AN ABSTRACT OF THE THESIS OF

Siva Sai Yerubandi for the degree of Master of Science in Electrical and Computer Engineering presented on April 07, 2004.

Title: Development and Enhancement of AE32000: An Embedded Microprocessor Core

Abstract approved: ______________________________

Ben Lee

AE32000 microprocessor was developed mainly to address the need for the reduction in the amount of memory accesses in embedded applications. One of the primary goals of a computer architect is the design and construction of machines, that support the efficient execution of the programs that will run on them. The simplicity of the instruction set provides a number of implementation advantages that can substantially enhance the performance of the machine. The use of fixed length instructions and a few formats permits simpler hardware, faster instruction execution and low power consumption. These advantages are influenced by the usage of LERI instruction. The number of ALU operations executed in AE32000, using the integer only instruction set, are greatly reduced compared to the mixed instruction set architecture like ARM.

Index terms: 16 bit instruction length, Extendable Instruction Set Computer, Load Extension Register Immediate LERI, elf binary, code density.

Development and Enhancement of AE32000: An Embedded Microprocessor Core

by

Siva Sai Yerubandi

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 07, 2004
Commencement June 2004

Master of Science thesis of Siva Sai Yerubandi presented on April 07, 2004

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Associate Director, School of Electrical Engineering and Computer Science

Dean of the Graduate School

Siva Sai Yerubandi, Author

ACKNOWLEDGMENTS

I deeply thank my advisor, Dr. Ben Lee, whose help, advice and supervision was invaluable. Without his help, this work would not have been possible. I would also like to thank the members of my committee who braved the storm of the century to attend my defense: Dr. Alexandre F. Tenca, Dr. Roger L. Traylor and Dr. Jack Higginbotham. Their advice and patience is appreciated. I thank Ferne and all the staff members of the Electrical Engineering and Computer Science Department who had the answers for all my administrative and other questions.

Special thanks to my officemates Weetit Wanaletlak, Balaji Megarajan, and John Mark Matson for their assistance, conversations and for generally putting up with me. Those long nights infront of the computer in Owen345 lab will not be forgotten. I would also like to thank Sanka Srinivas, Imran Khan, Harish Peddibhotla, Harish Gandhi, Sudheer Vemulapally, Hari Priya, Lakshmi Swetha, Anand Ganesan, Madhusudhanan who are not only colleagues, but life-long friends.

No words can express my debt to my family: my parents, my brother and my cousin Parthasarthy who have always put my interests before theirs. Whatever I am and wherever I shall be, I owe it all to them.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Development and Enhancement of AE32000: An Embedded Microprocessor Core

1. INTRODUCTION

In the past, a embedded processor used to mean a 4-bit or 8-bit low-end microcontroller, designed primarily to perform simple control applications. Presently, with the advent of technology the contemporary embedded system landscape cuts a broad swath from low-end microcontrollers to high-performance processing engines. The widespread development of the embedded processors in various mobile devices promises to open new frontiers in applications. These contemporary systems usually employ either one or multiple processor cores, integrated into a **S**ystem-**O**n-**C**hip (SOC) design. Usually, such cores are fixed processors taken from a library of well-known processor architecture families, which include ARM or MIPS. Integrating the core with reconfigurable logic is another way to boost software performance while retaining hardware-acceleration benefits.

However, the need for differentiation coupled with ever-increasing chip capacities, opens door for significant customization of the processor cores. This gives embedded system designers virtually unlimited choices in processor core architectures, allowing them to customize several features to suit the application and design constraints at hand. For instance, designers can tune the processor's instruction set architecture to the application's characteristics. Similarly, they can optimize the processor pipeline or data path components for critical code segments in specific application domains. The resulting processors remain programmable and, in principle, can run any application, but they are optimized for a targeted application domain [1].

The rapid advances in semiconductor technology throughout the last decade of the 20th century have enabled the development of powerful new embedded processors, bringing to the embedded world computational power, previously found only in mainframes and supercomputers. Due to its dissemination among various types of application domain, embedded systems are more affected by market constraints. These constraints include:

- Time-to-market: Embedded systems require longer development time, since their hardware and software design has to be finished before it can be released as a commercial product. This is because there can be no updated software versions at a later date.

- Safety: Often, embedded systems are part of a critical system, and hence subject to safety and reliability requirements, in order to guarantee a certain level of robustness. A more robust design tends to increase the cost, which cannot exceed a certain value.

- Flexibility: Since embedded systems are application-specific units, it is difficult to provide an appropriate degree of programmability.

- Mobility: Embedded systems aimed at targeting mobile applications, such as wireless telecommunication or portable information processing devices, are likely to be severely affected by other factors, like size and energy consumption.

Taking into consideration the time factor, reliability and endurance tests for the production of any embedded processor, the software developer has to verify for hardware engineers on the same simulated configurations so that product development time is saved. A lot of integrated profiling enables software optimization with the target architecture comprehending memory and micro-architecture. A software solution for an embedded processor design and optimization environment slashes months of hardware design time. To do all this a cycle-accurate, execution based simulator is the best option. A cycle-accurate simulator should have all the modules needed, like memory, cache, system calls and statistics gathering modules. This led to the development of the simulator for the AE32000 embedded processor. AE32000 was a primitive core processor that had very limited capabilities. There was no support for any information needed by a designer to use this simulator. One by one advanced modules for ported on to the simulator and finally develop a primitive simulator into a cycle-accurate simulator. AE32000, at this present stage, has enabled to advance the software development cycle and also debugging of the actual target hardware.

1.1. Organization of Thesis

The thesis is organized as follows: chapter 2, discusses the previous work done on latest ideas suggested and implemented in the field of embedded processor technology. Chapter 3 presents the organization and hierarchy of AE32000 simulator. Modules used to enhance the functionality of the simulator are explained in chapter 4. Chapter 5 discusses the various bugs found in the simulator and compiler; and explains the ways to rectify the errors. Chapter 6 gives a detailed report on benchmarking. The concluding remarks are provided in chapter 7.

2. LITERATURE REVIEW

This Literature review investigates the previous research work done to enhance the performance of embedded processors.

> **Definition:** *Embedded systems are (inexpensive) mass-produced elements of a larger system providing a dedicated, possibly time-constrained, service to that system* [3].

The technological developments that allowed single-chip processors(microprocessors) made the embedded systems inexpensive and flexible. For example in modern cars such as the Mercedes S-class or the BMW 7 series there are more than 60 embedded processors that control a multitude of functions e.g., the fuel injection and the anti-lock breaking system (ABS), that guarantee a smooth and foremost safe drive [3]. Applications like handheld, palmtop, network PCs, require storage media, display and interface to communicate with the outside world. This have made the embedded processors incorporate capabilities traditionally associated with conventional CPUs-but with a twist. They are always subject to challenging cost, power consumption and application dictated constraints.

2.1. Power Consumption

System design is a process of implementing the desired behavior while optimizing the objectives and satisfying some constraints. For example, a desktop computer, is optimized for maximum clock rate while meeting a price target. On the other hand, a battery-operated portable embedded processor system for information processing and wireless communication will be optimized for minimum power dissipation subject to a required performance. There exists a dichotomy in the design of modern electronic systems: the simultaneous need to be low power and high performance [4]. This arises largely from their use in battery operated portable platforms. Accordingly, the goal of low-power design for battery-powered embedded system is to extend the battery service life. The twin issues of modelling and optimization of power consumption have to be addressed at several layers for an accurate and thorough result [5]. Previous compiler optimization research in this area has either relied on actual power measurements of the processors [6] or has relied on architectural simulation [7], mathematical techniques and regression to model the behavior of the hardware. One of the power research infrastructures consists of an optimizing compiler infrastructure called Trimaran [8]. A backend of Trimaran, called Triceps has been developed to generate code, which targets the ARM [9] architecture. Other than the hardware approach of saving power for embedded processors, there are a variety of software design techniques can reduce power consumption.

- Intelligent waiting : Many of the latest embedded processors include run-time power modes to scale power consumption. The most common case is idle mode. In this mode instruction-executing portion of the processor core shuts down, while all peripherals and interrupts remain powered and active. Idle mode consumes substantially less power than when the processor is actively executing instructions [10]. A key aspect of idle mode is that it requires little overhead to enter and exit this state. But however to gain maximum power efficiency software designer has to be very careful in designing the software

- Event reduction : Intelligent waiting enables the processor to enter its idle mode as often as possible. Event reduction attempts to keep the processor in idle as long as possible. It is implemented by analyzing code and system requirements to determine a way if programmer can alter the way interrupts are processed [10]. A multitasking operating system using time–slicing to schedule threads, sets a timer interrupt to occur at the slice interval. In a case where software code makes good use of intelligent waiting techniques, the operating system will frequently find opportunities to place the processor in idle mode, where it stays until it's awakened by an interrupt.

- Performance control: Dynamic clock and voltage adjustments represent the cutting edge of power reduction capabilities in microcontrollers. The energy consumed by a processor is directly proportional to the clock frequency driving it and to the square of the voltage applied to its core [11]. Processors allowing dynamic reductions in clock speed provide a first step towards power savings; cut the clock speed in half and the power consumption drops proportionately. To implement effective strategies using this technique alone is tricky, since the code being executed may take twice as long to complete. In this case, no power can be saved. In dynamic voltage reduction, an increasing number of processors allow voltage to be dropped in concert with a drop in processor clock speed, resulting in a power savings even in cases where clock-speed reduction alone offers no advantage.

- Intelligent shutdown: In all above cases the device is running; in this case device is turned off. In an intelligent shutdown procedure this effectively tricks any executing application software into thinking that the device was never turned off at all. When user turns the device off by pressing the power button, an interrupt signals the operating system to begin a graceful shutdown that includes saving the context of lowest-level registers in the system. The operating system does not actually shut programs down, but leaves their contents (code, stack, heap, static data) in memory. It then puts the processor into sleep mode, which turns off the processor core and peripherals but continues to power important internal peripherals, such as the real-time clock. In addition, battery-backed DRAM is kept in a self-refresh state during sleep mode, allowing its contents to remain intact. When restarted, an interrupt signals the processor to wake up. The wakeup ISR uses a checksum procedure to verify that the contents of the DRAM are still intact before restoring the internal state of the processor [10].

2.2. Code Optimization

Efficiency of the generated code is very important for embedded systems, due to limited system-on-a-chip memory sizes, real-time constraints of embedded applications, and the need to minimize power consumption. Code Optimization can be done at different levels in the compilation flow, reaching from source level to assembly level techniques [12]. All reasonable compilers perform machine-independent standard optimizations, such as constant folding, common subexpression elimination, or jump optimization [13] [14] [15]. These techniques need only a minimum of machine specific information and are useful for most programs.

- Address code transformation : The high-level address code transformation techniques described in [16] can be regarded as an extension of the standard optimizations. This is targeted towards memory-intensive applications. The main idea here is to simplify the array index expressions beyond the classical induction variable elimination technique (see e.g. [13]). At the expense of larger code size, this leads to a reduction of up to 50% in instruction cycles for array intensive nested loops [12].

- Loop transformations : These transformations are use for effective code optimization in case of multimedia applications mapped to VLIW processors. Loop unrolling is a classical example where loop iterations are duplicated, resulting in larger basic blocks and thereby in a higher potential for parallelization of instructions during scheduling. Its counterpart is loop folding or software pipelining [17] [18], where loop iterations are restructured in such a way, that the critical path length within the loop body is reduced. These loop optimizations come at the price of an increased code size.

- Function inlining: This is a technique where function calls are replaced by copies of function bodies, so as to reduce the calling overhead. Compilers use local heuristics in order to identify suitable candidate functions for inlining, while mostly neglecting code size constraints. In contrast, the inlining technique described in [19] aims at a maximum program speedup for a given global code size constraint, and thus better meets the demands of embedded processors. It is based on profiling information, code growth estimation, and a branch-and-bound optimization procedure. This technique also exemplifies a common concept in code generation for embedded processors. This approach is valid, since for embedded systems high code quality is much more important than high compilation speed.

All the above said optimizations are for the machine independent statements. After this optimizations the machine specific features such as special-purpose registers complex instruction patterns are taken into account. Tree pattern matching is the very basic technique for instruction set mapping or code selection [20]. Instructions are represented in the form of data flow trees, where tree nodes corresponds to variables,constants and operations, while edges denote data dependencies [12]. An optimum mapping is given by minimum cover of DFT by cost attributed instruction patterns (figure 2.1). Of the many tools used for automatic generation of tree pattern matchers from instruction set grammar specifications are, IBURG [21], BEG [22], OLIVE [20].

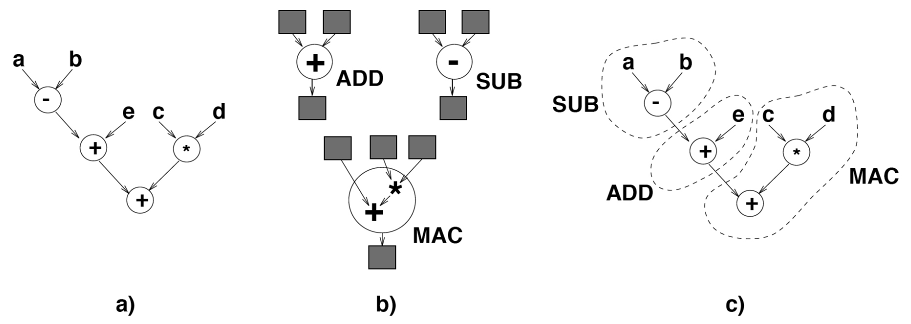

FIGURE 2.1. Page translation.

a) Data flow tree, b) available instruction patterns, c) optimal tree cover

After all these code optimizations there is still a potential left in the code optimization with respect to memory access organization and instruction scheduling. Instruction scheduling assigns generated machine instructions to control steps, which is important for VLIW-like multimedia processors as well as DSPs with limited instruction level parallelism.

3. AE32000 SIMULATOR

AE32000 is part of a family of microprocessors using the EISC (**E**xtendable **I**nstruction **S**et **C**omputer) core developed to address the need to reduce the code size and the number of memory accesses in todays embedded applications. EISC is a new architecture and is a combination of RISC and CISC. ESIC not only includes the advantages of both RISC and CISC but also has a very simple architecture. One of the main advantages of ESIC is its code density; a typical program size of EISC is **40%** smaller than the RISC, **20%** smaller than CISC.

AE32000 is a powerful, flexible and user-customizable 32–bit core providing a faster design solution for embedded processor based designs. AE32000 belongs to the family of EISC series and is optimized for Embedded Application for high–performance microprocessor computing. EISC AE32000 processor uses a 16–bit ISA. It contains **74** (**24** data transfer, **20** ALU, **18** branch, **3** shifts, **7** Miscellaneous and **2** coprocessor) instructions (no floating-point instructions) with **16** 32–bit General Purpose Registers (GPRs) and **7** Special Purpose Registers (SPRs). The main advantage of using EISC processor's ISA is that the hardware (i.e., HDL description) for the processor is already available. Various advantages of ADC's EISC AE32000 core other than its simplicity is robustness, code density of the binary file generated.

3.1. General Description

The AE32000 microprocessor has a high-performance 32-bit EISC architecture that is designed for an embedded application system with 16-bit instruction set and 32-bit data/address bus. The AE32000 provides the following features.

- 5 Stage Pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access(MEM), Write Back (WB)
- Harvard architecture
- Support 32bit ALU Operations, 32bit x 32bit = 64 multiplier, MAC unit and Huffman decode
- 16 general-purpose register and 7 special-purpose register
- Folding of the function for extension instruction (LERI) to obtain increased performance.
- Push/Pop Reg. List : Registers to maximum 8 in one instruction Push/Pop
- 14 types of conditional branch instructions
- Little endian

3.2. Architecture Overview

This section describes about the summary of the AE32000s hardware architecture, AE32000s peripheral block, memory architecture, and instruction folding without support for the additional delay of extension instruction and co-processor interface.

3.2.1. Architecture Description

The AE32000 is based on the Harvard architecture which separates memory spaces, one for instruction and one for data, each with their own bus and consists of a five stage pipeline. Its code density is very high because the size of instruction set is fixed 16-bit length. The extension instruction easily creates a 32–bit immediate value and can address large memory locations. The Instruction folding of an extension instruction reduces the additional delay caused by extension instruction.

The AE32000 has the strong arithmetic function that can be used for DSP applications. It has an ALU and Barrel Shifter that support 32–bit arithmetic and 32 x 32 bit multiplier. Also, it has the CNT instruction (used to count leading 0's/1's in a register) to support effectively Huffman decoding used in special purpose application and MAC operator that is used for DCT arithmetic or other DSP application. AE32000's functionality can be extended through co-processors: Up to four co-processors can be supported. Co-processor 0 is reserved as a system co-processor for managing MMU, OSI, Memory bank control and co-processor 1 as a floating-point arithmetic co-processor.

3.2.2. Functional Block Diagram

Figure 3.1 shows AE32000s basic blocks and data path. More detailed explanation about each block is available in subsection 3.3. AE32000 has a 5-stage pipeline structure and three data buses (ABUS, BBUS, CBUS) that connect to the register file. ABUS and BBUS are the input busses to arithmetic block; CBUS is used as the output bus when the core writes back data to an internal register.

The AE32000 can prefetch up to 8 instructions and has an instruction folding unit.

3.3. Block Summary

This section briefly describes about peripheral blocks that are used in the AE32000 Simulator core.

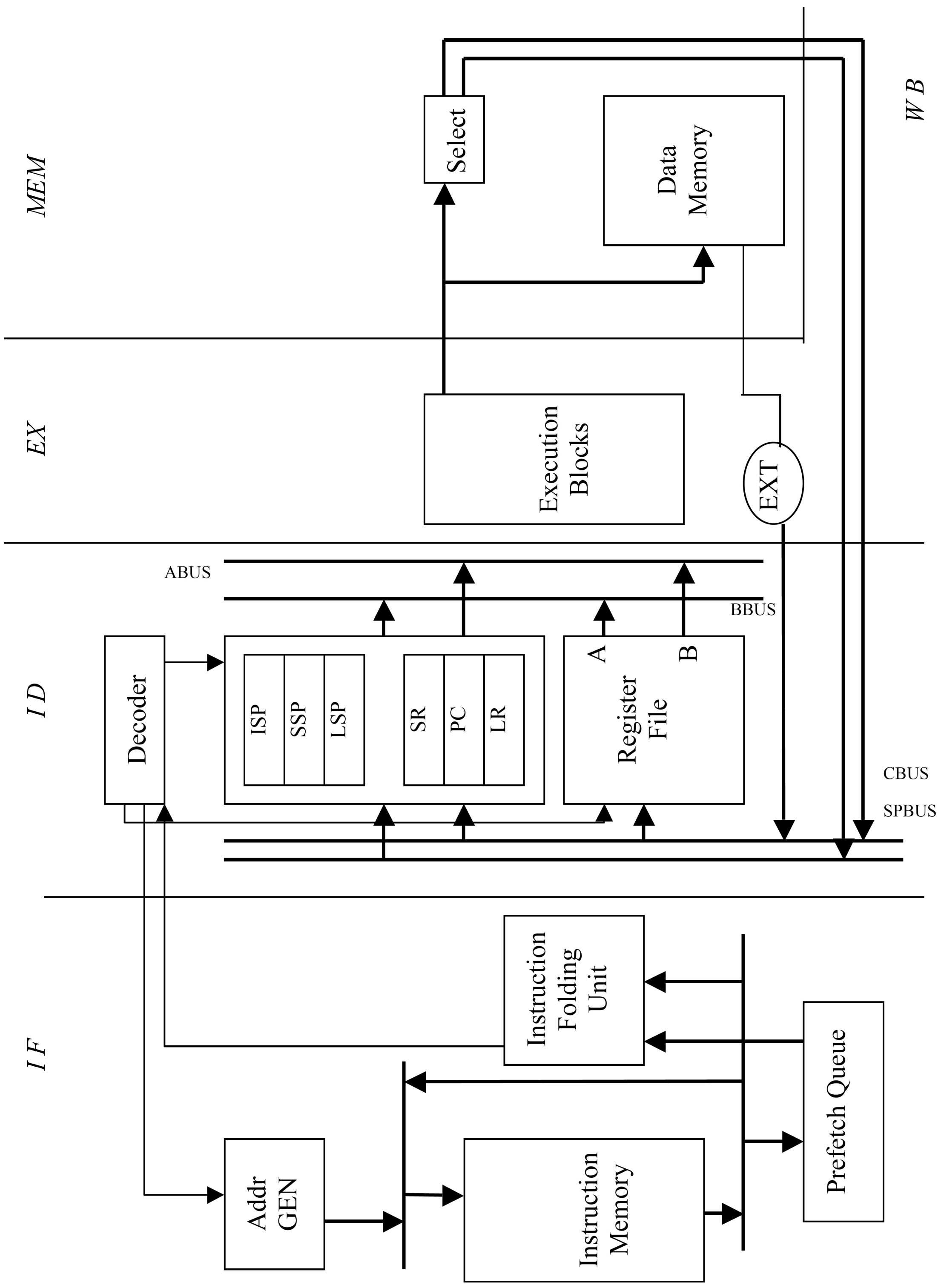

FIGURE 3.1. Functional Block Diagram.

3.3.1. Address Generator

The Address Generator Unit (ADDRGEN) pre-fetches a 32-bit address. The pre-fetched address is usually the last prefetch address + 4. This unit also has the capability to calculate the Branch Target Address (BTA), Interrupt vector address (EPC: Exception PC) and Interrupt handler address.

3.3.2. Prefetch Queue

The Prefetch Queue Unit consists of an 8 (16+1)-bit ring buffer queue register. It temporarily stores the prefetch address (16-bit) and also sets a bit flag to indicate if the instruction is an extension instruction.

3.3.3. Instruction Folding Unit

This Unit has 3×16-bit inputs and either 1×16-bit or 1×32-bit outputs. Depending on the instruction (check the extension flag) the output is decided. If the extension flag is set, the instruction passes through the Extension register. General instructions are sent on to the decoder unit.

3.3.4. Decoder

The Instruction Decoder Unit decodes the general instruction and also controls the pipeline, by changing the machine status.

3.3.5. Registers

All the registers used in the simulator core are 32 bits wide. The registers are mainly categorized into General Purpose Register File (GPR) and Special Purpose Registers (SPR). AE32000 simulator has 16 General Purpose Registers. These GPRs are used to store any general data and temporary value. The Special Purpose Registers are also 32-bit registers that provide the machine status. These are

- Status Register (SR) - Indicate the status of machine operation.
- Program Counter (PC) - Specifies the instruction address that is to be executed next.
- Link Register(LR) - Register for holding the subroutine return address.
- Extension Register (ER) - Register used for extending immediate value or address displacement for 32-bit instructions.
- Stack Pointers (SPs) are used to store a temporary data in memory.
  - OSI Stack Pointer (ISP)
  - Supervisor Stack Pointer (SSP)
  - User Stack Pointer (USP)

3.3.6. Arithmetic and Logical Unit

The Arithmetic Logic Unit has 2$\times$32 bit inputs and 1$\times$32 bit outputs. The inputs are the ABUS and BBUS. The output is available at the CBUS. This block runs the following basic arithmetic and logical operations:

- Monomial operations
  - NOT : Inversion
  - CVx : Convert to (bit/short) with mask
- Binominal operations
  - Logical operation of AND, OR, XOR etc.
  - Arithmetic operation of ADD/SUB and so on
  - Comparison operation such as CMP, TST

The Multiplier unit is a block outside the ALU. The Multiplier executes a multiplication of 2×32-bit inputs and the output is 64-bit wide. It can execute in a single clock cycle and does not effect any status registers. This multiplier block uses two GPRs to store a result. Using the Multiplier Unit for a multiply and add (MAC) instruction produces a 64–bit result and stores the result in R14 and R15 registers.

3.4. Simulator Structure

Figure 3.2 gives a general outline of the AE32000 simulator. Most of the performance core is optional. It is optional in the sense the most of cache parameters or the statistics gathering parameters can be added, modified or removed using the configuration file. The simulator already has a default configuration file. Modifications can be made to this file. A more detailed explanation of the configuration file is given in the sections 4.8.

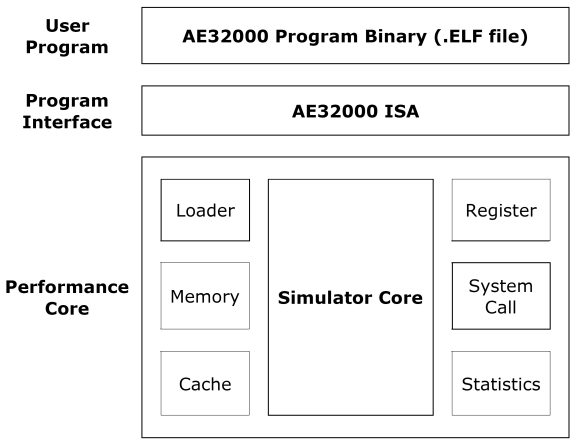

FIGURE 3.2. Simulator Structure.

3.4.1. Architectural Hierarchy of the Core

Figure 3.3 shows the overview of how all the files are organized. This is followed by detailed explanation of all the major files.

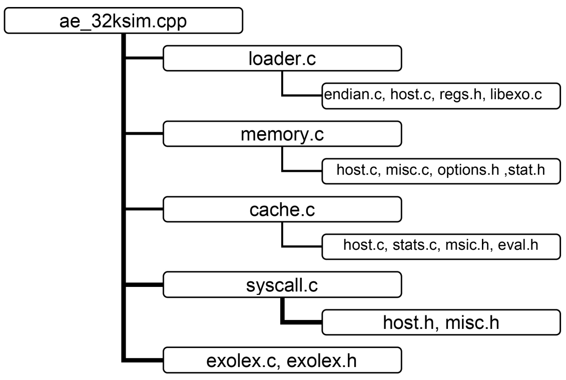
ae_32ksim.cpp
loader.c
endian.c, host.c, regs.h, libexo.c
memory.c
host.c, misc.c, options.h ,stat.h
cache.c
host.c, stats.c, msic.h, eval.h
syscall.c
host.h, misc.h
exolex.c, exolex.h

FIGURE 3.3. Simulator Hierarchy.

4. ENHANCED MODULES

The original version of the simulator had only the very basic features. The original version did not have any memory and cache modules. There were a very few options of gathering the statistics. With of the addition of the memory, cache, loader, system call modules into the simulator, a lot of changes were made in the main file of the simulator to accept the modules.

4.1. ae3ksim.cpp

This file is the main file in the simulator. This file controls most of the blocks in the simulator including the data flow and control. All the modules such as the loader, registers, memory, cache and statistics modules are initialized in this file. The main function takes in the parameters `argc` and `argv`, which includes the files that are simulated along with the input and output files. The configuration file used to initialize cache parameters is optional.

```
int main(int argc, char **argv, char **envp) {
        int size;
        int judge;
```

The input parameters for the file are usually the executable followed by the file to be simulated (elf file) followed by the input and output files (input and output files depend on the simulation program) and then finally the configuration file.

```
./ae32000 enocode.elf clinton.pcm out.adpcm -config default.cfg
```

Before beginning any job, simulator first initializes all the modules in the simulator. This is done using the function `sim_init()`. This function checks for the configuration file, initializes the register file, initializes the memory and resets all the statistic gathering modules.

Once the parameters are read in, the main module calls loading function. At this point the simulator needs only the parameters required to simulate the program. The rest of the input parameters like the simulator binary executable (`ae32000`), configuration file (`default.cfg`)are not used.

```
sim_load_prog(argv[1], argc-1, argv+1,envp);
```

After loading the file in the memory and initializing all the parameters required, the simulation of the program is started. The simulation enters the 5-stage pipeline. This module has **I**nstruction **F**etch (IF), **I**nstruction **D**ecode (ID), **Ex**ecution (EX), **Mem**ory Access (MEM) and **W**rite **B**ack (WB). This module also includes the pipe update function, which is called after WB. The pipeline when executed for the first time starts in the reverse order to clear the pipeline data path. Also the register files are written in the first half of the clock cycle and read in the second half of the clock cycle to remove any hazards in the WB stage.

The function used for the instruction decoding is "`decoder()`". The instruction decoding for a simple load instruction is shown here.

```
int decoder(uint16 ir) {
    int instr;
    pipeline[ID].ir = ir;

    if ((ir >> 12) == 0)
    {
        instr = LD;
        pipeline[ID].fp = call_ld;
        }
```

The data used in the decoder module comes in from the instruction register. Depending on the instruction and how the instruction is categorized (for example ALU, LOAD/STORE), the instruction is shifted and checked for its code.

Each Instructions' code is broken down to 4 parts. Each part performs a stage of the pipeline, `case ID` has all the code needed for the instruction to pass through the ID stage. All the instructions are called using `call_instructionname`. For example, load instruction is called using the function `call_ld()`. The code for the load instruction is shown here.

```
    void call_ld(int stage) {
        uint32 rdata;
        int mem_wait;
        switch (stage)
        {
            case ID :
                regs.regs_R[ABUS] = r0_set(0);
                regs.regs_R[BBUS] = r1_set(NA);
                regs.regs_R[IMM_ER] =
                imm_gen(regs.regs_R[ER], get_sr(EF),
                        4,  //er sft
                        4,  //offset_pos
                        2,  //offset_len_er
                        4,  //offset_len
                        2,  //offset_pos_aft
                        0   //kinds of extension
```

```
             );
        widx_set(8);
        spidx_set(NA);
        clr_sr(EF);
        break;
    case EX :
        regs.regs_R[MAR] = alu_func(regs.regs_R[ABUS],
                       regs.regs_R[IMM_ER], 0,OP_ADD);
        break;
    case MEM :
            mem_wait =
              dmem_read(regs.regs_R[MAR], mem, &rdata);
            if (mem_wait == 0) {
                stat.mstall = 0;
                regs.regs_R[LMD] = rdata;
            } else
                stat.mstall = 1;
                regs.regs_R[CBUS] = regs.regs_R[LMD];
                break;
    case WB :
            regs.regs_R[pipeline[WB].widx] =
                                    regs.regs_R[CBUS];
            num_inst_ld++;
            break;
    }
}
```

As the simulator starts, each stage in the pipeline is called using the functions `if_action()`, `id_action()`, `ex_action()`, `mem_action()`, `wb_action()` and `pipe_update()`. During the execution, forwarding of data at every stage is checked and handled by the function `forwarding()`. All the cases of forwarding are considered in this function.

The statistics module is also controlled using the function `report_stat()`. The statistics gathering module is powerful and user friendly so that the user can stop the simulation at a desired cycle, gather simulation statistics for the next "n" desired cycles and then later switch off the statistics module and proceed with the normal execution.

The other major functions used in the main file are the instruction memory read and data memory write functions. There are also functions for LERI folding unit, ALU, setting the values in the status register.

The function used to read in data from the instruction memory is `imem_read()`. This function can read in either one 16 bit instruction or directly 32–bit instruction (two 16 bit instructions) from the memory. This function has built in cache module. In case the data is missed in the cache this information is sent to the cache statistics module.

```
int imem_read(uint16 *intr,
               struct mem_t *mem,unsigned int addr)
or
int imem_read(uint16 *instr0, uint16 *instr1,
              struct mem_t *mem, unsigned int addr )
```

Data is written into the data memory using the function `mem_write()`. This function uses three other functions to write in data depending on the size the of data: BYTE, HALF WORD or WORD.

```
int mem_write(uint32 addr, uint32 wdata,
              struct mem_t *mem, int size) {
    ... ...
    ... ...
    ... ...
    MEM_WRITE_WORD(mem, t_addr, wdata);
    ... ...
    MEM_WRITE_HALF(mem, t_addr, wdata);
    ... ...
    MEM_WRITE_BYTE(mem, t_addr, wdata);
    ... ...
```

A similar function is used to read in the 32–bit data from the memory. The function is called `mem_read()`.

```
int dmem_read(uint32 addr,struct mem_t *mem,uint32 *rdata) {
```

In case the data required to read in from the data memory is only a BYTE or HALF WORD, this is taken care by the `LMD_EXT()` function.

Arithmetic and Logical Operations are handled by the function `alu_func()`, which takes in two 32–bit data and operation to be performed such as add, sub etc., the output is 32–bit data. The data sent as inputs to the ALU are through ABUS and BBUS. The output is collected at the other end of the ALU at CBUS. The ALU is not capable of performing multiplication operation.

```
int alu_func(uint32 a, uint32 b, int c, int operation) {
        int32 result;
```

Therefore, there is a separate multiplier unit that is capable of performing multiplication of two 32–bit numbers to generate a 64–bit data in just one clock cycle.

```
int64 multiplier(int32 a, int32 b, int operation) {
        uint32 ua;
        uint32 ub;
        int64 result;
        if (operation == OP_MUL)
        {
            result = (int64)a * (int64)b;
        }
        else if (operation == OP_UMUL)
        {
            ua = a;
            ub = b;
            result = (uint64)ua * (uint64)ub;
        }
return result;
}
```

The other functions used in the simulator core are pretty straightforward and do not need much explanation.

4.2. loader.c

This file has the loader module in it. The earlier module in the simulator was not capable of loading an ELF binary. When a program is compiled using AE32000 cross–compiler, we get two binary files one with the extension “`.elf`” and the other with the extension “`.bin`”. The loader function in the original AE32000 was able to load only the binary file into a flat memory space, and could not handle an ELF binary file. Thus, the main reason to change the existing loader function was to make the loader load an ELF (Executable and Linkable Format) binary file. The reason to use the ELF binary over a ordinary `bin` format binary is because of the many advantages like simplifying the task of making shared libraries and also this enhances dynamic loading of modules at runtime. A much more detailed explanation of ELF is given in APPENDIX A.

4.3. regs.c

The purpose of this file is to generate and initialize 32-bit registers.

```
/* create a register file */
struct regs_t * regs_create(void) {
    struct regs_t *regs;
    regs = calloc(1, sizeof(struct regs_t));
    if (!regs)
            fatal("out of virtual memory");
    return regs;
}
```

This part of the code is used to initialize the register file created.

```
/* initialize architected register state */
void regs_init(struct regs_t *regs)
/* register file to initialize */ {
memset(regs, 0, sizeof(*regs));
}
```

Data is stored as little endian, where LSB is stored at the lowest address location.

4.4. memory.c

The memory module of the AE32000 is very similar to the Simplescalar module. The memory module generates a memory space similar to the memory of any host machine. A flat memory space is created by the memory module.

```
/* create a flat memory space */
struct mem_t *
    mem_create(char *name)   /*name of memory space */
{
    struct mem_t *mem;
    mem = calloc(1, sizeof(struct mem_t));
    if (!mem)
            fatal("out of virtual memory");
    mem->name = mystrdup(name);
    return mem;
}
```

The memory module has function to translate the address of the simulated memory with the host memory. To do this, memory uses the function `mem_translate()`. To make the memory look like the host memory, the module uses `mem_newpage()`. All these pages are allocated on demand. The memory module uses pages and an inverted page table to keep track of its own memory usage, but memory is accessed by virtual addresses. The module also has functions to check the alignments and permissions, handles any natural transfer sizes; note, faults out if `nbytes` is not a power-of-two or larger than `MD_PAGE_SIZE`. And to do all this, the module uses a generic memory access function called `mem_access()`.

```
enum md_fault_type
mem_access(struct mem_t *mem, /*memory space to access*/
             enum mem_cmd cmd, /*Read (from sim mem) or Write*/
             md_addr_t addr, /*target address to access*/
             void *vp,       /*host memory address to access*/
             int nbytes)     /*number of bytes to access*/
```

The other functions that are used to access the data are `mem_strcopy()`, `mem_bcopy()`, and `mem_bcopy4()`. Depending on the size of the data to be transferred, different functions are used.

This module generates a virtual memory address space of 231 bytes. This memory is mapped from `0x00000000` to `0x7FFFFFFF`. Address space from `0x00000000` to `0x003FFFFF` is not used. The address space from `0x00400000` to `0x10000000` is used to map the program text (code), and accessing any memory outside of the defined program space causes an error to be declared. The address space from 0x10000000 to "`mem_brk_point`" is used for the program data segment. This section of the address space is initially set to contain the initialized data segment and then the uninitialized data segment.

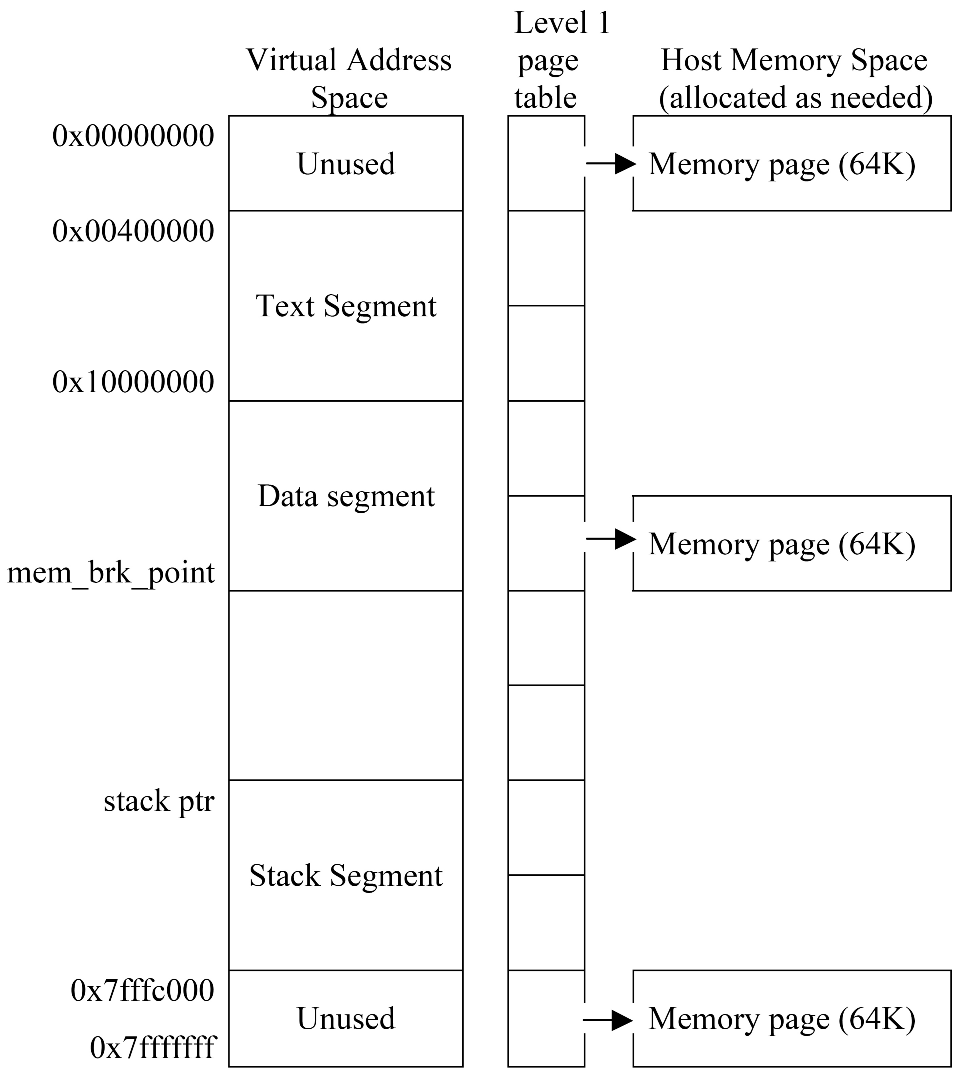

FIGURE 4.1. Memory Map.

The data segment can continue to expand until it collides with the stack segment. The stack segment starts at `0x7FFFC000` and grows to lower memory as more stack space is allocated. Initially, the stack contains program arguments and environment variables. The stack may continue to expand to lower memory until it collides with the data segment.

The virtual memory address space is implemented with a one level page table, where the first level table contains `MEM_TABLE_SIZE` pointers to `MEM_BLOCK_SIZE` and byte pages in the second level table. Pages are allocated in `MEM_BLOCK_SIZE` size chunks when first accessed; the initial value of page memory is all zero. The memory map is currently hard-coded into the loader script because the information is not in the binary file.

4.5. cache.c

The program that implements the cache functionality is called cache.c. There are both instruction and data cache. This module not only generates a cache but also has the capability of collecting all the cache statistics including number of misses/hits, miss/hit rate and also the replacement rate. The cache parameters can be set from the configuration file. Both caches can be configured in the same style, i.e., a colon delimited string of 4 values `<nsets>:<bsize>:<assoc>:<rpolicy>`.

- *nsets* is the number of sets in the structure, it must be a positive integer and a power of two.
- *bsize* is the block size, it also must be a positive integer and a power of two.
- *assoc* is the associativity, which must be a positive integer.

- *rpolicy* is the replacement policy and is either **l** (LRU), **r** (random) or **f** (FIFO).

The total size of the structure in bytes is the product of the associativity, block size and number of sets. For example, to configure the simulator with a 2–way set associative, 32KB first level data-cache, with 32B blocks and random replacement, use the following line:

```
-cache:dl1 512:32:2:r
```

If user does not want to include a certain structure in the hierarchy, replace the configuration string with the string none. For instance, this is the method to simulate a machine without a data cache:

```
-cache:dl1 none
```

The way to simulate a unified structure in this cache module (i.e., instructions and data living in the same structure) is to point the instruction structure to the corresponding data structure. For example, to implement a unified first–level cache, use:

```
-cache:dl1 512:32:2:r -cache:il1 dl1
```

To specify a direct-mapped, 32KB first-level data cache with 32B lines, LRU replacement, write through policy and a 2 item victim buffer, we use:

```
-cache:dl1 1024:32:1:l

-cache:dl1:wthru true

-cache:dl1:vb 2
```

After initialization, simulator interacts with the cache module using the function cache_access. `cache_access()` has the following signature:

```
unsigned int
cache_access(cache_t *cp,
             enum mem_cmd_t cmd,
             md_addr_t addr,
             unsigned int nbytes,
             tick_t now,
             miss_handler_t miss_handler)
```

Here is the explanation for parameters:

- *cp* is a pointer to the particular cache structure
- `cmd` is the operation you are trying to perform, the value of `cmd` is either Read or Write
- *addr* is the address which is being accessed
- *nbytes* is the number of bytes which are getting accessed
- *now* is the cycle number in which you are performing the operation. The cache module is also used in timing simulation, so internally it is able to keep track of how long things take.
- *miss_handler* is a pointer to a function that cache access will call internally whenever it needs something from the next level of the memory hierarchy. This function is explained later.
- *cache_access* returns an unsigned integer that is the latency of the operation.

Basically, miss handler is like cache access but without the `cp` pointer and its own `miss_handler` pointer. `cache_access` calls `miss_handler` internally whenever it needs to do something to the lower level of the memory hierarchy. This could be a read (if there is a miss), or a write (if there is a dirty replacement or a write-thru).

```
unsigned int
miss_handler(enum mem_cmd_t cmd,
             md_addr_t addr,
             unsigned int nbytes,
             tick_t now)
```

4.6. syscall.c

syscall.c implements the module for handling system calls. The system emulates the system call by translating it to an equivalent host operating-system system call and directing the simulator to execute the call on the simulated programs behalf. For example, if the simulated program attempts to open a file, this module translates the request to a call to `open()` and returns the resulting file descriptor or error number in the simulated programs registers.This module executes a system calls of the by translating system calls to the corresponding calls on the host operating system (Linux). The parameters that connects the main simulator file(`ae32k_sim.cpp`) and syscall module are `FUNC`, `PARM1`, `PARM2`, `PARM3`. Register 8 holds `FUNC`, `PARM1` is in Register 9, and next two parameters are stored in stack (`SP + 12`), (`SP + 16`).

4.7. stats.c

stats.c is used to collect statistics from various modules like cache, memory and loader. The `stat_register()` interface registers the simulator variables, instructions and cycles with the statistical module. This statistical package tracks updates to statistical counters, producing on request a detailed report of all model instrumentation. The `stat_formula()` interface allows derived instrumentation to be declared, creating a metric that is a function of other counters. In Following code, miss rate denotes a derived statistic equal to the number of misses executed divided by the total number of accesses.

```
stat_reg_formula(sdb, buf, "accesses", buf1, "%12.0f");
stat_reg_counter(sdb, buf, "misses", &cp->misses,
                    cp->misses, NULL);
sprintf(buf1, "%s.misses / %s.accesses", name, name);
stat_reg_formula(sdb, buf, "miss rate (i.e., misses/ref)",
                    buf1, NULL);
```

The main advantage of adding this file is that it has specific functions to calculate number of times an instruction has occurred, rate of any statistical value for example miss rate, hit rate. These functions can be used independently in all the other major modules where user needs to gather stats.

4.8. default.cfg

Here is example default.cfg file.

```
-cache:il1      il1:256:32:1:l # l1 inst cache config
-cache:il1lat   1       # l1 inst cache hit latency
```

```
-cache:il2      dl1          # l2 inst cache
-cache:il2lat   5       # l2 inst cache hit latency
-cache:dl1      dl1:256:32:1:l  # l1 data cache config
-cache:dl1lat   1       #l1 data cache hit latency
-mem:lat        10      # memory access latency
```

4.9. Problems integrating modules

The major problem with the integration of the modules for loader, memory, cache and statistics was all the modules were written in C code. Most of functions in the header files of the modules were getting accessed when called from the main function in the simulator file(`ae32k_sim.cpp`). To avoid this most of the function were transferred from the header files into the header files on the main simulator header file (`ae32k_sim.h`).

4.9.1. Problem with loading the binary

The first problem during the integration of the loader for ARM ELF binary, which was primarily written for a simplescalar compiler, is the value of `EM_ARM`. This value varies from compiler to compiler. `EM_ARM` defines for which cross-compiler the binaries are generated. Changes are done to this arm value so that the loader accepts binaries from AE32000 compiler. For simplescalar it is `40` and for AE32000 compiler it is `44482`. This value can be changed in the file loader.h.

4.9.2. sim.vct

The simulator was not capable of accepting the parameters from the command line. The simulator used to have a configuration file which has all the parameters to be passed into the simulator like input binary file (the input is a bin file not an ELF file), instruction memory base and instruction memory bound. The present simulator removes the idea of a `sim.vct` file and directly reads in command line parameters. All the parameters like the instruction memory base, instruction memory bound etc., are all hard coded in the loader module.

5. BUGS IN SIMULATOR

This chapter describes the bugs that were found in the simulator, problems faced because of these bugs and the efforts made to clear them out

5.1. Multiplication Result of two 32-bit numbers

One of the first bugs to be detected in the simulator was that, the multiplication of two 32-bit numbers was not producing the correct result in some cases. The reason to say, in some cases, is because for some 32-bit multiplication C code, the compiler was smart enough to change multiply as a loop of add functions. This bug was found when simulating Dhrystone benchmark.

```
a *= b;
```

For multiplication used in the above piece of code, the AE32000 compiler always generates a signed multiply float instruction (`mulsf`). Since this instruction is not defined in the instruction set, a macro is called in to perform the operation. However, the upper 32 bits of the returned result of was always set to "`FFFF FFFF`", which is incorrect. Therefore, the following code was used to replace the earlier code in the simulator.

To correct this error was simple but the main problem was to detect this error. This bug never prevented the benchmark to complete execution. The results produced by this simulator and gcc were different. The following code was used to replace the earlier code.

```
result = ua * ub;
```

is replaced by

```
result = (uint64)ua * (uint64)ub;
```

5.2. Forwarding Path for jump Instructions

Of the many Jump instructions in the AE32000 simulator jump register (`JR`) and jump and link register (`JALR`) instructions get the address to jump from a register. For these instructions, checking for a forwarding path was not accounted for. The forwarding path is inserted in all the three functions to ensure the proper functioning of the simulator. This part of the code checks for any dependencies and sets the flags and arranges data paths for forwarding.

5.3. Missing stall for load followed by JR and JALR instructions

A stall or a bubble in inserted in the pipeline in case of load followed by an immediate use. Any jump instruction in this simulator is designed to calculate the **B**ranch **T**arget **A**ddress in the instruction decode stage. Thus, if there is a load followed by any of these two jump instructions (`JR`, `JALR`), there has to be two NOP instructions or stalls in the pipeline. This is because the load instruction gets the data only in the memory stage.

5.4. Problems with compiler

Some very serious bugs were encountered during the compilation of various benchmarks.

5.5. bug in `for` loop

One of the interesting bug in the compiler was found in a `for` loop in EPIC benchmark. For the following piece of C code, the compiled binary was generating `JGE` instruction to check the condition statement in the for loop.

```
        for (first_time_flag =1;
            the_tag & ~BIN_TAG_INFO)
            {
            ... ...
            }
```

Generated binary code for the above C code

```
            condition checking
            code is here
            jge 488<.LM7>
            ... ... //loop code
            ... ... // is here
        <.LM7>
```

The value of `the_tag & BIN_TAG_INFO` is a non-zero value. `JGE` checks this non-zero value and always exits the loop. Compiler was generating a faulty code in this case, instead of going for `JGE` instruction, compiler was supposed to put in a `JZ` instruction. The C code was tweaked so the compiler generates the following code and the benchmark ran to completion without any problem.

```
            condition checking
            code is here
            jz 488<.LM7>
            ... ... //loop code
            ... ... // is here
        <.LM7>
```

Now the value of (`the_tag & BIN_TAG_INFO`), which is a non zero gets executed properly because of `jz` instruction.

5.6. cross compiler

The version of the cross-compiler provided to us, was built using a very old version of gcc (`gcc 2.9.1`). Many problems arose when compiling the other benchmarks present in the mediabench suite. There were a lot of libraries missing in the older version so there are problems compiling latest benchmarks.

6. BENCHMARKS

For a simulator to pass the reliability test it should undergo rigorous benchmarking. The original version of AE32000 passed the test for simple C program like hello.c, matrix multiplication, but was not able to provide correct results for real time application specific benchmarks. In most of the cases the simulation never ran to completion. Rigorous benchmarking using the Mediabench benchmark helped us the find various bugs present in the simulator as well as the compiler.

6.1. Mediabench

Mediabench is a benchmark suite composed of multimedia programs. Its main aim is to benchmark architectures for a multimedia utilization, as opposed to SPEC benchmarks [23] for example. Mediabench is composed of complete applications coded in high-level languages. All of the applications are publicly available, making the suite available to a wider user community. This is a type of benchmark that have a very high percentage of core contribution. The term core is defined when a set of loops whose execution time higher than the threshold value. The core contribution of mediabench benchmark is around 90%. A precise set of input test files are selected for each application. The simulator is tested for the following set of benchmarks in mediabench.

- `ADPCM` stands for "Adaptive Differential Pulse Code Modulation". This is one of the simplest and oldest forms of audio coding. This is a variation standard Pulse Code Modulation (PCM). A common implementation takes

16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

- `EPIC` is a wavelet-based image compression codec. The compression algorithm is based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters are designed to allow extremely fast decoding without floating-point hardware.
- `G.721` is a reference implementation of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions.
- `JPEG` is a standardized compression method for full-color and gray-scale images. It is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents. JPEG is lossy. Two applications are derived from the source code: `cjpeg` does image compression and `djpeg`, which does decompression. The input data can be either a large or small color image.
- `MPEG2` is the current dominant standard for high-quality digital video transmission and is also used for DVDs. The important computing kernel is a motion estimation for coding and the inverse discrete cosine transform for decoding. The two applications used are `mpeg2enc` for encoding and `mpeg2dec` for decoding.

6.2. Instruction Profiling

Instruction level profiling can be tuned in to provide useful information regarding the percentage of instruction executed in a program. Instruction profiling is classified into two types compilation based and simulation based profiler. The earlier one instruments the program by adding counters to various basic blocks of the program. The later one can be further classified into static and dynamic profilers. Simulation based static profiling is written to a trace and trace is processed to get the instruction count. Dynamic profilers obtain the instruction profile during the execution of the simulator. Though this method is slow when compared to compiler based but this method various architectural parameters can be tuned. AE32000 has many analysis routines that collect information at various parts of the program and dump the results into a separate files. AE32000s modules sample instructions as they move through the pipeline and report statistics like cache miss/hit rates. Modules annotates each instruction that reads and writes memory.

6.3. Results

Results gathered from the benchmarks gives a plethora of information regarding the simulator and the changes that can be made to further improve the simulator. To get a better understanding of the instruction set we compared two simulators Simplescalar-ARM and AE32000. Simplescalar-ARM's superscalar functionality is completely disabled and it functions like a proper 5–stage simple pipeline structure. Both the simulators are set to provide the greatest similarity in the 5-stage pipeline. All the Benchmarks when compiled are set to the same optimization level `(-O3)`.

Table 6.1 gives a general idea of how the benchmarks perform on both the simulators. Number of cycles required to complete the execution of each benchmark is given in the table. From the data shown, AE32000 is much slower when compared to Simplescalar-ARM.

	AE32000 *Clock Cycles*	*ARM* *Clock Cycles*
ADPCM	26,577,300	17,199,924
EPIC	7,468,254,770	107,765,075
G721	889,841,000	1,143,142,179
JPEG	165,001,000	32,043,419
MPEG2	111,669,128,844	17,66,823,273

TABLE 6.1. Number of Cycles.

Table 6.1 show the number of cycles required for each benchmark to execute.

The difference in the number of clock cycles required is primarily due to difference in the number of instructions executed for each benchmark. Table 6.2 shows the number of instructions executed for each benchmark. The reasons for the increased number of instructions in AE32000 compared to Simplescalar-ARM are due to the following:

- Shorter instruction length in AE32000.
- Compiler optimization between the two cross compilers.
- Problems due to instructions such as Push/Pop, Branch, NOP (explained further in the subsequent subsections).

6.3.1. Instruction Count

Two different sets of benchmarks are studied, because of the differences in the generated instruction set by the compiler. ADPCM, G.721 and JPEG

	AE32000	*ARM*
	Clock Cycles	*Clock Cycles*
ADPCM	22,980,455	14,787,963
EPIC	6,869,787,870	71,083,219
G721	657,787,782	755,955,225
JPEG	25,766,785	10,905,456
MPEG2	104,006,775,804	1,172,145,689

TABLE 6.2. Instruction Count.

use only integer instruction set, while MPEG and EPIC use floating point (FP) instructions. Table 6.2 shows the instruction count results. As can be seen, the total number of instructions executed by AE32000 drastically increases for FP benchmarks. The number of instructions increases by factor of **88** and **98** for MPEG and EPIC, respectively. The reason for this is because the two benchmarks require a large number of FP computations. However, AE32000 instructions set does not have any support FP instructions. So, AE32000 compiler substitutes a macro-level function for every floating-point instruction. On an average, each macro incurs an extra overhead of 250-275 instructions.

In benchmarks where there are few FP multiplications or divisions there is a little effect. In case of real time applications, such as MPEG and EPIC, this difference in the number of cycles and instructions is significant.

	Memory Access			
	AE		SS	
ADPCM	2,835,644	12.34%	2,813,210	19.02%
G.721	125,340,710	16.09%	168,312,725	22.26%
JPEG	5,571,797	21.62%	3,744,334	34.33%
EPIC	1,980,075,588	25.01%	22,519,935	31.68%
MPEG	27,033,833,028	25.99%	368,444,456	31.43%

TABLE 6.3. Memory Access.

	Loads				Stores			
	AE		SS		AE		SS	
	Count	%	Count	%	Count	%	Count	%
ADPCM	2,140,173	9.31	2,072,306	14.01	695,471	3.03	740,904	5.01
G.721	102,103,425	13.10	126,423,713	16.72	23,237,285	2.98	41,889,012	5.54
JPEG	3,726,388	14.46	2,671,540	24.50	1,845,409	7.16	1,072,794	9.84
EPIC	1,276,491,243	16.12	16,506,825	23.22	703,584,345	8.89	6,013,110	8.46
MPEG	17,914,963,386	17.22	324,567,259	27.69	9,118,869,642	8.77	43,877,197	3.74

TABLE 6.4. Loads and Stores.

6.3.2. Memory Access

Table 6.3 shows the number of memory accesses executed by both simulators. AE32000 has more number of memory accesses compared to Simplescalar-ARM.

The number of loads and stores for most of the cases is almost similar for both simulators. Only in cases for JPEG there is a difference in the compared results 6.4.

6.3.3. LERI

A very important feature of the AE32000 is the LERI (Load Extension Register Immediate) unit. AE32000 has fixed 16-bit instruction length, so for operations that require long immediate values, assistance is provided by LERI. The format of this instruction is 2-bit opcode and 14 bit immediate value. LERI instruction occurs in the following cases:

- Load immediate values

```
leri    0x1453      (0x1453)
ldi 0x1453E  %R9
```

- Load data into a register (when the value of offset is a large value)

```
leri    0x12 (0x12)
ld  (  %R2  +  0x12C )  %R8
```

- Store data to a register (when the value of offset is a large value)

```
leri    0xC         (0xC)
st  %R9  ,  ( %R10  +  0xC4)
```

LERI always adds an overhead to the performance of AE32000. The reason for this is a short instruction length. LERI accounts to around 5% of instructions [Table 6.5].

6.3.4. ALU Instructions

The results in Table 6.6 compare the Arithmetic and Logical Unit (ALU) operations. For AE32000 the number of ALU instructions executed for ADPCM

	LERI	
	AE	SS
ADPCM	16,655	0
G.721	79,625,014	0
JPEG	329,394	0
EPIC	524,107,668	0
MPEG	5,808,043,825	0

TABLE 6.5. LERI.

	ALU Instructions		FLOATING POINT	
	AE	SS	AE	SS
ADPCM	8,590,540	11,008,856	0	0
G.721	408,722,389	483,533,717	0	0
JPEG	12,707,420	6,321,881	0	0
EPIC	2,674,899,505	32,678,211	0	6,580,452
MPEG	33,246,065,965	674,767,471	0	52,453,399

TABLE 6.6. ALU.

and G.721 are smaller by 28% and 18%, respectively. AE32000 has a bigger and better ALU instruction set than Simplescalar-ARM. AE32000 has extra instructions such as LESA, LSEA, SMUL, EXTB, EXTS, CVB and CVS when compared to Simplescalar-ARM. These special instructions account for an average for 6-9% of ALU operations.

However JPEG has nearly twice the number of ALU operations. This is because of a few missing instructions in AE32000. Specific negative instructions like compare negative immediate, re-verse subtract immediate etc., are not present in AE32000. The operations of these instructions are of the form

Compare Negative:

```
CMN<suffix>  <op 1>, <op 2>
```

```
status = op_1 - (- op_2)
```

Reverse Subtract:

```
RSB<suffix>  <dest>, <op 1>, <op 2>
dest = op_2 - op_1
```

The other advantage of Simplescalar-ARM over AE32000 in case of the JPEG benchmark is the usage of Shift instructions in conjunction with add or subtract commands. Shift instructions account for about 5.5% to 11% of the total instruction count. Simplescalar-ARM provides this functionality by carrying out the shift operations as a part of other instructions.

Example of ARM Shift:

```
ADD r1, r2, r3, LSL#2
operation r1 = r2 + 4 * r3
```

In this instruction, LSL is for Logical Shift Left by number of bits specified in the immediate value. This single instruction can be executed in a single clock cycle. Other instructions similar to this are LSR, ASR, ASL, ROR, and RRX. All these instructions helped Simplescalar-ARM to reduce the total instruction count in JPEG benchmark.

When the FP instruction benchmarks EPIC and MPEG are considered, the presence of missing FP instructions is clearly visible by the huge number of operations executed by ALU in AE32000 simulator.

6.3.5. Branch

Major advantage Simplescalar-ARM has over AE32000 is the usage of branch instructions. In-side every instruction in Simplescalar-ARM there is con-

	Branch/Jump			
	AE		SS	
ADPCM	6,041,253	26.29%	965,277	6.53%
G.721	123,724,683	15.88%	104,108,783	13.77%
JPEG	1,601,227	6.21%	839,138	7.69%
EPIC	1,180,389,963	14.91%	9,304,461	13.09%
MPEG	12,243,239,149	11.77%	76,478,036	6.52%

TABLE 6.7. Branch/Jump.

dition testing. The conditions are tested for zero, negative, carry or overflow flags in the status register. All these four bits are embedded in each instruction (bits [31:28]).

Example of Embedded Branching:

```
CMP r0, #0
BEQ 20002dc
```

During execution of this piece of code in Simplescalar-ARM, the processor compares the condition bits to the present state of the status register and determines if the instruction should be executed. In case the condition within the status register is not satisfied, then the instruction is not executed and it is converted into a NOP (no operation) and a bubble is inserted it the pipeline. For integer only benchmarks the number of branches executed by AE32000 in an average is three times larger compared to Simplescalar-ARM. In case of Floating point benchmarks number of executed by AE32000 are 140 times more than Simplescalar-ARM [Table 6.7].

	PUSH/POP		
	AE		SS
ADPCM	4,829,578	21.02%	0
G.721	7,689,476	0.99%	0
JPEG	2,953,961	11.46%	0
EPIC	1,034,421,430	13.06%	0
MPEG	12,840,950,937	12.35%	0

TABLE 6.8. PUSH/POP.

6.3.6. PUSH/POP

The reason behind the large number of PUSH/POP instructions in the AE32000 simulator, com-pared to Simplescalar-ARM, is due to many subroutines generated by the AE32000 compiler in the assembly code. Each time a sub-routine is called in the AE32000 simulator, PUSH/POP instructions are called in to store the contents of the registers on to stack and retrieve them later [Table 6.8].

6.3.7. NOP

The inclusion of so many NOP instructions in the AE32000 simulator execution is also responsible for the huge cycle count, when compared with Simplescalar-ARM. A compiler optimization may be needed to remove these un-wanted NOP instructions [Table 6.9].

6.3.8. SYSCALL

The main reason syscalls occurs in all the five benchmarks is when any simulator wants to open, close, read and write into files. Reads and Writes are

	NOP			
	AE		SS	
ADPCM	4,829,578	2.90%	320	0.00%
G.721	7,689,476	4.38%	450	0.00%
JPEG	2,953,961	10.10%	430	0.00%
EPIC	1,034,421,430	6.62%	6234	0.01%
MPEG	12,840,950,937	12.34%	300246	0.03%

TABLE 6.9. NOP.

	SYSCALLS	
	AE	SS
ADPCM	670	570
EPIC	1384	160
G.721	1830	25
JPEG	236	103

TABLE 6.10. System Call.

the main reasons for the syscall count given in Table 6.10. The main reason for huge number of system calls in AE32000 is due to the limit of 400 bytes set by the compiler. So incase AE32000 wants to write 1586 bytes of data into the output file, write system call is called 4 times. There is no such limit set in the compiler for simplescalar-ARM.

7. CONCLUSION

Compared to AE32000, Simplescalar-ARM is competitive. From the results we get the following conclusions.

- The need for the reduction of number of memory accesses for a embedded process is one of the major concern. Results show AE32000 use comparable number of Loads/Stores.

- For completing ALU operations, AE32000 does a better job. But when it comes to FP instructions, it fails. Instead of adding in extra FP instructions it is better to add in a co-processor that can support FP instructions. Doing this saves in the extra floating point registers and other hardware that should be placed in the processor.

- Work needs to be done on the modules such as SYSCALL, PUSH/POP, NOP. Improving these modules can show a significance improvement.

- Developing a cross compiler with a new version of the gcc and linking libraries. The compiler can be updated with a next version for better optimization. This will help in reordering the instructions so as to remove number of hazards and stalls and will generate more optimized assembly code.

- To change the instruction length from 16–bit to 32–bit is also not advisable. The fixed 32-bit instruction length negatively affected code density of programs for 32-bit RISC processors. Having a 16 bit instruction length simply localizes the program code except for handling addresses and immediate values.

As the definition for embedded processor states, embedded technology is developed for application specific purpose. AE32000 processor if used for ALU intensive applications, has added advantages. In case any extra feature is needed this embedded processor can be assisted by a co-processor. AE32000 has a four channel co-processor interface for additional Multimedia, DSP and FP applications.

7.1. Future Work

This thesis work has provided a direction in the development for future embedded cores in EISC Technology. From the work presented here, opportunities for further research in this area is more useful. Instruction profiling provides with most of the data, but still there are many unexplored areas in the simulator.

- Detailed Profiling on modules.
- Options are open to include a debugger module for the simulator.
- Research on cache modules for improved performance is also suggested.

BIBLIOGRAPHY

[1] Nikil Dutt and Kiyoung Choi,“Configurable Processors for Embedded Computing,” *IEEE Computer Society Press*, vol. 36, pp.120-123, 2003.

[2] Manfred Schlett,“Trends in Embedded-Microprocessor Design,” *IEEE Computer Society Press*, vol. 31, pp.44-49, 1998.

[3] S. Wong and S. Vassiliadis and S. D. Cotofana ,“Future Directions of Programmable and Reconfigurable Embedded Processors,” *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, pp.235-257, January 2004.

[4] Massoud Pedram,“Power optimization and management in embedded systems,” *Proceedings of the 2001 conference on Asia South Pacific design automation*, 0-7803-6634-4, pp.239-244, 2001.

[5] 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?,” *Proceedings of International Conference on Compilers, Architectures, and Synthesis of Embedded Systems*, pp.176-180, November 2001.

[6] Vivek Tiwari, Sharad Malik and Andrew Wolfe,“Compilation Techniques for Low Energy: An Overview,” *Proceedings of Symp. Low-Power Electronics*, 1994.

[7] David Brooks, Vivek Tiwari and Margaret Martonosi,,“Wattch - A Framework for architectural-level power analysis and optimizations,” *Proceedings of The 27th Annual International Symposium on Computer architecture*, pp.83-94, 2000.

[8] Trimaran, http://www.trimaran.org.

[9] Advanced RISC Machines Ltd.,Cambridge, UK.

[10] Nathan Tennies,“Software Matters for Power Consumption,” *Embedded Systems Programming*, Jan 22 2003.

[11] Miyoshi, Akihiko and Lefurgy, Charles and Van Hensbergen, Eric and Rajamony, Ram and Rajkumar, Raj,“Critical power slope: understanding the runtime effects of frequency scaling,” *Proceedings of the 16th International Conference on Supercomputing*, pp.35-44, June 2002.

[12] Rainer Leupers,“Code generation for embedded processors,” *Proceedings of the 13th international symposium on System synthesis*, 1080-1082, pp.173-178, 2000.

[13] A.V. Aho, R. Sethi, J.D. Ullman,“Compilers - Principles, Techniques, and Tools,” *Addison-Wesley*, 1986.

[14] S.S. Muchnik,“Advanced Compiler Design and Implementation,” *Morgan Kaufmann Publishers*, 1997.

[15] A.W. Appel,“Modern Compiler Implementation in C,” *Cambridge University Press*, 1998.

[16] S. Gupta, R. Gupta, M. Miranda, F. Catthoor,“Analysis of High-Level Address Code Transformations for Programmable Processors,” *Design Automation and Test in Europe (DATE)*, 2000.

[17] M. Lam,“Software Pipelining: An Effective Scheduling Technique for VLIW machines,” *ACM SIGPLAN Conference on ProgrammingLanguage Design and Implementation (PLDI)*, 1988.

[18] G. Goossens, J. Vandewalle, H. De Man,“Loop Optimization in Register-Transfer Scheduling for DSP Systems,” *26th Design Automation Conference (DAC)*, 1989.

[19] R. Leupers, P. Marwedel,“Function Inlining under Code Size Constraints for Embedded Processors,” *International Conference on Computer-Aided Design (ICCAD)*, 1999.

[20] Alfred V. Aho and Mahadevan Ganapathi and Steven W. K. Tjiang,“Code generation using tree matching and dynamic programming,” *ACM Transactions on Programming Languages and Systems,*, vol. 11, No. 4, 1989.

[21] Christopher W. Fraser and David R. Hanson and Todd A. Proebsting,“Engineering a Simple, Efficient Code-Generator Generator,” *ACM Letters on Programming Languages and Systems,*, vol. 1, No. 3, September 1992.

[22] H. Emmelmann, F.W. Schroer, R. Landwehr,“BEG - a Generator for Efficient Back-Ends,” *Proceedings of the Sigplan' 89 Conference on Programming Language Design and Implementation,*, No. 7, June 1989.

[23] Standard Performance Evaluation Corporation. http://www.spec.org/

APPENDICES

APPENDIX A. Executable and Linkable Format(ELF)

The Executable and Linking Format (ELF), originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI), is rapidly becoming the standard in file formats. The Tool Interface Standards committee (TIS) selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems. ELF standard is growing in popularity because of its greater power and flexibility than the "a.out" and "COFF" binary formats. This standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments. This should reduce the number of different interface implementations, thereby reducing the need for recoding and recompiling code.

ELF now appears as the default binary format on operating systems such as Linux, Solaris 2.x, and SVR4. Some of the capabilities of ELF are dynamic linking, dynamic loading, imposing runtime control on a program and an improved method for creating shared libraries. The ELF representation of control data in an object file is platform independent, an additional improvement over previous binary formats. The ELF representation permits object files to be identified, parsed, and interpreted similarly, making the ELF object files compatible across multiple platforms and architectures of different size.

There are three main types of object files.

- A *relocatable* file holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable* file holds a program suitable for execution; the file specifies how exec (BA_OS) creates a program's process image.

- A *shared* object file holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

A.1. File Format

Object files participate in both program linking and program execution. For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities.

An ELF header is at the beginning and describes the file's organization. Sections hold the bulk of object file information for linking view like instructions, data, symbol table, relocation information, and so on.

A program header table tells the system how to create a process image. This header table is optional. Files used to execute a program must have a program header table but relocatable files do not need one.

A section header table describes all the file's sections. Every section has an entry in the table. Each entry gives information such as Section name, Section size, etc. Files used during linking must have a section header table. For other object files this is optional.

A.2. Data Representation

The object file format can support various processors with 8-bit and 32-bit architectures. It is extensible to both larger as well as smaller architectures. Object files represent control data with a machine-independent format similar to

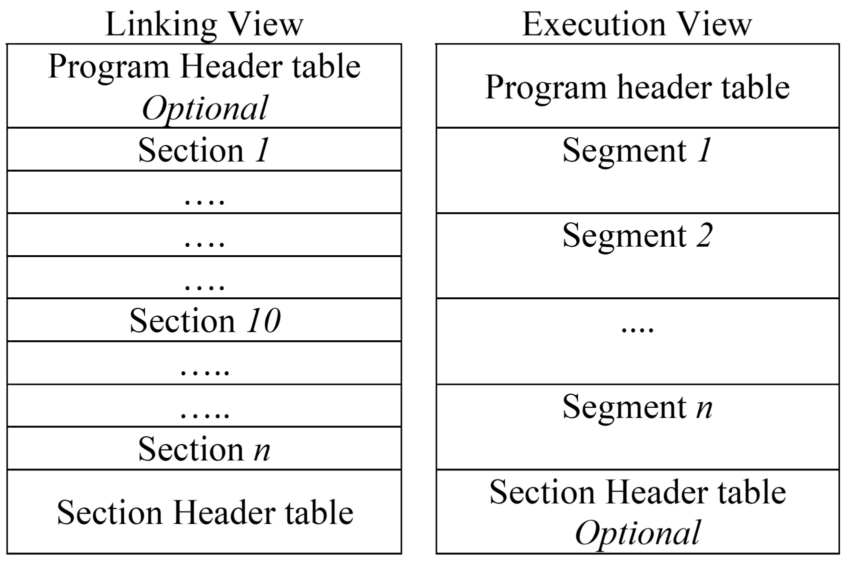

FIGURE A-1. Object File Format.

ordinary object files. Remaining data in the object file uses encoding of the target processor, independent of the machine on which the file is created.

A.3. ELF Header

The ELF Header is always the first section of the file. It is the only section that has a fixed position in the object file. The ELF Header describes the type of the object file (relocatable, executable, shared, core), its target architecture, and the version of ELF it is using. The location of the Program Header table, Section Header table, and String table along with associated number and size of entries for each table are also given. The ELF Header also contains the location of the first executable instruction. Below are a few of these.

```
#define EI_NIDENT       16
```

```
typedef struct
{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
```

- *e_ident*: The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents.
- *e_type* : This member identifies the object file type.
- *e_machine* : This member's value specifies the required architecture for an individual file.
- *e_version* : This member identifies the object file version. The value 1 signifies the original file format; extensions will create new versions with higher numbers.

- *e_entry*: This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.
- *e_phoff* : This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.
- *e_shoff* : This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.
- *e_flag* : This member holds processor-specific flags associated with the file. Flag names take the form EF_machine_flag.

A.4. Section Header

All sections in object files are the Section header table. The section header is similar to program header and is an array of structures. Each entry correlates to a section in the file. The entry provides the name, type, memory image starting address (if loadable), file offset, the section's size in bytes, alignment, and how the information in the section should be interpreted. The name provided in the structure is actually an index into the string table (a section in the object file) where the actual string representation of the name of the section exists. A few sections are described below.

- *.bss* : This section holds uninitialized data that contribute to the program's memory image. The system initializes the data with zeros when the program begins to run. This section occupies no file space, as indicated by the section type, SHT_NOBITS.

- *.comment* : This section holds version control information.
- *.data, .data1* : These sections hold initialized data that contribute to the program's memory image.
- .debug : This section holds information for symbolic debugging. The contents are unspecified.
- *.dynamic* : This section holds dynamic linking information. The section's attributes will include the SHF_ALLOC bit. Whether the SHF_WRITE bit is set is processor specific.
- *.dynstr* : This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries.
- .dynsym : This section holds the dynamic linking symbol table, as "Symbol Table" describes.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. A process image has segments that hold text, data, stack, and so on.

- *Program header* : This section describes object file structures that relate directly to program execution. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.
- *Program loading* : Given an object file, the system must load it into memory for the program to run.

- *Dynamic linking* : After the system loads the program, it must complete the process image by resolving symbolic references among the object files that compose the process.

A.5. Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file segment contains one or more sections, as "Segment Contents" describes below. Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's e_phent size and e_phnum members.

```
typedef struct {
        Elf32_Word  p_type ;
        Elf32_Off   p_offset ;
        Elf32_Addr  p_vaddr;
        Elf32_Addr  p_paddr;
        Elf32_Word  p_filesz ;
        Elf32_Word  p_memsz ;
        Elf32_Word  p_flags ;
        Elf32_Word  p_align ;
    } Elf32_Phdr ;
```

- *p_type* : This member tells what kind of segment this array element describes or how to interpret the array element's information.

- *p_offset* : This member gives the offset from the beginning of the file at which the first byte of the segment resides.

- *p_vaddr* : This member gives the virtual address at which the first byte of the segment resides in memory.

- *p_paddr* : On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because the System ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

- *p_filesz* : This member gives the number of bytes in the file image of the segment; it may be zero.

- *p_memsz* : This member gives the number of bytes in the memory image of the segment; it may be zero.

- *p_flags* : This member gives flags relevant to the segment. Defined flag values appear below.

- *p_align* : As "Program Loading" later in this part describes, loadable process segments must have congruent values for p_vaddr and p_offset, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, p_align should be a positive, integral power of 2, and p_vaddr should equal p_offset, modulo p_align.

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When - and if - the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

A.6. Program Loader

The program loader loads the program from into the simulated memory space with the file offset as the address locations. Here in this example we have a header file of length 0x100 bytes, text segment is of size 0x1d00 bytes, and a data segment of size 0xf100 bytes. The virtual address is the location of the data on the machine memory. In this example we assume that this address starts from 0x80000000. Virtual addresses and file offsets for the SYSTEM architecture segments are congruent modulo 4 KB (0x1000) or larger powers of 2. Because 4 KB is the maximum page size, the files will be suitable for paging regardless of physical page size.

Although the example's file offsets and virtual addresses are congruent modulo 4 KB for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.

File Offset	**File**	**Virtual Address**
0	ELF Header	
	Program Header File Other Information	
0x100	Text Segment	0x8000000
0x1e00	Data Segment	0x8001d00
0xff00	Other Information	0x800fe00

FIGURE A-2. Program Loading.

- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure each logical page in the address space has a single set of permissions.

The region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data. The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus, if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. "Impurities" in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (0x1000) pages. The figure below shows a more detailed version of the text segment and data segment. In a usual case the first and last part of both the segments are padded with zeros'. In this example we assume the text segment starts at a virtual address of 0x08048000

Virtual Address	**Contents**	**Segment**
0x08048000	*Header padding* 0x100 Bytes	Text
0x08048100	Text Segment	
0x08073f00	Data Padding 0x100bytes	

0x08074000	Text Padding 0xf00 bytes	Data
	Data Segment 0x4e00 bytes	
0x08079d00	Unitilized Data 0x1024 zero bytes	
0x0807ad24	Page Padding 0x2dc zero bytes	

FIGURE A-3. Process Image Segments.

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus, the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior.

Though the system chooses virtual addresses for individual processes, it maintains the segments' relative positions. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file.

A.7. How ELF File Looks

- Identifies the file as ELF
- The file contains 32 bit words. The lowest significant byte in a word is in the lowest address
- A relocatable program for an Intel386 cpu. There are no program sections for a relocatable file.
- The address of the first instruction to run is 0. The value is filled in later.
- The section header describing the executable code section. To identify the header add the value of the name field (1b) to the start of section 5 (64) and look at the string (in unix 'text' means program code).
- The section header describing the data for the program.

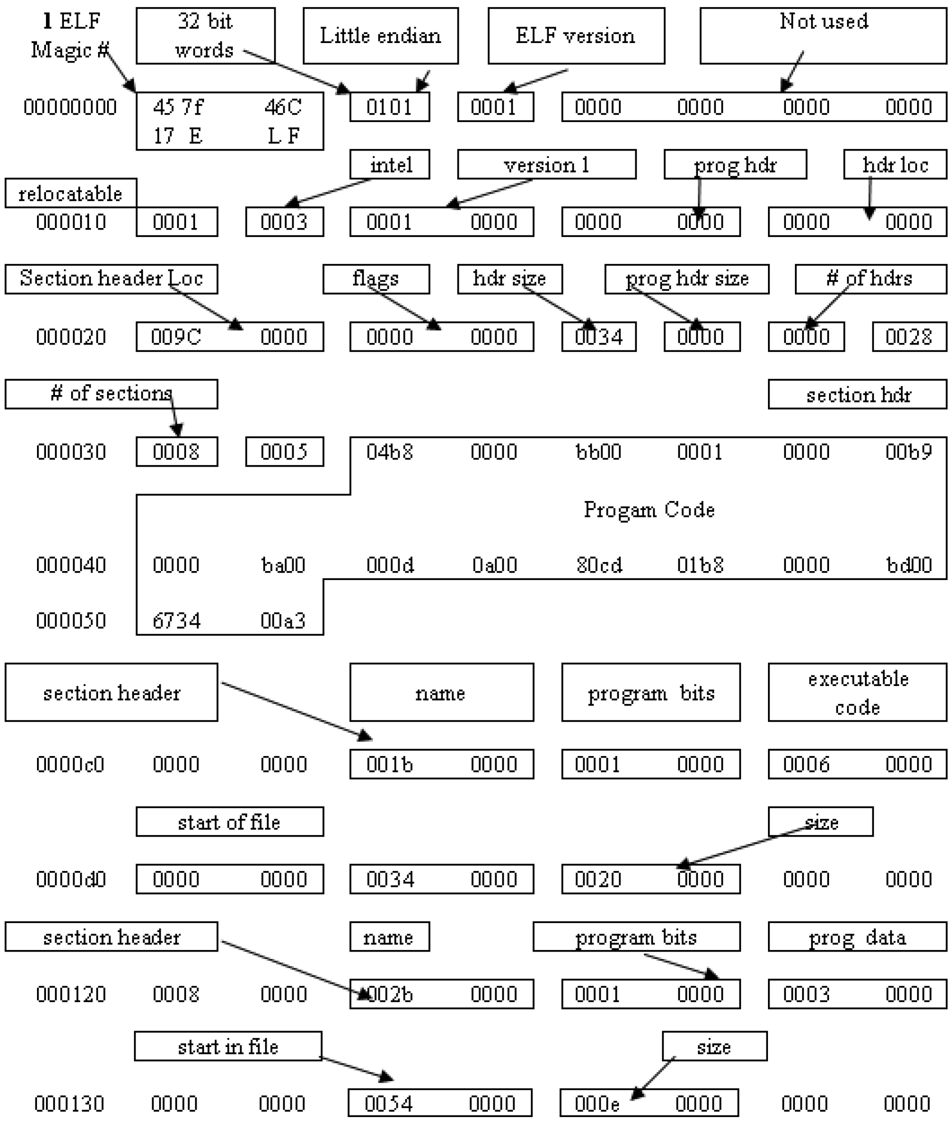

FIGURE A-4. ELF file.