystone benchmark.

```
./happy dhrystone.elf input.txt
```

At the start of the loading there is a check for the magic number, which compares the first four characters from the elf file to 0x7f, E, L, F. Magic numbers help to distinguish between types of executables under UNIX. The loader then allocates memory space for the different sections, load the binaries into the buffer space and then moves the buffer contents into the simulator target memory. The header also holds information about the size of memory(for instruction and data) required. Later the bounds for the memory space are set. The next step is to check the endainness of the host and to swap words if the endianness do not match. Finally verification is done to check if the written data are all within bounds, and then control is passed back to the simulator program.

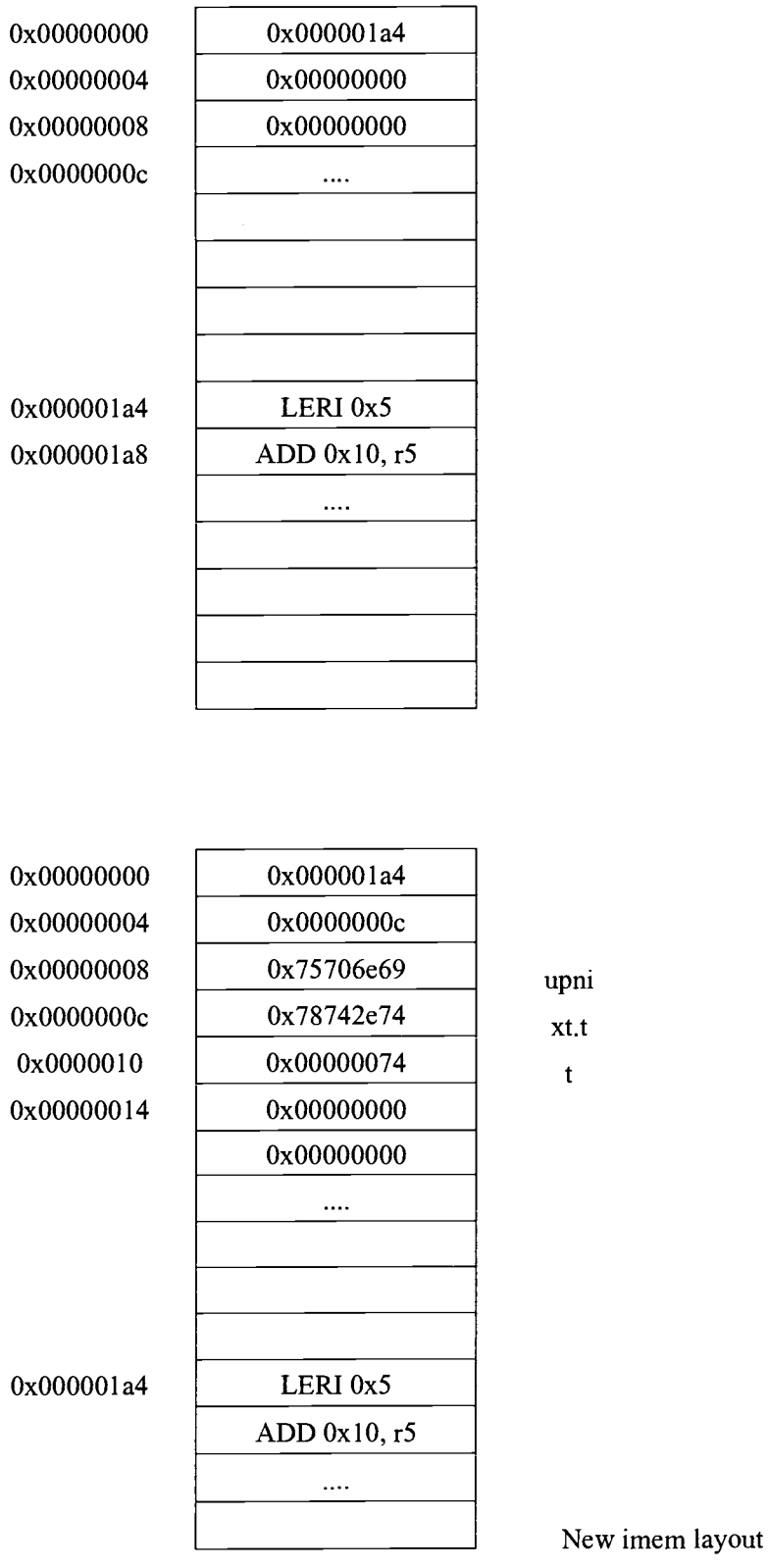

Figure 3.2. Setup at the start of the old and new instruction memories.

Register	*Content*
REG 8	Syscall Number
REG 9	File Descriptor
REG SP + 12	Pointer to Buffer
REG SP + 16	Number of Bytes

Table 3.1. Table showing the contents of different registers being passed to the syscall function.

3.4. Syscall Module

The AE32000 simulator did not have support for syscall execution. For every syscall occurring in the executed code, the simulator would replace it with an NOP. The SS-ARM proxy syscall handler was ported to the AE32000. The syscall handler program implements a subset of the Ultrix Unix system calls. Basically the algorithm decodes the system call, stores the arguments if any in to the simulated memory space, executes the system call, and finally it copies the results back in to the simulated memory. The syscall module is based on the unistd.h of the arm architecture. Depending on the type of the system call five registers will contain certain values. The instruction is decoded to find out the system call number. The return value is inside register 8. If the returned value is between -1 and -125, included, then an has occurred during the syscall execution. The table 3.1 shows the register numbers and their corresponding contents when the system call module is invoked.

Here the execution method for the READ syscall is shown. The following piece of C code calls the syscall module from inside the AE32000 simulator.

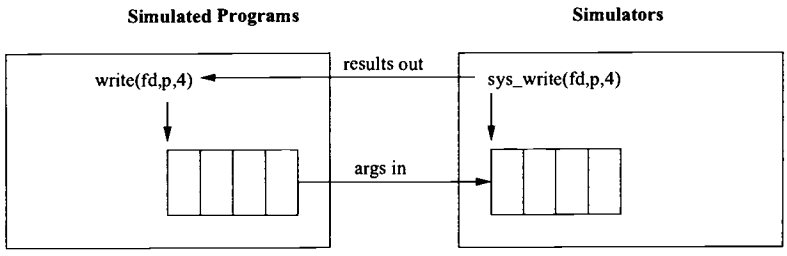

Figure 3.3. A write syscall implementation.

```
sys_syscall(&regs, mem_access, mem, FUNC, TRUE);
```

The above function call passes a pointer to registers, instructs the callee program that an access to data memory will occur, a pointer to the memory space to access, the system call number and also indicates that it will be a traceable system call. Once inside the syscall module, sufficient buffer space is allocated in the host memory and the file descriptor number, the address of the buffer space and the number of bytes to be read are passed to the actual read() function call. If the value returned from the read is between -1 and -125 then an error has occurred inside the host. Finally the contents are moved from the buffer in the host memory to the data memory location as specified in register 9. In case of a successful read, the value returned is the number of bytes read, which is saved in register 8.

3.4.1. Incorporation of the Syscall Module

In the disassembly file where the piece of code that called the SWI function, it was observed that the cross-compiler provided support for syscall execution. Once the basic programs were incorporated the source registers inside the syscall program are modified to match with those set by the AE32000 cross-compiler. The instruction stream dump was used to decide the corresponding register numbers containing the different values (file descriptor, number of bytes to be read/written, buffer location inside the dmem, etc.) being passed.

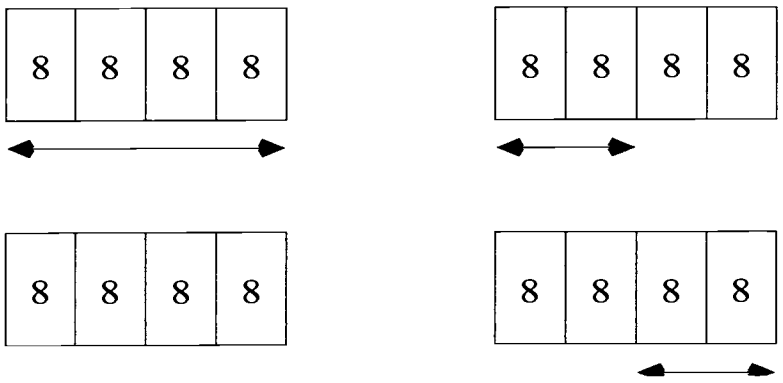

Figure 3.4. The arrangement of a word inside the memory.

3.4.2. Word Boundary Adjustment

In one of the benchmarks it was observed that during the open syscall, for particular filenames and file mode options the simulator could open the input file correctly, but if either the permissions or the file names were changed an error occurred due to which the file would not be opened. The files with the names that are either of four or eight character width are opened correctly. It was later found that a mismatch occurred just before the filename was read from the instruction memory. The SS-ARM memory copy functions are defined to recognize contents that lie only at the word boundary. So if there was a data memory read from a location that is not at the word boundary, then SS-ARM will adjust the address to the next word boundary location. The data is read from this location and later shifted to obtain the corresponding data. But the AE32000 was defined to read the contents from any memory location. The AE32000's address generation module was changed to always provide a word aligned address to the data memory reading functions.

3.5. Memory Module

This section presents the memory module of the AE32000 and the enhancements made. In the previous memory module, a script file contained the range for the instruction and data memory base and size and the BTB replacement methods. The loader would not check the limits of the available space and continue during the contents in to the memory. Also the space assigned was too small for our benchmark programs. Based on the script file sufficient character space is allocated and all this

space is cleared. In case of the ARM module the memory sizes are dynamically allocated at run time. Only the maximum space that the simulated memory could take up inside the host is set. The upper limit for the dmem is set at 0x0c100000. The loader transfers the instructions to the start of the memory from 0x0. After the imem the rest of the space is allocated to the dmem. The memory is set up so that after the program code and global variables the remaining space, called the heap, is used for the uninitialized segment.

The AE32000 had a memory that was a static array. The size of the array is set by the user at the start of execution and therefore needed us to have an estimate of the amount of memory space needed. It originally had 64K allocated for imem and 256K of space for dmem.

The ARM memory module has a virtual space of 2^{31} bytes. The address space is divided into Unused, Text, Data(Init/BSS/Heap) and Stack spaces. The address space from 0x400000 to 0x10000000 is for the program text, and then from here onto the top of the stack is the heap. The stack space is used to hold program arguments and the environment variables. Also the virtual memory space is implemented as a single level page table. The loader uses the following copy function to load each section into the target memory.

```
mem_bcopy(mem_access, mem, Write, shdr.sh_addr, buffer, shdr.sh_size);
```

where mem_access specifies the user specified memory accessors, mem specifies the memory space to access, cmd decides whether it is a read from or write to simulated memory, buffer gives the host address to access, and lastly shdr.sh_size indicates the number of bytes to be moved.

3.6. Description of ARM Architecture

The ARM architecture [3] has been designed to bring together a small and high performance implementation. It has features like a large uniform register file, load/store architecture, simple addressing modes and uniform instruction length in user mode operation. The other features are ability to control both the ALU and the data shifter, auto-increment and auto-decrement addressing modes to optimize the loops, load and store multiple instructions and conditional execution of every instruction to maximize the execution throughput. ARM has 31 general purpose 32-bit registers, but at any time only 16 registers are visible during the user mode. Out of these 16 registers two registers have special roles, one is register 14 which acts as the Link Register (LR) that holds the address of the next instruction after the Branch instruction(BL), and the other is register 15 which is the program counter(PC). All ARM instructions are 32-bits long, so they all occur at the word boundary in the memory.

3.6.1. Thumb Instruction Set

The Thumb instructions are basically a re-encoded subset of ARM instructions. Thumb is designed to increase the performance of ARM implementations that use a narrower data bus and allows better code density than ARM. Every Thumb instruction is encoded in 16-bits. Thumb merely presents the programmer restricted access to the ARM architecture. All Thumb instructions act on 32-bit instructions and also 32-bit addresses for both data access and instruction fetches. When the

processor is executing Thumb instructions only the eight GPR's from R0 to R7 are available and some instructions also access the PC, LR and stack pointer (R13) registers. Some instructions have limited access to registers 8 through 15, called the high registers.

Thumb execution is normally entered by executing the ARM BX (Branch and Execution) instruction. This instruction branches to the address held in the general purpose register and if bit 0 of that register contains 1, then Thumb begins to execute at the branch target address and if bit 0 of the target register contains 0, then the ARM execution begins from the branch target address.

4. RESULTS

The simulation model explained in the previous chapters was used to generate the following results. There are special modules inside every simulator which gives it an advantage in some cases. We are therefore bound to have differences in the results generated even after executing similar codes. The SS-ARM was used with configuration set to make it run as close as possible to a 5-stage pipeline. All the programs and the simulators are written in C. As we run the ARM in user mode there is no switching to the thumb instructions.

Table 4.1 shows the cycle count. The number of cycles for the integer benchmarks(ADPCM,G.721,JPEG) are nearly the same with a difference in the JPEG benchmark. We can see that the number of cycles for the floating point benchmarks(MPEG,EPIC) are extremely large. Some of the reasons for these differences are the better gcc version used for the compiler in SS-ARM, shorter instruction length in AE32000, embedded instructions in SS-ARM. The detiled results are shown in the future pages.

ARM has consistently lesser number of instruction count compared to AE32000 according to the following table 4.2. The instruction count for the floating benchmarks are also drastically more than the integer benchmarks. From table 4.2 it is seen that MPEG and EPIC take nearly 90 times more instructions for execution. The AE32000 uses macros for decoding such floating point instructions. In this case it was observed that a single macro could cost up to 250 more instructions inside the AE32000 for every floating point instruction.

The ARM also has LDM/STM instruction which are multiple loads and stores. The ARM processor executes more number of ALU operations in most of

	Total number of cycles	
	AE	*SS-ARM*
ADPCM	26,57,300	17,199,924
EPIC	7,468,254,770	107,765,075
G.721	889,841,000	733,605,810
JPEG	165,001,000	32,043,419
MPEG	111,66,128,844	1,766,823,273

Table 4.1. Cycle count.

	Total number of instructions	
	AE	*SS*
ADPCM	22980455	14787963
EPIC	6869787870	71083219
G.721	657787782	755955225
JPEG	25766785	10905456
MPEG	104006775804	1172145689

Table 4.2. Instruction count.

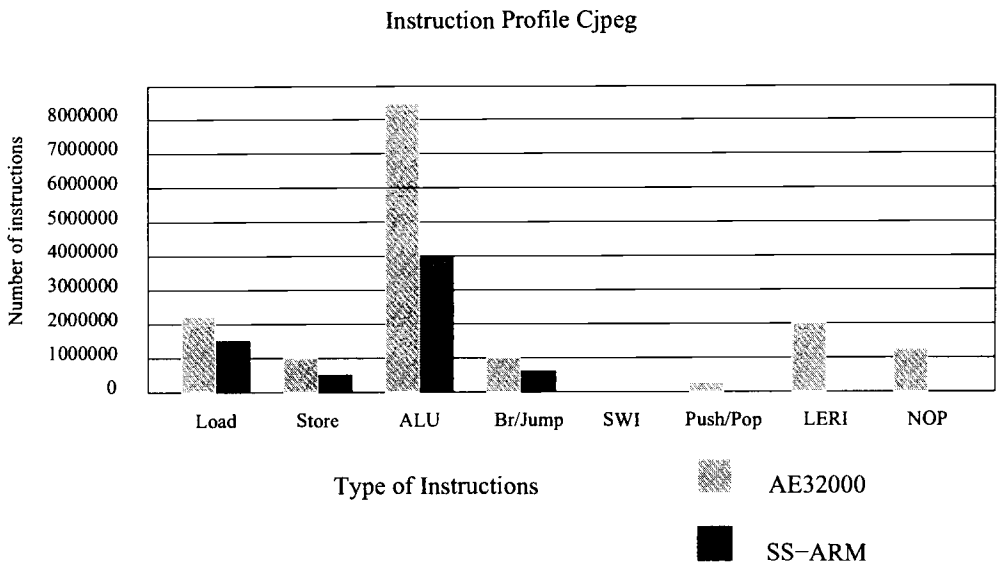

Figure 4.1. Profiled values under AE32000 for the JPEG Compression.

the benchmarks tested. But as most of the loads and stores are LDM/STM ARM would have fewer number of accesses to the memory. For LDM the core pushes the values of the consecutive registers in to the stack memory at continuous locations. For STM it does it the other way round to fetch the data from the stack.

In few benchmarks it is observed that the number of system calls is nearly twice in case of AE32000 as compared to Simplescalar. This happens even though the compilation options being kept same at -O3. One of the possible reasons for this difference could be that the AE32000 cross-compiler is designed such that for each Read or Write syscall executed it can transfer only a maximum of 400 bytes. SS-ARM does not have such a hard upper limit on the number of bytes to transfer during Read or Write syscalls.

	AE32000 *number of Syscalls*	*ARM* *number of Syscalls*
EPIC	1384	160
JPEG	236	103
G721	1830	25
ADPCM	670	570

Table 4.3. Profiled values for Syscalls during execution.

The execution core of the ARM has only three stages Fetch, Decode and Execute for execution of instructions. So this reflects in the throughput for the simulations. On an average the AE32000 takes 30 times more amount of time for the same benchmark as would be taken by SS-ARM. This will result in a reduction of the pipeline latency in SS-ARM.

Table 4.4 shows the number of ALU instructions executed in both the simulators. For G.721 and ADPCM these numbers are lesser by 18% and 28%. The reason for difference in the results for JPEG is given on the next page. The AE32000 instruction set does not have support for executing floating point operations. From table 4.2 it is seen that MPEG and EPIC take nearly 90 times more instructions for execution. The AE32000 uses macros for decoding such floating point instructions. In this case it was observed that a single macro could cost up to 250 more instructions inside the AE32000. For the ADPCM and G721 benchmarks, which do not have floating point operations, it is seen that the percentage of ALU operations in AE32000 is lesser compared to SS-ARM. The possible reason could be that

	AE32000 *percentage of ALU ops*	*ARM* *Percentage of ALU ops*
ADPCM	8,590,540	11,008,856
EPIC	2,674,899,505	32,678,211
G.721	408,722,389	483,533,717
JPEG	12,707,420	6,321,881
MPEG	33,246,065,965	674,767,471

Table 4.4. Percentage of ALU operations in AE32000 and SS-ARM

there are more different ALU operations in AE32000 which can better perform these operations.

ARM also has the feature of embedded instructions, in which there is a shift instruction inside another ALU instruction. This instruction occur frequently inside the JPEG benchmark.

```
ADD r5, r6, r7, LSL \#2
operation r5 = r6 + 4 * r7
```

Here LSL is logical Shift Left by the number of bits as specified by the immediate value. This instruction can be executed in a single clock. ARM is able to reduce the count for ALU operations with such embedded instructions.

For branch execution ARM has conditional execution for any instruction. Depending on the content of the Current Program Status Register(CPSR) the negative, zero, carry and overflow flags are set. On comparison with the Status Reg-

ister the condition can be evaluated. There are instructions like Test(TST), Test Equal(TEQ), Compare(CMP) and Compare Negative(CMN) which can set the flags. This results in only two branch instructions inside the SS-ARM compared to 15 branch instructions inside the AE32000.

```
ADDEQ r1, r2, r6
operation r1 = r2 + r6 ; if zero flag is set
```

5. CONCLUSION AND FUTURE WORK

Our comparison of the AE32000 and the Simplescalar-ARM instruction sets shows that the AE32000 with its shorter fixed length instruction set causes fewer memory accesses. But, when it comes to performance then ARM is better in terms of cycle and instruction count. Based on the profiling of the instruction sets, directions have been found in which if further development and testing is done then the AE32000 would certainly find much wider applications. AE32000 is better when it comes to ALU operations. But for floating point programs ARM easily outscores.

A new cross-compiler with better libraries for AE32000 can also speed up its performance especially for floating point intensive operations. More testing for the other benchmarks in the Mediabenchmark suite can also be done provided a better compilation environment is developed. Support for shared libraries in the AE32000 simulator can be added. This simulator can evolve into a general framework for future embedded core simulators which would help in prototype development.

BIBLIOGRAPHY

[1] http://www.embedded.com/

[2] H.C. Oh, H.G. Kim, H.S. Jung, J.W. Lee, B.J. Kim, J.Y. Jung, B.G. Min, J.Y. Lim, H. Lee, and K,-H. Kwon, "AE32000: An Embedded Microprocessor Core", *Proc. of 2nd AP-ASIC*, Aug 2000, pp. 255-258.

[3] David Seal, "ARM Architecture Reference Manual", *Addison-Wesley*, 2001, Great Britain.

[4] K.D.Kissell, "MIPS16: High-Density MIPS for the Embedded Market", *MIPS Tech., Inc.*, http://www.mips.com/Documentation/MIPS16Whitepaper.pdf.

[5] Heui Lee, Paul Beckett, Bill Appelbe, "High-Performance Extendable Instruction Set Computing", *IEEE*, Proc. of 6th ACSAC 2001, Jan 2001, pp. 89-94.

[6] Peter Bird, and Trevor Mudge, "An instruction Stream Compression Technique", *Technical Report CSE-TR-319-96*, EECS Department, University of Michigan, Nov 1996.

[7] Andrew Wolfe, Alex Chanin, "Executing compressed programs on An Embedded RISC Architecture", *Proc. 25th Int. Symposium on Microarchitecture* , pp. 81-91, Dec 1992

[8] Manfred Schlett,"Trends in Embedded Microprocessor Design", *Computer*, 31(8):44–49, Aug 1998.

[9] Peter L. Bird, Trevor N.Mudge, "An Instruction Stream Compression Technique",*CSE-TR-319-96*, EECS Department, university of Michigan, November 1996.

[10] Young-Jun Kwon, Danny Parker, and Hyuk Jae Lee, "TOE: Instruction Set Architecture for Code Size Reduction and Two Operations Execution", *Int. Workshop on Compiler and Architecture Support for Embedded Systems*, Oct 1999.

[11] Dinesh C. Suresh, Walid A.Najjar, Frank Vahid, Jason R Villarreal, Greg Stitt, "Profiling Tools for HardwareSoftware Partitioning of Embedded Applications", *LCTES '03*, June 11-13, 2003, San Diego, California, USA.

[12] L.N.Chakrapani, P.Korkmaz, V.J. Mooney III, K. Palem, and W.F. Wong, "The Emerging Power Crisis in Embedded Processors: What can A Poor Compiler

Do?," *Proceeding on International Conference on Compilers, Architectures and Synthesis of Embedded Systems* pp.176-180, November 2001, Atlanta, Georgia.

[13] Preeti Ranjan Panda, Nikhil D. Dutt, and Alexandru Nicolau, "Memory Data Organization for Improved Cache Performance in Embedded Processor Applications," *IEEE Transactions on Computer Aided Design of Embedded Systems* vol.18, no.1, January 1999