Threading model optimization of the AEMB Microprocessor

By

MOSTAFA MOHAMED IBRAHIM MANSOUR

FINAL PROJECT REPORT

Submitted to the Electrical & Electronics Engineering Programme
In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Engineering (Hons)
(Electrical & Electronics)

Universiti Teknologi Petronas
Bandar Seri Iskandar,
31750 Tronoh,
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Threading model optimization in the AEMB Microprocessor

by

Mostafa Mohamed Ibrahim Mansour

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

____

Mr. Patrick Sebastian
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

August 2014

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

Mostafa Mohamed Ibrahim Mansour

ABSTRACT

AEMB is a 32-bit RISC architecture processor with multi threading. It is a soft core processor designed for FPGA implementation and available as an open source. The processor runs on the instruction set of the Microblaze processor developed by Xilinx. The current threading model in AEMB is a fine grained model that interleaves threads one instruction at a time with separate register sets for each thread. This project aims at understanding the architecture of the AEMB and improving the performance of its threading model.

The chosen optimization is to change the current threading model to a coarse grained one that switches threads on branch instructions. The advantage of this approach is that the pipeline no longer has to stall on every branch instruction executed as the processor will be executing instructions from another thread. Thus, branches cause the processor to stall only when there is back to back branch instructions or when two branch instructions with one gap between them and the first of them has no delay slot. This is quite an improvement over the previous case where the processor stalls for one cycle on any branch instruction encountered.

The disadvantage to the coarse grained threading model is that data hazards that can’t be forwarded can now cause the processor to stall up to three cycles in the worst case scenario compared to only one cycle stall in the old model.

As for Area consumption on FPGA, synthesis showed that the modified core utilizes double the number of LUTs that the original AEMB needs but there was no significant increase in the number of register.

Further quantitative analysis is necessary to determine the total gain in performance by running the suitable benchmarks on both versions of the processor. The results are expected to be in favor of the design if the improved case is more common that the negatively affected cases.

ACKNOWLEDGEMENTS

I would like to extend my sincerest gratitude to my supervisor, Mr. Patrick Sebastian, for his guidance patience and continuous support to my project. In addition I would like to thank Mr. Lo Hai Hiung for his assistance and enthusiasm for the project. Last but not least many thanks are owned to Dr. Shawn Tan for allowing me to use his idea for my FYP and continuously guiding me in technical matters related to AEMB.

Table of Contents

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Computers are providing a very unique addition to every aspect of our lives at very low cost and with technology that is becoming abundant day after day. Thanks to the countless breakthroughs in the architecture of microprocessors, very small processors nowadays can handle tasks that required mainframes decades ago. In addition, computers are present at a very low cost and small size. An example is microcontroller chips like PIC or ATMEL chips. With these chips, whole systems can be built that fit in our palms. Another important factor is that the technology and concepts necessary to design computers are accessible by average individuals. Hence, with the presence of Microcontrollers that target hobbyists and FPGAs that can host chip designs, the boundaries on computer and system design are dissolving day after day.

This explains why computers are embedded in our mobile phones, tablets and other personal devices that are slowly growing into a necessity for individuals. Computers are embedded in our houses, security systems, cars and endless other examples. On the industrial level, the rewarding increase in production resulting from automation brings computers into every industry. With this high penetration level of computers in our everyday lives and industries, various traits and requirements are being used to judge a computer. Size and efficiency are two of those important traits and they are the focus of this project

With the emergence of smart phones and tablets, the trend is now for personal, portable devices to be able to handle sophisticated processes that used to be run by main frames a few decades ago and were considered heavy duty by desktops few years ago. Thus, increasing the computational power of computers embedded in those devices while maintaining their small size is a persisting need. For a while, processor architects had it easy by increasing the performance of processors through making use of the developing semiconductor process. However, nowadays both the clock rate and the size of CMOS technology elements are hitting an upper limit. Hence, architectural optimizations and improvements in efficiency are becoming more important than ever.

At the heart of every computer is a microprocessor. This project focuses on optimizing an existing microprocessor to achieve higher computational power through threading model architecture.

1.1 Background

AEMB is an open source soft core processor developed by Aeste Works (M). Open Source refers to the availability of the source code of a product publicly and free of charge online. Furthermore, AEMB can be used in any commercial product without having to pay royalties to the designing company. Open source products are usually released under different versions of the general public license.

Being a soft core means that AEMB was designed specifically for FPGA implementation. While FPGA and ASIC designs share many of the initial design stages, tools and coding languages, some minor differences still exist when designing for either of them. Those differences stem primarily from the available synthesizable digital elements and the complexity of the process.

Currently, AEMB uses a fine grained threading model. It performs efficiently when running two threads together. Furthermore, AEMB has two distinct sets of registers for each thread. While AEMB was developed based on the Microblaze instruction set, its architecture differs from the Microblaze's with threading being one of the major differences.

This project aims at modifying the existing AEMB core to produce a new optimized version. The design goal for that version is to optimize the threading model for AEMB to perform efficiently when running single thread programs. The key focus is achieving higher computational speed through efficient use of clock cycles.

The project aims at providing a properly tested design by simulation and through FPGA implementation. One of the key goals of the project is to incite deeper understanding of microprocessor architecture are design.

1.2 Problem Statement

- Fine-grained threading model performs poorly when running a single threaded application
- With the variety of applications for microprocessors, different judging criteria emerge for each application which adds to the demand for application specific designs.
- Application-specific computers require a wide range of microprocessor designs to cater for each need.

1.3 Objectives

- To improve the performance of the threading model of the AEMB Microprocessor.
- Quantitatively characterize the performance of the new AEMB core.
- FPGA implementation for the newly designed core.

CHAPTER 2
LITERATURE REVIEW

The architecture of microprocessors has been ever evolving and ever expanding with ideas being researched every day to improve the efficiency and speed of Microprocessors. One of the basic concepts is Instruction Level Parallelism (ILP) which aims at exploiting independent instructions and running them in parallel. With ILP optimizations hitting a roof [1], Thread Level Parallelism (TLP) opens new horizons for improving the architecture. More interestingly, TLP and ILP are both orthogonal i.e. they can both be exploited in the same architecture without having to favor one over the other or balancing a trade-off between both architectures.

Threads are different blocks of code that is executed independently with each one having its own instructions, data, program counter (PC), registers and page table. Threads can be different programs running at the same time or different parts of a single program. Threads can run on different functional units or they can interleave sharing the same functional units. Threads can really have their own PC or registers or the processor can be simply shifting between two sets of registers while actually containing only a single register set. While threads are supposed to be independent pieces of programs, this is rarely the case and dependencies do exist between threads raising a need for communication between threads and sharing data.

Threading Models have many issues to worry about. The first issue is the granularity which is essentially a decision of when to switch between threads in order to achieve maximum efficiency out of available clock cycles. Fine-grained threads switch between threads after every clock cycle. This allows new threads to cover up stalls of other threads [2, 3, 4]. Coarse-grained threads, on the other hand switch threads only when big stalls are present for one thread such as cache misses [5]. The short coming of fine-grained is its poor performance when running single threaded program as it will fail to fill the stalls of this one and only thread. Coarse-grained on the other hand can leave full clock cycles empty if there is no thread ready for execution. That’s why architectures are going towards combining wide ranges of granularities [6, 7].

Dependencies are another issue with threading. When threads contain data dependencies they need to interact to exchange the correct data. Synchronization is one way to solve dependencies [8]. The compiler inserts synchronization points where data dependencies are expected to occur. Unfortunately, during run-time, some dependencies don’t really occur yet clock cycles are still wasted to synchronize. Speculation is the other approach, threads continue to process and only when real data dependencies occur then any incorrect data is squashed and the thread is supplied with the correct input [6, 9, 10].

AEMB is currently using a fine-grained threading model. The model allows the architecture to achieve almost 100% efficient use of clock cycles when running more than one thread but its efficiency drops when running single threaded programs. The purpose of the first modification of the AEMB core is to use coarse-grained threading model to improve the efficiency of the processor when running single threaded programs.

CHAPTER 3
METHODOLOGY

To begin with, a thorough understanding of the AEMB architecture is a must before editing the core. To achieve that, the HDL code for the AEMB is the only source for the task as the datasheet for the AEMB provides minimal explanation. At the same time it is important to understand the basics of microprocessor architecture and relate the reading with the design of the AEMB. The initial stage began with analyzing the HDL code of the AEMB and comparing it with the instruction set of the Microblaze. Following that several drawings were generated representing the circuit for all of AEMB modules based on the HDL code. This approach provided a proper schematic of the AEMB and allowed all its subtle details to be properly observed.

Following that, it was necessary to monitor the pipeline and various modules of the processor in action. The purpose of this is to be able to understand how all the modules of AEMB work together and how the control signals are handled. The most efficient way was to compile some program for AEMB and simulate it through the processor and monitor the waveform.

Last but not least, it is crucial to be aware of the inferred circuits that represent this processor on the FPGA. This is easily obtained by synthesizing the processor using the suitable synthesis tools for the target FPGA. This way when doing any changes, the designer will be aware of the key circuits that were targeted in the FPGA and will avoid altering the code that infers them or if necessary create better code to infer more suitable circuitry.

The next step is to research on the required modifications for this project. Through proper literature review, research papers and textbooks were a great help in understanding the concepts of threading and its different models. Research papers provided a very rich material that contains not only theoretical information but also the findings of distinct researchers regarding threading.

After being familiar with the AMEB architecture and having a strong theoretical base regarding threading, the design stage commenced. Design is about choosing the most promising modifications and adjusting the structure of the processor accordingly.

Once design and modification are done the first thing to check is the functionality of the processor. Functionality means that the processor still performs all instructions properly and yields the correct results without any regard to performance. This can be verified by running a bench mark that tests all the functionality of AEMB.

Finally the most important part is testing the new design through simulation by running various benchmarks that can expose the effect of the modifications on the performance of the processor. The data acquired are then compared with the original AEMB design. If the performance is worse or the improvement is not satisfying enough then new concepts need to be explored until the required results are obtained. All results and tests need to be professionally documented as they act as a proof to the performance of the processor.

After the processor performs well in the simulation stage it is then tested on FPGA for a proof of concept. This stage proves that the synthesis tool infers the correct circuitry for the processor and that all HDL code is synthesizable. It might be necessary to provide a test bench and a sample program that users can use to observe the effect of the new changes.

3.1 Project Milestones

3.1.1 First Semester

1. Understand basics of microprocessor architecture.
2. Understand & characterize the AEMB core.
3. Research on threading models.
4. Understanding the threading model of AEMB.
5. Understanding the program flow of AEMB.
6. Understanding the forwarding unit of AEMB.

3.1.2 Second Semester

7. Identify data dependencies that needs resolution due to the new model
8. Preparing a list of necessary changes for the new version.
9. Coding the new design
10. Testing the design by simulation
11. Comparing the performance of the new version against the old version
12. Testing on FPGA and adjusting design
13. Finalizing results, writing test benches and programs for users.

3.2 Gantt chart

3.2.1 First Semester

	January				February				March				April		
Milestones	Wk	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	1	■	■	■											
	2			■	■	■	■	■	■	■					
	3										■				
	4											■	■		
	5													■	
	6														■

3.2.2 Second Semester

	May				June				July				August			
Milestones	WK	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	7		■	■												
	8				■	■	■									
	9							■	■	■	■					
	10											■	■	■		
	11														■	
	12														■	
	13															■

CHAPTER 4
RESULTS AND DISCUSSION

While the primary objective of this project is to optimize the threading of AEMB, a great deal of time was spent in understanding the initial architecture of AEMB. Hence the results will be divided into three sections. The first section explains the architecture of the AEMB and the operation of its original threading model. The second section explains the new threading model and the changes that took place to the various parts of the core to adapt coarse grained threading. Last but not least the third part provides a qualitative analysis of the performance of the new core. It's worth stressing that a quantitative analysis still needs to be conducted to properly characterize the performance of the new core.

4.1 AEMB Architecture

The ISA of AEMB is pretty much the same ISA for the Microblaze EDK 6.3. Instructions are all 32 bits and they come in two types. Type A have two source registers, A & B, and a destination register, D. Type B has one source register, one destination register and a 16 bit immediate value. The Immediate value can be sign extended to 32 bits or the top 16 bits can be supplied from the immediate instruction to create a 32 bit value. Internally, AEMB groups the instructions into the following types; Arithmetic, Multiply, Barrel Shift, Return, Conditional and Unconditional Branches, Immediate, Load and Store and Get/Put instructions. While Microblaze uses the GET instruction to access its Fast Simplex Link, FSL, AEMB uses it to access its accelerator bus.

4.1.1 Memory Buses

AEMB uses Wishbone bus for all of its interfaces with the outside world. It has three buses each dedicated for interfacing with one of the following; instruction memory, data memory and accelerators. I/O devices can be memory mapped through the data bus.

AEMB interfaces with the instruction memory through the module IWBIF and uses an instruction cache; ICHE. In edk63 version of AEMB, this cache can store 512 words (32 bits per word). It keeps track of which instructions are in the cache through a look-up table. Each entry in this look-up table can keep track of the status of one cache line (16 words). The highest most 21 bits of an instruction address are used as a tag value saved in the look-up table. The next 5 bits are used to select a word in the look-up table. The status of a word is expressed by a single bit. First when an instruction is needed the processor looks it up in the cache. It loads it if there was an instruction hit. On misses, IWBIF fetches the instruction from memory.

AEMB communicates with data memory and accelerator bus using the DWBIF and XSLIF respectively. In these interfaces, the wishbone signals are defined and the data coming in from wishbone is latched on. AEMB accesses the data memory only on Load or Store instructions. Unlike with the instruction interface, the wishbone cycle and strobe signals in the data interface are not connected to each other. There are 12 instructions responsible for memory access. Half of them are load instructions that read data from memory and write it back to the destination register. The data read can be a word, half a word or a byte. The memory address can be calculated by adding two register together or a register and an immediate value. The store instructions write data from a register to the memory and they come with the same options as the load instructions.

The Accelerator bus is accessed on Get and Put instructions. AEMB only uses the blocking instructions which freeze the pipeline until a transaction is complete with the accelerator and doesn’t put a limit on the number of cycles that this takes. Those instructions specify the address of the accelerator to be read as an immediate value and they have a bit to specify whether the register targeted at the device is a control/status or data register.

4.1.2 Pipeline

AEMB creates its internal clock, reset and flag interrupts and the signal that switches between active threads, GPHA, in the PIPE module.

There exists 4 special purpose registers. Those are Program Counter (PC), Machine Status Register (MSR) and two more registers that contain the exception status and address (ESR & EAR). While all 4 registers can be read, only the MSR can be edited by the user. There exists three instructions to write to the MSR, one copies the content of the register to it and the other two set or clear bits from the MSR lower half depending on the values of an immediate value. One instruction exists to read any of the 4 special purpose registers and store their content into the destination register.

Instruction decoding takes place in the CTRL module. The OPCODE is decoded to identify the type of instruction and this information is forwarded to the rest of the processor. Addresses of operand and destination registers are obtained from the instruction word and saved into registers to free up the pipeline for the coming instruction. Furthermore, control signals for forwarding the operands or the destination are created in this module and if any forwarding is needed for those data to be used in the next cycle then it takes place in this module.

AEMB uses a register file that contains 64 general purpose registers. This is double the number of registers used in the Microblaze. This is because each 32 registers are used for one thread. The GPHA signal is used as the MSB of the address of the registers to determine which thread is currently active.

4.1.3 Arithmetic Logical Unit

The execution unit contains an integer unit, a multiplier and a barrel shifter. The integer unit handles addition, subtraction, logical operations and one bit shifting either logically or arithmetically.

AEMB arithmetic instructions come in a variety. Addition and subtraction can be carried by 16 instruction each of which giving the user a special choice of options. Subtraction is done by adding the 2′s complement of operand A to the other operand. The operands can be either two registers or a register and an immediate value provided by the instruction. Furthermore, the instructions have the option to add the value of the carry bit into the operation. An interesting ability is to keep the current carry flag unaltered. If the K bit is set, the carry bit is not altered by the operation otherwise the carry bit receives the carry out value. An interesting compare instruction exists that subtracts operand A from B and then sets the MSB of the destination register if operand A is bigger than B or clears it otherwise. There exists two variations of this compare instruction, one can consider A & B to be signed numbers and the other considers them unsigned numbers.

AEMB can perform 4 logical instructions; OR, AND, XOR and AND Complement. 8 instructions exist for those logical operations; 4 of which take in two registers as operands and the other four use one register and an immediate value.

As for shifting, 3 variations exist for a right shift. Basically operand A is shifted to the right and place into the destination register. The carry bit of the MSR takes the value of the bit shifted out of A. The variations of shifting define the value of the MSB in the destination register. A logical shift clears this bit while an arithmetic shift extends the MSB of A into the MSB of the destination. Finally, a carry shift will place the carry bit into the MSB.

AEMB has two Sign Extend instructions. It can extend a Byte or half a word of one register and place it in another. Barrel shifting has 6 different instructions. The instructions can shift to the right or to the left by a quantity specified in operand B or immediately in the instruction. Shifting can be logical or arithmetic similar to the shift instruction. However arithmetic shifting is only an option when shifting to the right. As for multiplication, two instructions allow multiplication of a register with a register or an immediate value. The value stored in the destination register is the lower word of the multiplication result. Multiplication and Barrel shifting are the only arithmetic instructions that take more than 2 cycles to execute.

The execution unit considers any floating point instruction as an exception. Moreover, the integer unit contains the Machine Status Register (MSR). A major difference between the MSR of AEMB and the MicroBlaze is that bit 29 of AEMB MSR contains the current threading phase.

4.1.4 Program Flow Control

As for Program Flow Control, AEMB supports exceptions, one external hardware interrupt and the regular branching scheme. Exceptions in AEMB occur when internal errors are detected. Those include improperly aligned data. AEMB considers any floating point instruction as an exception. AEMB supports external interrupt through one interrupt channel.

Branching in AEMB mimics the branching scheme in Microblaze. Conditional branches are resolved at a special unit, the Branch Condition Check (BRCC) unit. If branches are not taken, the branch instruction has a latency of only one cycle. Moreover, AEMB supports a delay slot after branches. Branches with delay slots have a latency of 2 cycles while remaining branches have 3 cycle latency.

Program flow instructions are the biggest family of instructions in the AEMB. 24 instructions exist for conditional branching, 12 for unconditional branching and 4 instructions for returning. Conditional branching compares the contents of register A against zero. All comparison variations exist. There is equal to zero, not equal, greater than or less than, greater than and equal to zero or less than or equal. The address is calculated by adding either the content of register or an immediate value to the current PC value. Moreover, all conditional instructions have an option to include a delay slot after them.

Unconditional branching comes in a combination of options. First, there is the option to branch and link where the content of the current PC is first copied to a register before branching. Second the option exists as to whether the branch target should be an increment of PC or an absolute value. Finally like conditional branches, the presence of a delay slot is also an option. One special branch instruction is a break instruction which manipulate the break in progress bit in the MSR.

Return instructions all calculate the value of the target address by adding an immediate to the contents of a register. However, the difference between the 4 instructions is in the way they manipulate the bits of the MSR. There exists return from exception, interrupt, break or subroutine.

4.1.5 Data and Control Hazards

AEMB has two kinds of hazards, branching and forwarding. When a hazard occurs, the control unit inserts a NOP into the pipeline. Branch hazards occur when a branch with no delay slot is taken. Forwarding hazards occur on data dependencies between instructions.

Data dependencies are handled through forwarding and bubble insertion when forwarding is not possible.

There are instructions which don't write back to the register and hence their following instructions can never have a data dependence on them. Those instructions are conditional branches, unconditional branches except when a link is required, return instructions, store instructions, PUT instructions, move to special purpose register and the immediate instruction.

Adding to that, there are three instructions that don't access the registry file and hence can't depend on their previous instructions. Those are the GET instructions, move from special purpose register and finally the immediate instruction.

There are instructions that have a latency of 2 cycles and hence can't accommodate forwarding. Those are multiplication, barrel shifting, LOAD and GET instructions and move from special purpose register. When these instructions are followed by an instruction that depends on them a bubble is inserted in the pipeline by not allowing a write back to the register file and the depending instruction is loaded once more into the pipeline to give the preceding instruction a chance to retire.

Thus forwarding in AEMB is done only with arithmetic, logical and shift instructions i.e. instructions that go through the ALU and have a latency of only one cycle which is enough to forward their data to the following instruction before they reach the execution stage.

4.1.6 Original Threading Model

In AEMB, each thread has a separate set of register file and hence no dependencies are expected to occur between threads. More importantly, AEMB has two modes of accessing the threads. The first is to issue the same instruction twice in consecutive clock cycles, that is, to use the same instruction to write to both threads. This is used in the beginning of the program before the flow is split between the two threads to setup data memory and the register files for both threads. The second mode takes place after the program is split into two threads and instructions are interleaved in a fine manner.

The splitting of threads takes place in the program init function and it depends on the 4th bit in the MSR; the MUTEX bit. When AEMB attempts to set the MUTEX bit it is set only for one thread but not the other as the instruction path for MSRSET has an extra register in the pipeline which delays its write back by one cycle. Thus the write back takes place only to one thread. After this instruction is executed a branch instruction is used to check the register where it was written and hence the branch condition turns true for only one thread and the two threads split from each other and are no longer running the same instructions.

The way the threads are utilized as explained before makes it not necessary for the original AEMB to resolve data dependencies between back to back instructions as no back to back instructions will exist from the same thread. Data dependencies that can't be resolved will only stall the pipeline for one clock cycle. However, whenever a branch instruction occurs the pipeline needs to be stalled once until the branch target is calculated.

Thus, it can be clearly observed that AEMB uses it's threading model to cover up for back to back data dependencies and reduce necessary stalls of the pipeline. The processor is kept busy by fetching an instruction from another thread instead of stalling till the current instruction resolves. Those covered up dependencies save on chip area as not much forwarding is needed.

4.2 Modified AEMB Core

The key modification is replacing the fine grained model by a coarse grained model. The rest of the modifications are adjustments to different parts of the core to properly function with the new threading model.

The Coarse grained model of AEMB changes thread whenever a branch instruction, be it conditional, unconditional or a return is inserted into the pipeline. The main motivation behind this choice is to cover up the stalls that AEMB need to insert whenever a branch instruction is taken.

In the initial stages of the program both threads begin at address zero. The first address starts execution and once it hits a branch the core switches to the other thread which starts at address zero as well. This way the two threads will be executing the same initial instructions necessary to initialize the core until threading split takes place. The splitting mechanism is exactly the same for the original core. It causes a branch instruction to evaluate true for one thread but not the other hence setting the threads on different paths.

The address unit was modified to support two necessary abilities, first being able to fetch the next instruction straight away after the current one with no gap between and branches need to be detected as early as the instruction is fetched to enable the next cycle to switch threads right away.

The effect of the new threading model is very crucial on branch instructions. One way to think of it is that the new core branches to another thread whenever a branch instruction takes place. It remains in the other thread till a branch instruction is encountered and then it returns to the first thread precisely at the branch target where it was supposed to go. This way the pipeline is given a chance to resolve the branch target and hence no stalls are required except in two cases; if two branch instructions exist back to back or if two branch instructions have one instruction gap between them and the first branch instruction has no delay slot. The first case should be observed rarely in programs.

As for data hazards, the new model will cause instructions from the same thread to enter the pipeline in proper order without any gap between them like the previous fine grained model. Thus several changes took place to resolve those data hazards. First forwarding from the ALU unit back to the decode stage need to be implemented. A new forwarding path was established and the circuit to detect back to back hazards was added. Moreover, for instructions that can't be forwarded dependencies with two instruction gap need to be resolved. In this case stalling is utilized to resemble the way the original AEMB solves those dependencies.

4.3 Performance

The new changes improved some part of the processor, handling control hazards and may have a negative effect on other parts, data hazards particularly those that can't be resolved by forwarding. The overall performance of the modified core will depend on which of the affected cases are more common to occur in test benches and hence in programs. This is a clear manifestation of the design principle that states that optimization reflects only on the overall performance is affected by how often the optimized cases occur. To properly measure that, the two cores need to run several suitable benchmarks and based on the result a conclusion can be drawn on the efficiency of this modification. It is unfortunate that such characterization is yet to be executed and hence a quantitative analysis can't be presented in this dissertation.

Table 1 reflects the Area of both AEMB cores. Synthesis was done using Xilinx ISE 14.4 for a Spartan 6 FPGA. It is clear that the modifications required a very small number of registers on top of the original AEMB core. However, a significant increase in the number of used LUTs. The number of LUTs used by the modified core is almost double the number of LUTs used by the original core. This is due to the hazard detection circuit that needs to be added for two extra stages of the pipeline. It is worth mentioning that the current modifications are simply a first version which can be optimized to decrease the number of inferred LUTs.

Point of Comparison	**Original AEMB Core**	**Modified AEMB Core**
Slice Registers	953	959
Slice LUTs	1221	2376

Table 1, Consumed Area after Synthesis

CHAPTER 5
CONCLUSION

AEMB, an open source soft microprocessor core, needs modifications to improve its threading model. Currently, it is using a fine-grained threading model which switches between threads every clock cycle. This model is not efficient for running single threaded programs. Coarse-grained model can be helpful in optimizing the performance of AEMB for running single threaded programs. The process to implement those changes to produce a new AEMB starts with designing, coding and finally benchmarking. So far, AEMB has been thoroughly studied and the necessary planning for the new core has been conducted and all parts that need to be changed have been identified. Currently the design is in the coding stage and will be entering the final verification and testing stage soon.

REFERENCES

[1] D. W. Wall, “Limits of instruction-level parallelism,” Digital Equipment Corporation, Tech. Report. 93/6, Nov. 1993.

[2] M. Amamiya, H. Tomiyasu, S. Kusakabe, “Datarol: A Parallel Machine Architecture for Fine-Grain Multithreading,” in Proc. Third Working Conf. on Massively Parallel Programming Models, 1997, pp. 151 – 162.

[3] M. Loikkanen, N. Bagherzadeh, “A Fine-Grain Multithreading Superscalar Architecture,” in Proc. Conf. on Parallel Architectures and Compilation Techniques, 1996, pp. 163 - 168

[4] J. Kim, S. Ha, C. S. Jhon, “Evaluation of Various Node Configurations for Fine-grain Multithreading on Stock Processors,” in HPC Asia High Performance Computing on the Information Superhighway, 1997, pp. 349 – 354

[5] R. Gerndt, R. Ernst, “An Event-Driven Multi-Threading Architecture for Embedded Systems” in Proc. 5th international Workshop of Hardware/Software Codesign, 1997, pp. 29-33.

[6] T. Ohsawa, M. Takagi, S. Kawahara, S. MatsushitaT, “Pinot: speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities,” in Proc. 38th Annu. IEEE/ACM Int. Symp. Microarchitecture, 2005, pp. 81-92.

[7] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, “Exploiting Fine–Grain Thread Level Parallelism on the MIT Multi-ALU Processor,” in Proc. 25th Annu. Int. Symp. Computer Architecture, 1998, pp. 306-317.

[8] D. M. Tullsen, J. L. Lo, S. J. Eggers, H. M. Levy, “Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor” in Proc. 5th International Symp. on High-Performance Computer Architecture, 1999, pp. 54-58.

[9] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukolun, “The Stanford Hydra CMP,” Micro, IEEE, vol. 20, no. 2, pp. 71-84, Mar/Apr 2000.

[10] J. -Y. Tsai and P. -C. Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation” in Proc. 1996 Conf. Parallel Architectures and Compilation Techniques, pp. 35-46.