SP

COSC
KEP
02/03

COSC 4235

Thesis in Computer Science

Investigating The Advantages Of A 64 Bit Microprocessor Architecture

By: Trevor Keppel-Jones

Advisor: Klaus Peltsch

Abstract

The main purpose of this project is to explore the function and capability of 64 bit processors. Sixty-four bit processors have been on the market for roughly 5 years now, but it is not clear what their advantage is over the 32 bit processors. The 64 bit processor would be expected to provide benefits in the area of computation speed and mathematical accuracy. My project explores the advantages of the 64 bit processor over the 32 bit processor specifically in the area of floating point computation.

Table of Contents

Chapter 1 Introduction

1.1 Background 5

1.2 Thesis Plan 5

1.3 Key Terms 7

1.4 Outline of Report 8

Chapter 2 Literature Review

2.1 64 bit vs. 32 bit 9

2.2 RISC vs. CISC 10

2.3 Benchmarking 11

2.4 Floating Point Arithmetic 13

2.4.1 IEEE Standards 14

Chapter 3 Methods

3.1 Hardware Profiles 15

3.2 FPU test 17

3.3 Crank Nicolson Algorithm 18

3.4 Eliminating Errors 19

3.4.1 Loop and Declaration Overhead 21

3.4.2 Compiler Optimization 22

3.4.2.1 Unrolling The Loop 22

3.4.3 Operating Systems 24

Chapter 4 Results

4.1 FPU test 24

4.1.1 486 DX4 100 MHz 24

4.1.1.1 Eliminating Overhead 24

4.1.1.2 FPU test results 25

4.1.1.2 MFLOPS graphs 27

4.1.2 DEC Alpha AXP 21064 166 MHz 30

4.1.2.1 Eliminating Overhead 30

4.1.2.2 FPU test results 30

4.1.2.2 MFLOPS graphs 32

4.2 Crank-Nicolson Algorithm 34

4.2.1 Sample Output from Algorithm 34

4.2.2 Speed Results and Discussion 35

Chapter 5 Discussion 37

Chapter 6 Conclusions 38

Chapter 7 Code: 7.1 test.cpp 39

7.2 FPUtest.cpp 40

7.3 crank.cpp 42

7.4 test.c 45

7.5 FPUtest.c 46

7.6 crank.c 47

Bibliography 49

1. Introduction

1.1 Background Information

In mid 1994, a group of local information technology (IT) professionals partnered with Sault Ste. Marie's Economic Development Corporation (EDC) to form a "Telecommunications Committee". Their mandate was to develop and support strategies that promote information technology-enabling business opportunities, including the establishment of community and regional information networks. Then, in May 1995, several members of this committee initiated the formation of SSMCCN (Sault Ste. Marie Community Computing Network), with the specific purpose of implementing a Sault Ste. Marie "freenet". SSMCCN then procured a $20,000 DEC Alpha AXP 21064 as a donation from DEC and ONYX, sixteen modems from Bell Canada, Algoma Business Computers, and Soonet, a $7000 terminal server from Gandalf, and many other donations of time and money from other local area businesses.

After discussing possible topics for a thesis with my advisor, we decided that the DEC Alpha AXP 21064 (which has a 64 bit processor) would be a suitable test machine for investigating what a 64 bit processor's advantages are over a 32 bit processor.

1.2 Thesis Plan

The purpose of this project is to investigate the advantages of a 64 bit processor over a 32 bit processor. This project will specifically address the differences between the FPU's (Floating Point Units) of a 64 bit processor and that of a 32 bit processor. The two areas that will be focused on are speed and accuracy. To do this, two algorithms will

be executed on both a 64 bit and a 32 bit processor and the results will be compared. To compare speed, a small program that executes different floating point instructions millions of times will be timed. To compare accuracy, the plan was to code the Crank-Nicolson algorithm for solving partial differential equations and perhaps use a larger precision on the Alpha than the PC, but when it was discovered that the DEC C compiler on the Alpha did not support IEEE double extended precision and the Borland C++ compiler on the PC did, it was decided that this algorithm could also be used for a good measure of speed on a real world application.

A few issues that will not be addressed in depth are the differences in the compilers (on the 32 bit PC a Borland C++ compiler was used and on the Alpha, a DEC C compiler was used), and the finer details of the architectures of the two machines.

1.3 Key Terms

The key terms that will be used in this paper are defined as follows:

DEC -	Digital Equipment Corporation
FPU -	Floating Point Unit
MFLOPS -	Millions of Floating Point Operations per Second
MIPS -	Millions of Instructions per Second
RISC -	Reduced Instruction Set Computer
CISC -	Complex Instruction Set Computer
SPEC -	System Performance Evaluation Corporation
TPC -	Transaction Processing Performance Council
IEEE -	Institute of Electrical and Electronics Engineers
SSMCCN -	Sault Ste. Marie Community Computing Network

1.4 Outline of Report

This report is organized as follows:

a) a short literature review of what has been said on the topic of 64 bit processing vs. 32 bit processing, CISC vs. RISC chips, benchmarking and floating point standards,

b) an architectural background of the test machines, the DEC Alpha AXP 21064 and the 486DX4 PC (this includes published benchmarks, and interesting features of the processors),

c) a description of the research methods that were used, and the equipment needed (including software and hardware) to carry out the project (here is an analysis of the code that was written and tested on the different machines and an analysis of the effectiveness of the software),

d) the results of the testing that was done - this contains all data collected and all other relevant information that was used when determining what the results of the testing were,

e) the discussion of the results - here will be an analysis of the results and a discussion on what they mean and what the implications are of these findings,

f) a discussion of the factors that limited the accuracy of the results and what was done to try to eliminate these factors,

g) a final statement of what was learned from the completion of the project.

2. Literature Review

2.1 32 bit versus 64 bit

The CPU's of today's desktop machines are usually either "32 bit" or "64 bit" processors. This means that the machine's registers, addresses, integer units, floating point units (usually have significantly more bits than the architecture of the rest of the CPU), and buses are all 32 or 64 bits wide. DEC's Alpha uses a 64 bit architecture, whereas most desktop PC's are still based on 32 bit architectures (there are still some 16 bit machines around). The debate arises as to whether there is a need yet to move to a 64 bit architecture. A few years ago, DEC and SGI were pushing toward the 64 bit architecture because they felt that users were being held back by the 4 gigabyte limit on real memory address space imposed by a 32-bit architecture (although there are not too many users around that need to address more than 4 gigabytes of memory). Another issue even today is whether there is software to support a 64 bit architecture. It can be argued that there is not much point in having a 64 bit processor if your C compiler (or other software) is designed for a 32 bit architecture. Although it is fairly likely that it would be faster because two 32 bit values could be moved at a time, instead of one. IEEE standards have been defined for single extended and double extended precisions (see 2.4.1) but these two standards are not yet supported in all computer architectures or computer software.

The question of software alone makes it impractical for today's average consumer to purchase a machine based on a 64 bit architecture. There have been rumors of a 64 bit version of UNIX, but other than this, it is unknown whether there is an operating system

in existence today that is based on a 64 bit architecture. And although it is very possible to run 32 bit applications on a 64 bit architecture, it appears to be a waste of the system's resources and the cost of a 64 bit machine does not make it viable for the average consumer.

2.2 RISC versus CISC

"The first modern RISC machine was the 801 minicomputer built by IBM, starting in 1975" (Tanenbaum, 435). RISC stands for Reduced Instruction Set Computer, and a RISC architecture is characterized by a small set of fixed-length instructions that each take only one clock cycle to execute, simple memory addressing modes, and a strict decoupling of load/store memory access instructions from register-to-register arithmetic instructions (Sites, 37). What this means is that only Load and Store instructions can access memory. With a good compiler making sure that the instruction in the pipeline after the Load does not need the item being loaded from memory, the pipeline continues executing one instruction per cycle and the machine continues at full speed. If the instruction after the Load does use the item being loaded, and the compiler cannot find another instruction to insert in the pipeline after the Load, then the compiler may insert a NO-OP and waste one cycle (Tanenbaum, 440). A RISC architecture is usually highly pipelined, and the complexity is found in the compiler.

CISC stands for Complex Instruction Set Computer, and a CISC architecture is characterized by many variable-length instructions that take multiple clock cycles to execute, a wide variety of memory addressing modes, and instructions that combine one

or more memory accesses with arithmetic. CISC designs express computation as a few complex steps (Sites, 37). The instructions in a CISC architecture are interpreted by a microprogram and the complexity is in the microprogram. The CISC architecture uses less pipelining than the RISC, if at all.

2.3 Benchmarking

Benchmarking is the technique of measuring a device's performance against its counterparts using some universal measuring tool for that type of device. In the world of computers, the measuring tools were for a long time MIPS and MFLOPS (Millions of Instructions Per Second and Millions of Floating Point Operations Per Second). In 1975, Curnow and Wichman invented the "whetstone", a benchmark that measured floating point operations as whetstones per second. Then, in 1984, the "dhrystone" appeared as a benchmark for integer operations. These benchmarks were good (they are still used today), but with the advent of the RISC architecture around 1975 (did not really show up on the scene until the early 1980's), measuring instructions became more difficult. The instruction was not a clearly defined unit anymore because the RISC architecture executed one instruction per cycle, whereas the more complex CISC instructions were broken down into microinstructions. After a while, MIPS came to mean "meaningless indicator of processor speed" (Morse, 80). Although this does not discourage computer corporations from publishing their new processor's MIPS number. This meant that it was time to come up with a more standardized set of benchmarks that would not depend on the architecture of the hardware. This need spawned two benchmarking companies:

SPEC (System Performance Evaluation Corp.) and TPC (Transaction Processing Performance Council).

"The SPEC suite of benchmarks is a set of CPU intensive programs that measures a machine's CPU speed against a Digital Equipment Corp. VAX 11/780. The actual measurement is the number of work tasks accomplished in a fixed period of time, which means higher numbers are better" (Morse, 80). Back in 1989, the SPEC benchmark was called SPECmark89 and it was the mean of the output of four integer and six floating point programs. This value was then compared as a ratio to that of the VAX 11/780 to reveal a result. There were problems with this method because the result represented a combination of both integer and floating point calculations, and with compiler optimization tricks, it was hard to tell if the result of SPECmark89 was actually a good measure of the CPU's performance. So, in 1992, the SPEC benchmarks were split up into two main categories, one to focus on integer arithmetic and one to focus on floating point arithmetic. These new benchmarks were called SPECint92 and SPECfp92 and these new benchmarks used a larger number and a wider variety of algorithms to test CPU performance. The six integer programs included applications such as a spreadsheet and a data compression program, and the fourteen floating point programs included algorithms to solve problems in astrophysics, quantum chemistry, hydraulics, plasma physics, optics, neural nets, medical research and matrix analysis (Morse, 80). The calculation for the final numerical result was similar to that of SPECmark89. Recently, SPEC came out with its newest set of benchmarks, SPECint95 and SPECfp95. SPECint and SPECfp have been the most predominant benchmarks over the last four years on the

market for evaluating performance of processors in the areas of integer and floating point computational ability.

Transaction Process Performance Council, the other major benchmarking corporation, has focused on other aspects of the performance of a computer. Their benchmarks, TPC-A, TPC-B, and TPC-C test performance of processor and I/O subsystems, networks, and reading from and writing to databases (Morse, 82). The A and B benchmarks "entail significant disk activity but only moderate demands on the process; they both require that the integrity of all transactions be maintained" (Morse, 82). The C benchmark is similar to the A benchmark "but it is much more complex in both the database design and the transaction scripting" (Morse, 82). TPC's "point of view is definitely that of business and commercial users, since the model database mimics the operations of a commercial bank's branches or a wholesale supplier" (Morse, 82).

"In the end, benchmarks are both measurement tools and marketing tools" (Morse, 84).

2.4 Floating Point Arithmetic

Over the years, different methods of representing real numbers on computer systems have been suggested (floating slash, signed logarithm, etc.), but the floating point convention is now the one used by most computer systems.

Floating point numbers are defined by three parameters, a base, a precision and a sign. The base could actually be any number greater than or equal to two, but the bases that are most often used are 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal).

The precision represents the number of digits that the number will be displayed with, and the sign represents whether the value is positive or negative. Therefore, if we wanted to represent 0.1 as a floating point number with base 10 and precision 3, it would appear as 1.00 * 10^(-1) (Goldberg, 6). But computer systems work in binary at the hardware level, so it becomes necessary to define standards by which all systems will work. Different standards have been defined for different machines over the years (i.e. VAX, IEEE). Intel used the IEEE standards in it's 8088 processor (which was used by IBM in the original PC) and the IEEE standard has now been entrenched in most of today's desktop computers.

2.4.1 IEEE Standards

In the late 1970's, IEEE (Institute of Electronics and Electrical Engineers) saw a need for standardizing floating point computation and so they came up with a set of standards for representing floating point numbers, methods for rounding them, the use of a guard digit to reduce error in calculation etc. The IEEE standards for single precision, single extended precision, double precision and double extended precision are as follows:

Parameter	Single	Single Extended	Double	Double Extended
Precision	24	>= 32	53	>= 64
Exponent Max.	+127	>= +1023	+1023	> +16383
Exponent Min.	-126	<= 1022	-1022	<= 16382
Exponent width in bits	8	>= 11	11	>= 15
Format width in bits	32	>= 43	64	>=79

(Goldberg, 18).

For example, the breakdown for an IEEE double precision floating point number is as follows: 64 bits total are made up of 1 sign bit, 11 exponent bits, and a 53 bit mantissa. The usefulness of having a standard on which all machines can be built is this: "Once an algorithm is proven to be correct for IEEE arithmetic, it will work correctly on any machine supporting the IEEE standard" (Goldberg, 19).

3. Methods

3.1 Hardware Profiles

The two main machines that were used for testing were a DEC Alpha AXP 21064, and a 486DX4 PC.

The Alpha has a RISC architecture (see 2.2), that includes 168 instructions in its instruction set. It is a dual issue CMOS processor, meaning that it issues 2 instructions per clock cycle to any two of the four functional units:

1. the integer unit,
2. the floating point unit,
3. the load/store unit, and
4. the branch unit.

It's obvious that 2 instructions cannot be issued to the same unit in the same clock cycle, but there are some more subtle rules involved. These rules are as follows:

1. any load/store in parallel with any operate (integer or floating point) is possible,
2. an integer operate in parallel with a floating operate is possible,

3. a floating operate and a floating branch is possible,
4. an integer operate and an integer branch is possible,
5. an integer store and floating operate are not allowed, and
6. a floating store and integer operate are not allowed.

The 21064 is a 64 bit processor (meaning that registers, internal buses, and execution units are all 64 bits wide) that runs at 166 MHz. It has a 10 stage fully pipelined floating point unit and a 7 stage fully pipelined integer unit. Pipelining is the idea in which an instruction is fetched in the first stage of the pipeline (during one clock cycle), then in the next clock cycle, that instruction moves into the second stage and is decoded while the next instruction in the program is being fetched. This continues until the last stage of the pipeline in which each instruction is executed. Under ideal circumstances, the pipeline will make it possible to execute one instruction per cycle. In practice this does not work due to branch delays, etc. Under ideal conditions, the 21064's pipelined FPU can produce a 64 bit result every clock cycle for every operation except divide, which is handled by a non-pipelined, single bit per cycle dedicated divide unit. The operating system used on the Alpha was UNIX and the programming language used was C, with a DEC C compiler. Published benchmarks on the 166 MHz 21064 are:

SPECint92/SPECfp92:	90/140
MIPS:	400 (peak)
MFLOPS:	200 (peak).

(Source for all DEC Alpha AXP 21064 architectural information was Digital Technical Journal Vol. 4, No. 4, 1992)

The PC that was used in testing contained an AMD 486DX4 100 MHz processor. The 486DX4 is only a single issue 32 bit CISC processor (has 32 bit registers, data bus, address modes, etc.) with a floating point unit. An interesting feature of this processor is the write-back cache. This design is an improvement on write-through cache. In write through cache mode, data is written to cache and main system memory at the same time, whereas write back cache minimizes the number of writes to the main system memory, increasing the performance of the CPU. Published benchmarks on the 100 MHz 486DX4 are as follows:

SPECint92/SPECfp92: 48/24 (Snow, 1)

MFLOPS: 2.34 (Juffa, ctest260 result).

3.2 FPU Test

The FPU test is a piece of simple code that was written to get a measure of MFLOPS from the two machines. This code was just for testing raw speed, so it was decided that both machines would work with double precision values. It was first written on the PC in C++, and then later translated into C for the Alpha. It was used to test how fast the two FPU's were on the different types of floating point operations (adds, subtracts, multiplies, and divides). To do this, it was first necessary to decide what type of floating point numbers to work with. It was decided that the most accurate measure would be using irrational numbers because this would force both FPU's to also do some rounding. At this time it was also necessary to decide whether to try to write an algorithm to create irrational numbers, and write them to an array so that they could then

be accessed from inside the loop. It was decided that accessing an array from inside the loop would slow the CPU down and give an inaccurate reading of MFLOPS. So, it was decided that just two irrationals would be used and that they would be defined at the beginning of the program. The two irrationals were defined as pi and e as double precision values.

Then on both machines, two sets of code were run:

1. loops containing all four operations inside (+,-,*,/), and
2. loops containing each operation separately.

These results were then recorded and analyzed.

3.3 The Crank Nicolson Algorithm

After a meeting with Dr. Lawson to discuss a numerical problem that would push a 64 bit processor to the limits, it was decided that the Crank Nicolson Algorithm (found in Numerical Mathematics and Computing, pg. 464) would be used. The Crank Nicolson Algorithm is a method of solving parabolic partial differential equations, and in particular, the heat equation. The example that is used for demonstration in the book is a heat equation with initial boundary conditions:

$$\frac{\partial^2}{\partial x^2} u(x,t) = \frac{\partial}{\partial t} u(x,t)$$

$$u(0,t) = u(1,t) = 0$$

$$u(x,0) = \sin \pi x$$

This example could be explained using a thin rod of length one insulated so that heat can escape only from the ends, which are maintained at a temperature of zero degrees, and

with an initial temperature distribution along the rod. Now this example can easily be solved exactly, and that is why it is used. In the algorithm, the Crank Nicolson solution is compared to the exact solution to show that the Crank Nicolson algorithm is stable. The Crank Nicolson algorithm is of course needed for problems which cannot be solved exactly.

The algorithm found in "Numerical Mathematics and Computing" is designed to compare the results of the two methods, but the reason that this algorithm was chosen for this thesis was to demonstrate that with a 64 bit architecture, using double extended precision would give a more accurate solution than double precision on a 32 bit architecture in a faster time. But because of the fact that the DEC C compiler did not support double extended precision, and the Borland C++ compiler did, this is not what the Crank Nicolson Algorithm was used for. It was decided that the time of the execution of the algorithm would be the result that would be scrutinized. In order to do this, the number of points calculated along the rod had to be increased and the number of points in time t had to be increased so that the timing would give a larger and more accurate number. It was quickly discovered that the C++ compiler could not handle more than a 40 by 150 double precision two dimensional matrix. So the 40 was set as the number of points along the rod and 150 was taken as the number of points in time that the temperature along the rod was calculated. Then, on the DEC Alpha, the dimensions of the matrix were changed and through trial and error a size was found that, when executed, took the same length of time to execute as the algorithm on the PC.

3.4 Eliminating Errors

To begin the comparison, it must be pointed out that the two machines were running on two different operating systems and two different compilers were being used. The Alpha was running a version of UNIX and the PC was running Windows95. As for compilers, the PC was running Borland C++ 4.51, and the Alpha was using a DEC C compiler (which have been said to be among the best compilers around).

The key to testing a CPU for any type of functionality is the availability of a good timing function somewhere in the system. In UNIX, there is a shell timing function which will tell the user several things about an executable that is timed. It will tell the user how much real time was used, how much time the user used, and how much time the system used. This timing function proved to be highly effective. On the PC however, the timing function was found inside C++ and was found to be slightly less than reliable in that the results for the same function timed twice could vary drastically. This was one of the first problems that needed to be addressed.

Also, when comparing the results of the FPU test to published numbers for both machines, it was immediately apparent that there were large discrepancies. After looking more closely at Juffa's ctest260 code written in Pascal, it became evident that Juffa had timed certain functions and then subtracted the clock function time from the total time to give a more accurate MFLOPS reading. It was obvious that this type of approach would be needed to improve the results of the FPU test.

3.4.1 Loop and Declaration Overhead

To eliminate the time that the CPU was spending on incrementing the loop, a loop of different sizes was set up and timed. On the PC, a clock function was set up inside the loop so that this overhead could also be eliminated (this was not needed on the Alpha because the clock function was an external shell function). It became apparent that the clock function in C++ either was not very accurate or had been implemented badly because the results came back varying by up to 3.5% on loops of one million iterations. One of the tests that was run was to time the loop by itself, and then time the loop with the clock function inside it. Then, the time for the loop was subtracted from the loop and clock time, to reveal the time the clock function took by itself. This test actually revealed just how inaccurate the clock function was because roughly half of these calculations resulted in negative values (implying that the clock function took negative time). So, to stabilize the results and get as accurate a number as possible, a one million iteration loop was set up with the clock function in it, and this was timed one thousand times and averaged. By this time the variance was down to 0.3%, and these values were then averaged another ten times to give the result that would be subtracted from the FPU test timings on the PC.

On the Alpha, the results for timing loops were stable, so it was not necessary to go to the same lengths as on the PC. All that was needed here was to time the declaration of variables and loops of three different sizes: one billion, one hundred million and ten million iterations.

3.4.2 Compiler Optimization

During the testing process, the question of compiler optimization arose. It was discovered that both compilers had switches that could be used to try to optimize the code using different techniques. These switches were not used, because it would not be clear exactly what the compiler was doing, and the results would not be as meaningful. So, after comparing the results of the FPU test with published results and, in the case of the PC, the results of the ctest260, and finding that the FPU test gave relatively low numbers, it was decided that some type of optimization must be looked in to. So, a manual "unrolling of the loop" was implemented.

3.4.2.1 Unrolling the Loop

Unrolling the loop is a compiler optimization trick that lets the CPU do a better job of scheduling instructions. It is a method in which the compiler will actually repeatedly write out the code inside a loop and take iterations out of the loop for the compiled version. Thus, the same number of total iterations will be executed, but the compiler will save CPU time for branch penalties, loop incrementation, and counter checking when the code is executed. The user would think that with today's branch prediction algorithms that a branch delay would not waste more than perhaps one clock cycle, but in actuality, even with a good branch prediction, the pipeline will be partially flushed (depending on the machine), resulting in wasted clock cycles. This does not take into account the clock cycles needed to increment the loop and check if the loop has reached the final iteration.

So the method of optimization used on both machines was a manual unrolling of the loop. This was done several times on each machine to get results at several stages of the unrolling. It becomes evident that the optimization reaches a limit and a graph of the MFLOPS plotting different stages of unrolling the loop shows that an asymptote is eventually reached.

On the PC, with both sets of code, the loops were started with between one and four instructions in them and then unrolled several times until there were up to five hundred instructions in them. At this point, it was evident that the limit for optimization had been reached.

On the Alpha, we recall that the pipeline is said to be able to output a 64 bit result for all operations but divide, and that there is a single bit per cycle dedicated divide unit. Well, for the operations add, subtract, and multiply, it was found that the MFLOPS numbers were identical and the limit of optimization was reached when the loop had been unrolled to five hundred iterations. Divide was an interesting case, but it proved to give the expected results. When the loop only had one division in it, the results were not much better than that of the PC. But, as the loop was unrolled, it became apparent that the dedicated divide unit could perform extremely well under the right conditions. The divide operation reached the optimal limit at around two thousand iterations inside the loop. All four operations were also put into a loop and it was unrolled to five hundred iterations before it approached the optimal limit.

3.4.3 Operating Systems

The operating system is a very crucial element when trying to evaluate a computer system's overall performance. The big difference between the PC and a machine like the DEC Alpha is that the Alpha is a multi user system and the PC is a single user system. A multi user operating system is constantly switching between different user's tasks. This is what allows the timing function to be able to output so much information about how much time any one task is using. In contrast to this is the PC, which is a single user system running Windows95. It is unclear exactly what effect this had on the overall results of all the code that was executed on both machines, but it definitely is a factor.

4. Results

4.1 FPU Test

The results for the FPU test were collected from both machines and will be presented first for the 486DX4 and then for the DEC Alpha.

4.1.1 486DX4

4.1.1.1 Eliminating Overhead

To eliminate the overhead, several algorithms were tried until the following results were achieved from the .cpp file "test.cpp". Each value in columns one and two represents one billion iterations each. Column three represents the difference of columns one and two. All times are in seconds.

test #	1. Timing for just loop	2. Timing for loop and clock	3. Timing for just clock
1	0.15363	0.15456	0.00093
2	0.15348	0.15424	0.00076
3	0.15357	0.15474	0.00117
4	0.15351	0.15447	0.00096
5	0.15345	0.15446	0.00101
6	0.15362	0.15450	0.00088
7	0.15355	0.15461	0.00106
8	0.15344	0.15444	0.00100
9	0.15329	0.15449	0.00120
10	0.15366	0.15438	0.00072
ave.	0.15352	0.15449	0.00097

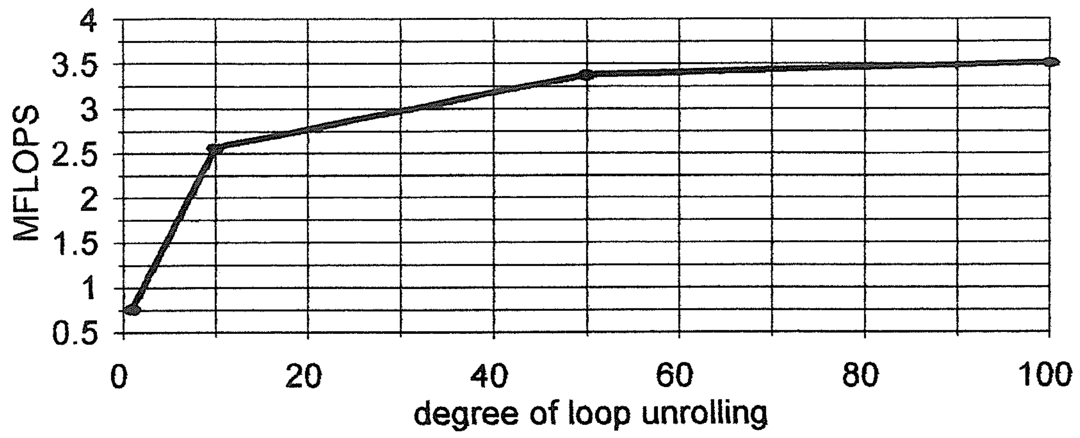
The number that was eventually used for all corrections on the PC was the average from column two, the timing for one million iterations of the loop and clock functions.

4.1.1.2 FPU Test Results

The following table contains the results of both testing the operations separately, and testing all four operations together.

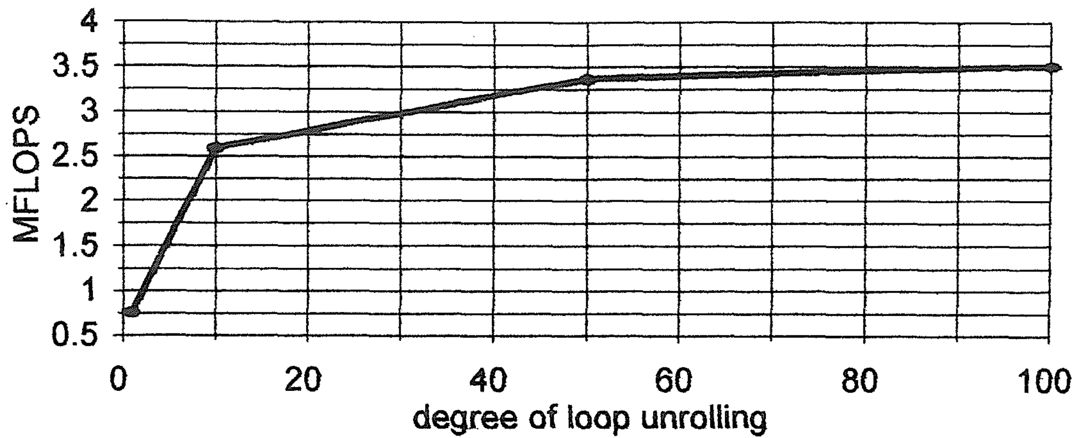
Operation	Degree of Loop Unrolling	MFLOPS
add	1	0.76
	10	2.56
	50	3.38
	100	3.51
subtract	1	0.76
	10	2.59
	50	3.37
	100	3.51
multiply	1	0.67
	10	2.24
	50	2.80
	100	2.89
divide	1	0.50
	10	0.95
	50	1.05
	100	1.06
all operations together	4	1.21
	40	2.07
	400	2.16

4.1.1.3 MFLOPS Graphs



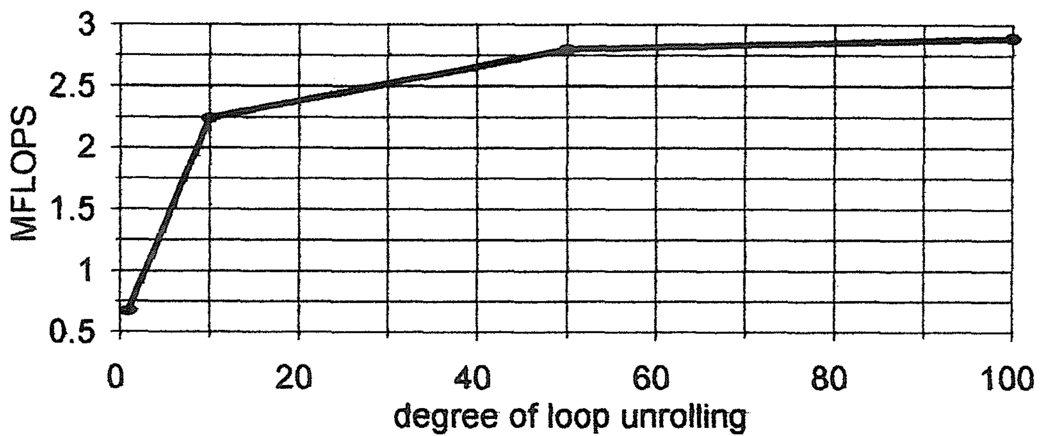
486DX4

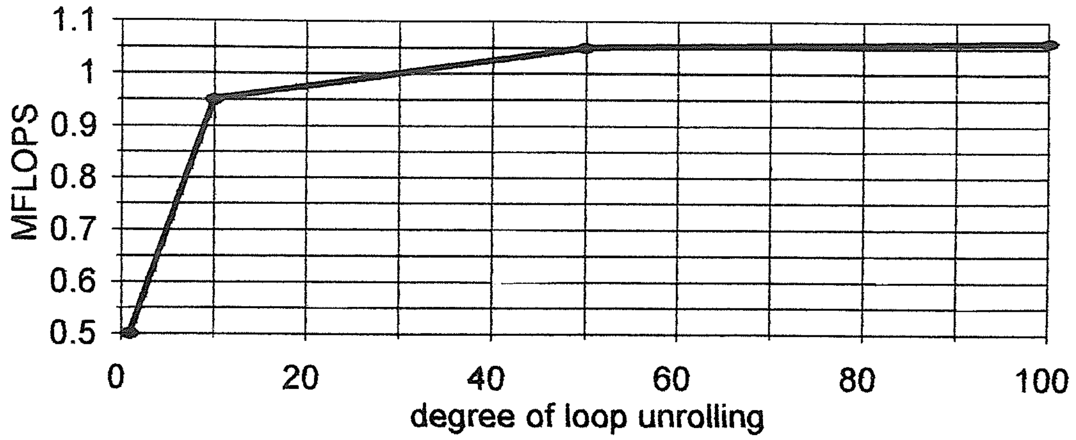
Subtract MFLOPS

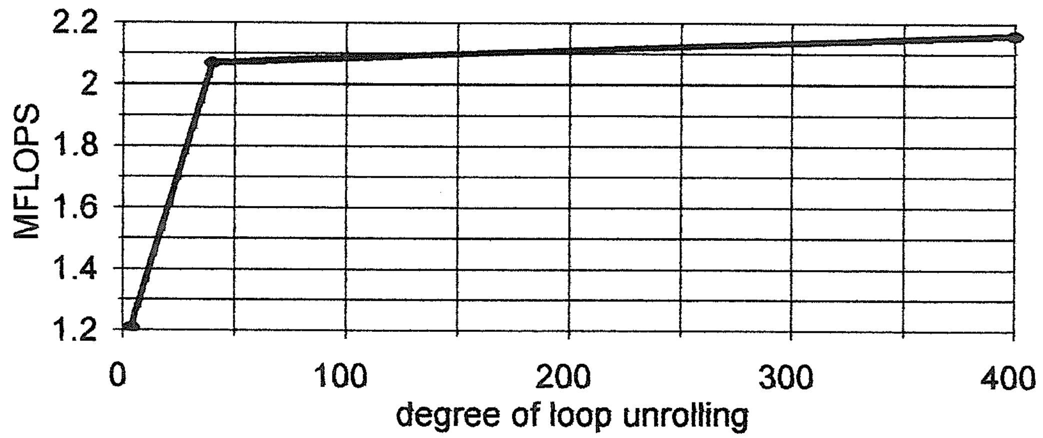


486DX4

Multiply MFLOPS




486DX4
Divide MFLOPS
MFLOPS
1.1
1
0.9
0.8
0.7
0.6
0.5
0
20
40
60
80
100
degree of loop unrolling


486DX4
General MFLOPS
MFLOPS
2.2
2
1.8
1.6
1.4
1.2
0
100
200
300
400
degree of loop unrolling

4.1.2 DEC Alpha AXP 21064

4.1.2.1 Eliminating Overhead

To eliminate the overhead on the Alpha, both the variable declarations and the loop iteration time had to be timed because for the FPU test, the entire piece of code was timed, including declarations. The times were recorded for three different loop sizes and the results were as follows:

Loop Size	Time
10 million	0.4 seconds
100 million	3.7 seconds
1 billion	36.6 seconds

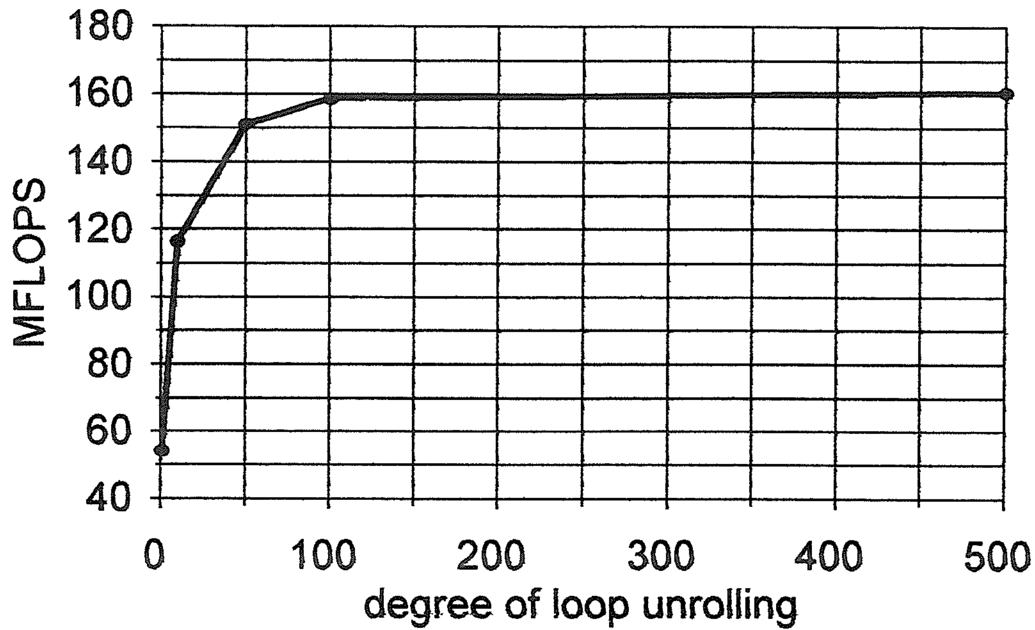
4.1.2.2 FPU Test Results

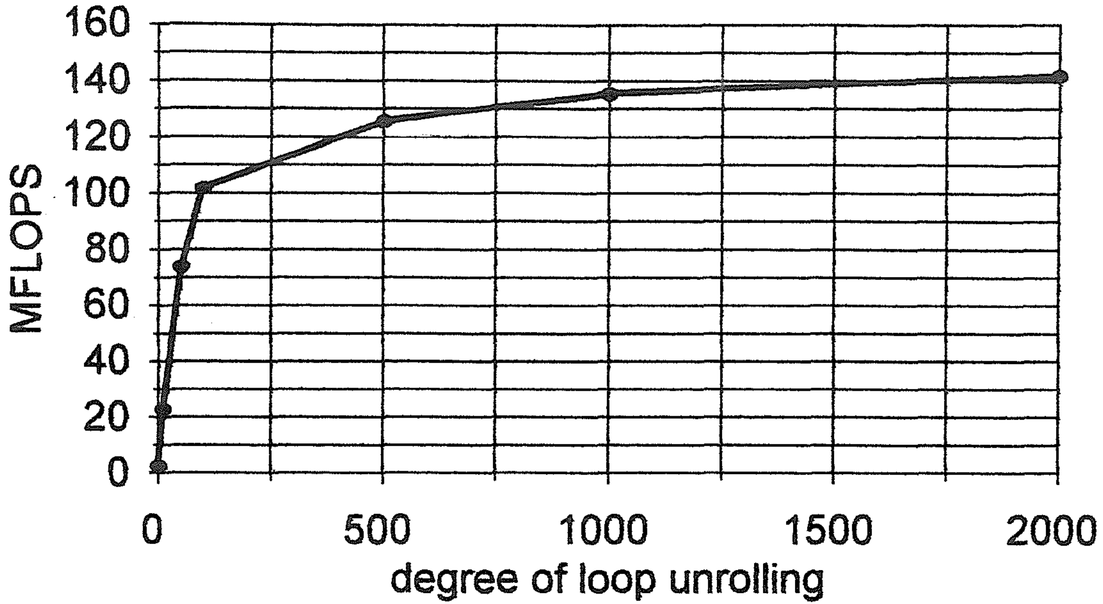
The following table contains the results of both testing the operations separately, and testing all four operations together.

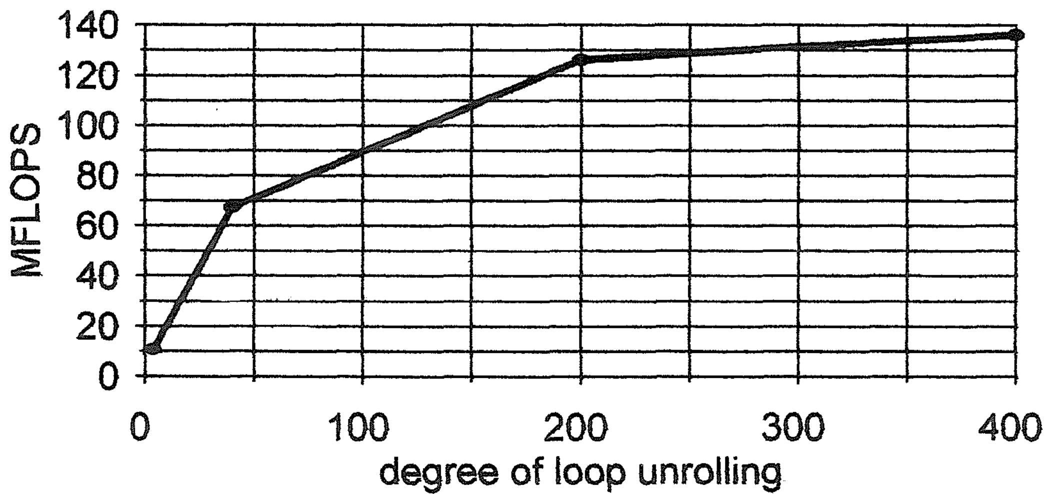
Operation	Degree of Loop Unrolling	MFLOPS
add	1	54.35
	10	116.69
	50	151.33
	100	158.73
	500	160.77
subtract	1	54.35
	10	116.69

	50	151.33
	100	158.73
	500	160.77
multiply	1	54.35
	10	116.69
	50	151.33
	100	158.73
	500	160.77
divide	1	2.73
	10	23.04
	50	73.75
	100	102.15
	500	125.75
	1000	135.67
	2000	142.05
all four operations together	4	10.91
	40	67.57
	200	126.58
	400	136.52

4.1.2.3 MFLOPS Graphs




DEC Alpha AXP 21064
Divide MFLOPS
160
140
120
100
80
60
40
20
0
MFLOPS
0
500
1000
1500
2000
degree of loop unrolling


DEC Alpha AXP 21064
General MFLOPS
140
120
100
80
60
40
20
0
MFLOPS
0
100
200
300
400
degree of loop unrolling

4.2 Crank Nicolson Algorithm

4.2.1 Sample Output

The following is a sample output for m = 20 and n = 10 as the example in the book suggests. This output is identical for both machines because they were both working in double precision.

USING THE CRANK NICOLSON METHOD:

t	u[1]	u[3]	u[5]	u[7]	u[9]
0.000	0.30902	0.80902	1.00000	0.80902	0.30902
0.005	0.29460	0.77127	0.95334	0.77127	0.29460
0.010	0.28085	0.73528	0.90886	0.73528	0.28085
0.015	0.26775	0.70097	0.86645	0.70097	0.26775
0.020	0.25525	0.66827	0.82602	0.66827	0.25525
0.025	0.24334	0.63708	0.78748	0.63708	0.24334
0.030	0.23199	0.60736	0.75074	0.60736	0.23199
0.035	0.22117	0.57902	0.71571	0.57902	0.22117
0.040	0.21085	0.55200	0.68231	0.55200	0.21085
0.045	0.20101	0.52625	0.65048	0.52625	0.20101
0.050	0.19163	0.50169	0.62012	0.50169	0.19163
0.055	0.18269	0.47828	0.59119	0.47828	0.18269
0.060	0.17416	0.45597	0.56361	0.45597	0.17416
0.065	0.16604	0.43469	0.53731	0.43469	0.16604
0.070	0.15829	0.41441	0.51224	0.41441	0.15829
0.075	0.15090	0.39507	0.48834	0.39507	0.15090
0.080	0.14386	0.37664	0.46555	0.37664	0.14386
0.085	0.13715	0.35906	0.44383	0.35906	0.13715
0.090	0.13075	0.34231	0.42312	0.34231	0.13075
0.095	0.12465	0.32634	0.40338	0.32634	0.12465
0.100	0.11883	0.31111	0.38455	0.31111	0.11883

EXACT SOLUTION:

t	u[1]	u[3]	u[5]	u[7]	u[9]
0.000	0.30902	0.80902	1.00000	0.80902	0.30902
0.005	0.29414	0.77006	0.95185	0.77006	0.29414
0.010	0.27997	0.73298	0.90602	0.73298	0.27997
0.015	0.26649	0.69769	0.86239	0.69769	0.26649
0.020	0.25366	0.66410	0.82087	0.66410	0.25366
0.025	0.24145	0.63212	0.78134	0.63212	0.24145
0.030	0.22982	0.60168	0.74372	0.60168	0.22982
0.035	0.21876	0.57271	0.70791	0.57271	0.21876
0.040	0.20822	0.54514	0.67383	0.54514	0.20822
0.045	0.19820	0.51889	0.64138	0.51889	0.19820

Sites, Richard L. "Alpha AXP Architecture", Communications of the ACM, Vol. 36, No. 2, pp. 33-43, February, 1993.

Snow, Gary. Chips and Systems SPEC Chart. Vancouver, WA. WWW search, 1994.

Sterbenz, Pat H. Floating Point Computation. Englewood Cliffs, New Jersey: Prentice Hall, 1974.

Tanenbaum, Richard S. Structured Computer Organization. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

Where m represents the number of times that the temperatures on the rod are calculated and n represents how many points on the rod that the temperatures are calculated. These results show that the Alpha was able to calculate two times as many m values and three and a half times as many n values to the same precision in the same amount of time. This equates to a total of seven times the calculations in the same amount of time.

5. Discussion

The numbers resulting from FPUtest reveal much of what was expected. It is true that both the PC's and the Alpha's MFLOPS numbers were lower than published numbers, but this is easily explained by the simple fact that it is unknown what type of functions the manufacturers used to determine these numbers. The FPUtest gave equal weighting to the four operations, including divide that was noticeably slower than the others. One interesting result that was perhaps expected, was the MFLOPS result from the Alpha on divides. This result showed that if dedicated divide unit is used optimally, it can produce 64 bit divide results almost as quickly as the floating point unit can produce 64 bit results for the other operations. The PC showed that it was not as effective at executing divides, even with optimization, this operation took considerably longer than the multiply, which took slightly longer than the add and subtract.

The Crank Nicolson Algorithm revealed that even though the Alpha can outperform the PC by a factor of roughly 74 in straight MFLOPS, when a real world application without any optimization is tested, the Alpha's performance is cut quite considerably by divides. The Alpha could actually only output roughly seven times as many double precision floating point values in the same amount of time as the PC when running the Crank Nicolson Algorithm.

6. Conclusions

During the testing of these two machines, it became evident that, like MIPS, the term MFLOPS becomes meaningless in comparing the performance of two machines with architectural differences as diverse as a PC and a DEC Alpha. This is shown very well when looking at the results of the FPUtest and then looking at the results drawn from the Crank Nicolson algorithm.

The results of the Crank Nicolson algorithm on the Alpha indicate that if a user is planning to write code for the Alpha for which time is a large factor, it may be wise to try to convert as many of the divides as possible to multiplications, unless the divides can be written in an optimal way.

It can also be observed from the work done in this thesis, that as far as programming environments, the PC with an operating system like Windows95 is not as friendly as the UNIX environment on an Alpha.

7.1

```
//      test.cpp
//      Written for Borland C++ 4.51 on a 486DX4 PC
//      This code times two loops, one that just loops one million times, and the other
//      contains a clock function inside a one million iteration loop.  These results are
//      used to eliminate error from the FPUtest.cpp file.
#include <iostream.h>
#include <time.h>
#include <math.h>
void main()
{
        clock_t start1, end1, start2, end2,start3,end3,start4,end4;
        double ticks,ticks2,adjusted,result;
        long x = 1000000;
        int y = 1000;
        ticks = 0;
        //              FOR LOOP TESTING
        for (int a = 0; a < y; a++)
        {
                start1 = clock();
                for (long i=0; i < x; i++){};
                end1 = clock();
                ticks = ticks + (end1 - start1)/CLK_TCK;
        };
        result = ticks/y;
        cout<<"iterations:          "<<x<<endl;
        cout<<"just loop average:     "<<result<<endl;
        //              CLOCK TIME TESTING
        for (int b=0; b < y; b++)
        {
                start4 = clock();
                for (long j = 0; j < x; j++);
                {
                        start3 = clock();
                        end3 = clock();
                };
                end4 = clock();
                ticks2 = ticks2 + (end4 - start4)/CLK_TCK;
        };
        ticks2 = ticks2/y;
        adjusted = ticks2 - result;
        cout<<"loop and clock average:  "<<ticks2<<endl;
        cout<<"just clock average:     "<<adjusted<<endl;
};
```

7.2

```
//      FPUtest.cpp
//      Written for Borland C++ 4.51 on a 486DX4 PC
//
// This code executes one main loop that contains between one and five hundred floating
// point operations.  The time is recorded and from it is subtracted the overhead time that
// is collected from the "test.cpp" code.  From this number, the MFLOPS are calculated.

#include <iostream.h>
#include <time.h>
#include <math.h>


double a,b,c,t,t2,total,mflops,mflops2;
long x,mil;
int y, unrollfactor;
clock_t start1,end1;

void main()
{
        double adjustment = 0.15449; // result taken from file "test.cpp"
                                     // represents time adjustment for time taken to
                                     // increment loop counter and time to record
                                     // time in clock function
        mil = 1000000;

        a = 2 * (asinl(1.0));        // a and b represent the two floating point
        b = exp(1.0);                // numbers that are used in the calculations of
                                     // the mflops.

        cout<<a<<"   "<<b<<endl;     // Output the two floating point values to be
                                     // used in the calculations.  They are Pi and e.


        x = 1000000;          // Inner loop iteration number
        y = 10;               // Outer loop iteration number(used to average results)
        total = 0;
        unrollfactor = 1;  // The factor with which the loop has been unrolled.
                           // In this case, there is one add operation, so the
                           // unroll factor is 1.
        for (int d=0; d<y; d++)
        {
                start1 = clock();
                for (long e=0; e<x; e++)
                {
```

```
            c=a+b;  //  This is the location of all the floating point
                    //  calculations.  The example here is the add operation
                    //  with a loop unrolling factor of 1.
        };
        end1 = clock();
        total = total + (end1-start1);
    };

    total = total/CLK_TCK;                    //  Calculation of total time
    t = total/y;                              //  Time without adjustment
    t2 = total/y - adjustment;                //  Time with adjustment
    mflops = ((unrollfactor*x)/mil)/t;        //  Mflops without adjustment
    mflops2 = ((unrollfactor*x)/mil)/t2;      //  Mflops with adjustment
    cout<<"MFLOPS without adjustment:  "<<mflops<<endl;
    cout<<"MFLOPS with adjustment:     "<<mflops2<<endl;

}; // end main
```

7.3

```
//      Modified Crank Nicolson Algorithm On The PC
// Source:  Numerical Mathematics And Computing, pg. 464
//      Written for Borland C++ 4.51 on a 486DX4 PC
//
// This algorithm solves partial differential equations
// (specifically heat equations).
// In the book, the algorithm solves the heat equation of a rod
// (initial conditions, see 3.3)
//      In effect, the CPU has to deal with an array of size m by n, containing
// double precision values.  In this environment, the largest allowable array
// size was 150 by 40.
//      This version, for the PC, has output at the end, outside the timing function.
//  This output is just the temperatures along the rod at the last calculated time for both
//  the exact solution and the Crank Nicolson algorithm.

#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include <time.h>

        clock_t start1,end1;
        double pi,xmult,expi;
        int n,m;
        double timer,h,k,r,s,t;
        double c[50];
        double d[50];
        double u[50];
        double v[50];
        double ue[150][40];
        double x[50];

void main()
{
        start1 = clock();
        int count=0;
        int count2=0;
        pi = 2 * (asinl(1));
        n=40;
        m=150;
        h=1.0/n;
        k=h*.05;
        s=(h*h)/k;
        r=2+s;
```

```
for (int i=1; i <= (n-1) ; i++)
{
        d[i]=r;
        c[i]=(-1);
        u[i]=sinl(pi*i*h);
        ue[0][i] = u[i];
};

for (int j=1; j <= m; j++)
{
        for (int i1=1; i1 <= (n-1); i1++)
        {
                d[i1] = r;
                v[i1] = s*u[i1];
        };

        //      PROCEDURE TRI
        for (int i2=2; i2 <= (n-1); i2++)
        {
                int calc = i2 - 1;
                xmult = (-1)/d[calc];
                d[i2] = d[i2] + xmult;
                v[i2] = v[i2] - (xmult * v[i2-1]);
        };

        x[n-1]= v[n-1]/d[n-1];

        for (int i3 = n-2; i3 >= 1; i3--)
        {
                x[i3] = (v[i3] + x[i3+1])/d[i3];
        };

        for (int i9 = 1; i9 <= n-1; i9++)
        {
                v[i9] = x[i9];
        };
        // END PROCEDURE TRI


        t = j*k;
        expi = exp((-t) * pi * pi);


        //              EXACT METHOD
        for (int i4=1; i4 <= (n-1); i4++)
```

```
            {
                    u[i4] = expi * sin(pi * i4 * h);
                    ue[j][i4] = u[i4];
            };

            for (int i5=1; i5 <= (n-1); i5++)
            {
                    u[i5] = v[i5];
            };
    };
    end1 = clock();
    cout<<endl;
    timer = (end1 - start1)/CLK_TCK;

    printf("Total execution time (without I/O):  ");
    printf("%1.2f\n\n",timer);

    printf("  At time t = 0.1875 (the last iteration): \n\n");
    printf("  Using the Crank Nicolson Method \n");

    for(int i8=1; i8<=n-1; i8++)
    {
            count2 = count2 +1;
            if(count2==8){printf("\n");count2=1;};
            printf("%1.5f  ",u[i8]);
    };

    printf("\n\n");
    printf("  The Exact Solution \n");

    for (int i6 = m; i6 <= m; i6++)
    {
            for (int i7 = 1; i7<=n-1; i7++)
            {
                    count = count + 1;
                    if (count == 8){printf("\n");count=1;};
                    printf("%1.5f  ",ue[i6][i7]);
            };
    };

};
```

7.4

```
//      test.c on the Alpha
//      Translated from C++ to C
//
//      This code tests the time the machine requires to declare variables and
// execute loops containing nothing different numbers of times.  The results were
// used to adjust the results from FPUtest.c.

#include <math.h>
#include <float.h>

void main()
{
        double a,b,c;
        long int x,e;

        a = 2 * asin(1.0);
        b = exp(1.0);
        x = 10000000;

        for (e=0; e<x; e++)
        {
        };

};
```

7.5

```
//      FPUtest.c on the Alpha
//      Translated from C++ to C
//
//      This algorithm executes a loop containing different floating point operations.
//      The different operations were entered in the loop and unrolled to different
//      degrees. Correction factors for loop and declaration overhead were taken from
//      test.c.

#include <math.h>
#include <float.h>

void main()
{
        double a,b,c;
        long int x,e;

        a = 2 * asin(1.0);
        b = exp(1.0);
        x = 100000000; // This is where the loop iteration number is entered.
                       // This was either 10 million, 100 million or 1 billion.

        for (e=0; e<x; e++)
        {
                c=a+b; // all floating point instructions were entered in this
                       // loop. The example here is of an add instruction
                       // with no loop unrolling.
        };

};
```

7.6

```
//      Modified Crank Nicolson Algorithm On The Alpha
// Source:  Numerical Mathematics And Computing, pg. 464
//      Translated from C++ to C
//
// This algorithm solves partial differential equations
// (specifically heat equations).
// In the book, the algorithm solves the heat equation of a rod
// (initial conditions, see 3.3)
//      The size of the array used on the Alpha was 300 by 140.

#include <float.h>
#include <math.h>
#include <stdio.h>

        double pi,xmult;
        int n,m,i,i1,i2,i3,i4,i5,i6,i9,j;
        double h,k,r,s,t,expi;
        double c[300];
        double d[300];
        double u[300];
        double v[300];
        double ue[450][300];
        double x[300];

void main()
{
        pi = 2 * (asin(1.0));
        n=140;
        m=300;
        h=1.0/n;
        k=h*0.05;
        s=(h*h)/k;
        r=2+s;
        // setting the u array values
        for (i=1; i <= (n-1) ; i++)
        {
                d[i]=r;
                c[i]=(-1);
                u[i]=sin(pi*i*h);
                ue[0][i] = u[i];
        };

        for (j=1; j <= m; j++)
        {
```

```
			for (i1=1; i1 <= (n-1); i1++)
			{
				d[i1] = r;
				v[i1] = s*u[i1];
			};

			// Procedure TRI (Numerical Mathematics and Computing, pg. 251)
			for (i2=2; i2 <= (n-1); i2++)
			{
				int calc = i2 - 1;
				xmult = (-1)/d[calc];
				d[i2] = d[i2] + xmult;
				v[i2] = v[i2] - (xmult * v[i2-1]);
			};

			x[n-1]= v[n-1]/d[n-1];

			for (i3 = n-2; i3 >= 1; i3--)
			{
				x[i3] = (v[i3] + x[i3+1])/d[i3];
			};

			for (i9 = 1; i9 <= n-1; i9++)
			{
				v[i9] = x[i9];
			};
			//	End Procedure Tri

			t = j*k;
			expi = exp((-t)*(pi*pi));

			//	Exact Method
			for (i4=1; i4 <= (n-1); i4++)
			{
				u[i4] = expi * sin(pi * i4 * h);
				ue[j][i4] = u[i4];
			};
			//	End Exact
			for (i5=1; i5 <= (n-1); i5++)
			{
				u[i5] = v[i5];
			};
		};
	};
```

Bibliography

Cheney, Ward et al. Numerical Mathematics and Computing. Pacific Grove, California: Brooks/Cole Publishing Company, 1994.

Dobberpuhl, Daniel W. et al. "A 200-MHz 64-bit Dual-Issue CMOS Microprocessor", Digital Technical Journal, Vol. 4, No. 4, pp. 35-50, Special Issue 1992.

Edmondson, John H. et al. "Internal Organization of the Alpha 21164, a 300-MHz, 64 bit Quad-issue CMOS RISC Microprocessor", Digital Technical Journal, Vol. 7, No. 1, pp. 119-135, 1995.

Goldberg, David. "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, Vol. 23, No. 1, pp. 5-49, March, 1991.

Juffa, Norbert. "Everything You Always Wanted To Know About Math Coprocessors", October, 1994, WWW search.

Kantrowitz, Michael et al. "Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor -- the Alpha 21164 CPU chip". Digital Equipment Corporation. WWW search.

Morse, Stephen. "Benchmarking The Benchmarks", Network Computing, pp. 78-84, February, 1993.

NexGen Inc. "When is a Floating Point Useful?", August 1995, WWW search.

Shiflet, Angela B. Problem Solving in C Including Breadth and Laboratories. New York, New York: West Publishing Company, 1995.

Sites, Richard L. "Alpha AXP Architecture", Communications of the ACM, Vol. 36, No. 2, pp. 33-43, February, 1993.

Snow, Gary. Chips and Systems SPEC Chart. Vancouver, WA. WWW search, 1994.

Sterbenz, Pat H. Floating Point Computation. Englewood Cliffs, New Jersey: Prentice Hall, 1974.

Tanenbaum, Richard S. Structured Computer Organization. Englewood Cliffs, New Jersey: Prentice Hall, 1990.