Computer Architecture Laboratory

Performance Measurements

The purpose of this lab is to measure the time needed to perform basic arithmetic operations on a Pentium (or Pentium II or Pentium III) processor. By using the library-supplied time function, the time needed to perform an arithmetic operation a large number of times may be measured (perform the operation 10^9 times for example). From this data, the time for one operation may be calculated. (Hint: when using very large integers, be careful that your data type is able to contain the number, i.e. think "long" not "int")

Your experiment must take into consideration the loop overhead, and therefore the measurement of an empty loop (or loops) must be made and its time subtracted from the measured time for the loop containing the operation. In addition, the compiler may provide additional complications. To look at compiler complications consider the following:

- If the operation C = A + B is looped, does the compiler create code that fetches the data A and B, and store C on each operation thereby measuring data fetch/store as well as data addition? Or are those numbers prefetched and stored in a pre-fetch buffer?
- Does the result differ if A and B are defined as constants in a #define statement instead of as variables?
- Does the result differ if explicit constants are used (i.e. is there a difference between C = A + B where A and B are variables, and C = 2 + 3 ?)
- Is there a difference if the use of registers for storage of A and B is forced? (Read the Microsoft C++ Help Files for more information on the forcing of registers.)
- What is the impact of putting the instruction in assembly code? (See Appendix C.)

To answer these questions read Chapter 8 "Hypotheses and Experiments" in Writing for Computer Science by Justin Zoble and try to formulate short experiments to determine the answers. In developing a methodology for your experiments, you must answer these types of questions and discuss your approach in the "methodology" section of your lab report (see below). It may be that you just have to "live with" measuring both the data fetch/store and the operation. In that case, you must decide -based on your data - if any of your measurements have any meaning (remember, that relative measurements may give useful information.)

Execute your programs from .exe files, not from within the development environment. There may be overhead from that environment. Also, be sure to compile as a "release version" that has all debug information turned off. Debug information would add overhead. (See Appendix A)

Do experiments to make the four comparisons which follow on the next page. Give the basic data in an appendix and a summary of the data in the body of the report. Use a combination of text, tables and graphs to aid in the explanation of your results.

Report all results both in

- absolute time in nanoseconds (ns) and
- number of clock cycles (i.e. you'll have to know the clock speed of the machine you use.)

Whenever you report a result such as "Operation X is n% faster than Operation Y" remember the standard definitions of speedup (and use them!)

Comparison #1 Measure the time for the following operations

+ - * / % between two integers

- Analyze the relative performance of the various arithmetic operations for integers
- Explain what you have observed using words, tables and graphs.

Comparison #2 Measure the time for the following operations:

+ * / between two floats

- Compare the performance of these operations with their integer counterparts
- Explain what you have observed using words, tables and graphs.

Comparison #3 Measure the time of the FP operations:

sqrt log sin

- Compare the times necessary to do these operations with the times of simpler arithmetic operations done in Comparison #2.
- Explain your comparisons using words, tables and graphs.

Comparison #4 Turn off the numerical processor (see Appendix B) and use floating point emulation to examine the difference between doing FP in hardware and in software.

Using software emulation, measure the execution time for the FP operations:

+ * / sqrt log sin

- Compare your results with those obtained from Comparisons #2 and #3 where the FP calculations were done in hardware.
- Explain your comparisons using words, tables and graphs.
- Reconcile your results with the estimate you made of speedup for the FPU made in the lab where you wrote an FP addition using only integer arithmetic.

The Lab Report

Before beginning the Lab Report, read the following chapters from Writing for Computer Science by Justin Zoble:

- 1 Designing an Article
- 2 Writing style: general guidelines
- 3 Writing style: specifics
- 6 Graphs, figures, and tables
- 9 Editing

Please follow the advice given in this text when writing your lab report.

The Lab Report should contain:

1. Goals: Explain the purpose of the measurements and discuss the four different comparisons.
2. Methodology: Explain the method used to make the measurements. A printout of the code used should be included in an appendix.
   - Give an explanation of any weaknesses of the method.
   - Be sure to address the questions asked on the first page of the lab sheet, the questions about the use of variables and constants.
   - Describe the experiments you performed to determine answers to these questions, give the data you obtained from these experiments, analyze the data and present your conclusions justifying your methodology
3. Results: Give the data obtained in each of the comparisons. If multiple runs are used (is this a hint?), put the raw results in an appendix and give the averages as the data. Report all results in both clock cycles and nanoseconds. Use tables to group results coherently.
4. Analysis: Analyze your data and draw conclusions for each of the comparisons requested. This is probably most meaningful when reported in clock cycles. Use graphs to show your comparisons.

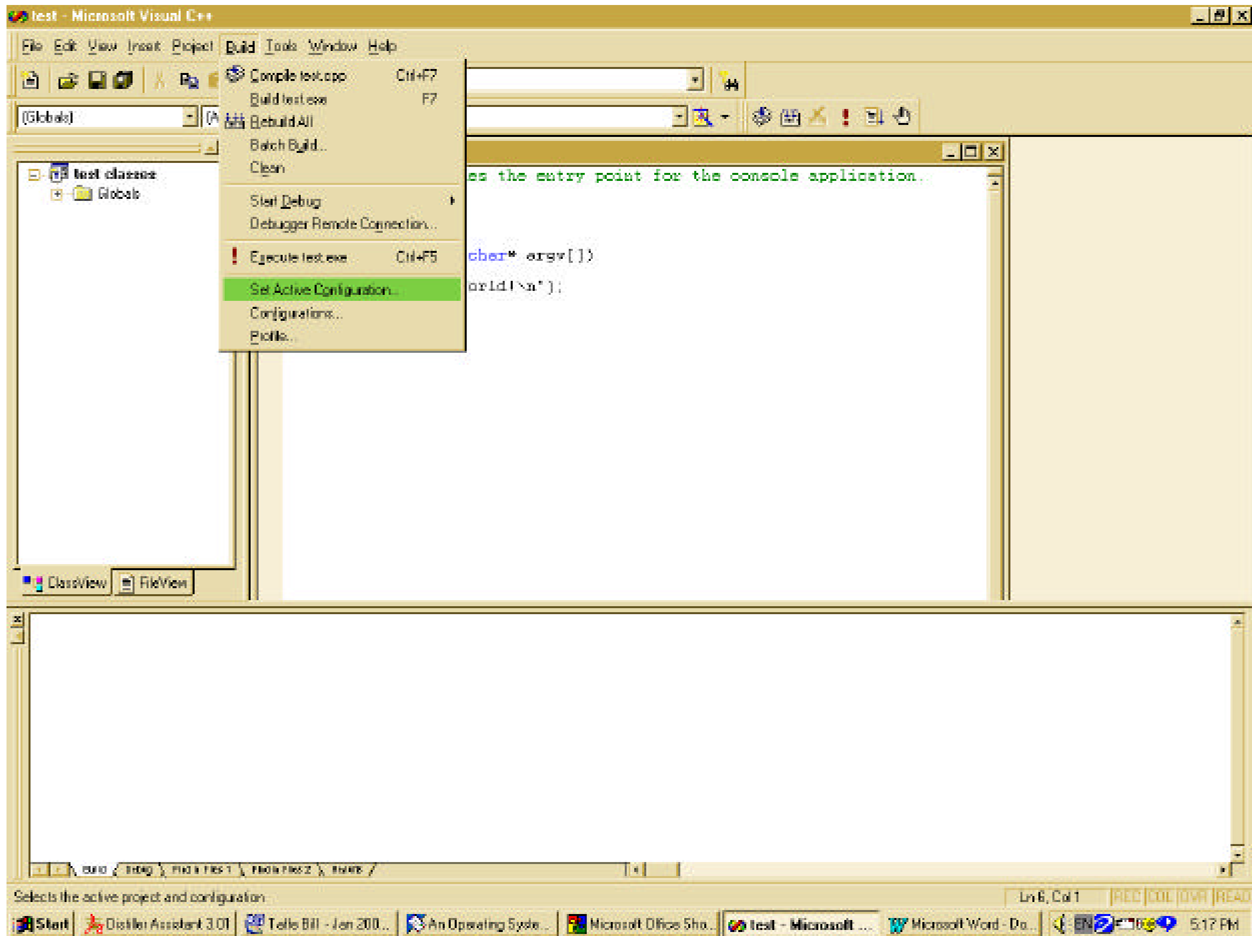
You have two weeks to do this lab and one additional week to do the report. Therefore the report is due on November 16. (Note: there is no lab the preceding week ... Nov 9 ... there is an exam during the lab time.)

This report is a major writing assignment. It is important to do it well and thoroughly. A "slap – dash" incomplete scribble will not be accepted.

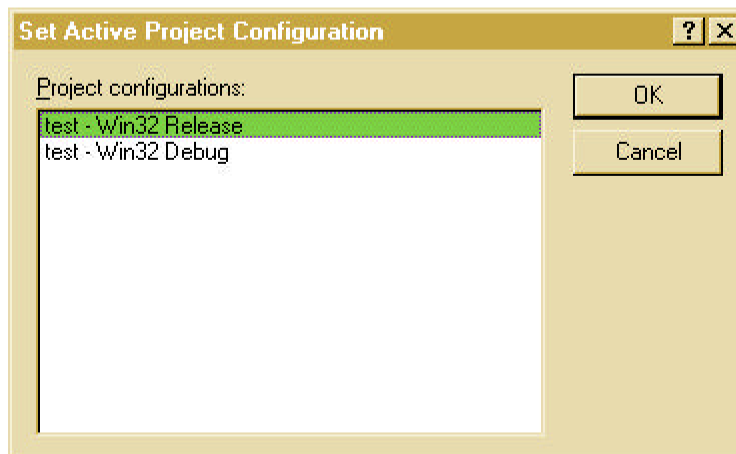
Appendix A

Generating the Release Version

To turn off debug information or generate the release version,
choose Build -> Set Active Configuration



This will bring a window (see below) that permits the selection of the “release version” rather than the “debug version” that has all the debug code and information in it.



Appendix B

Turning Off the Numerical Processor

Windows 95

This method works on Windows 95A … the release B (Windows 95B) "seems to support it" but crashes!

⇨ Right click on "My Computer", then choose Properties -> Device Manager -> System Devices -> Numeric Data Processor -> Settings -> "Never Use the Numeric Data Processor"

⇨ Reboot the computer. (This is where Windows95B hangs the system.)

⇨ Set the compiler option in Visual C++ to emulate FP in software instead of doing it in hardware.

Windows 98

Not yet tried but I think that in MyComputer/Properties you can get a list of the hardware devices and disable the numeric processor. It is very similar to the Windows 95 method.

Windows NT (thanks to Steven Fonden for finding this.)

Pentnt Detects floating point division error (if present) in the Pentium chip, disables floating point hardware, and turns on floating point emulation.

pentnt [-c] [-f] [-o] [-?|-h]

Parameters

-c Enables conditional emulation. Floating-point emulation is forced on only if the system detects the Pentium processor floating-point division error at start time. If you select this parameter, you must restart the computer for the changes to take effect.

-f Enables forced emulation. Floating-point hardware is disabled and floating-point emulation is always forced on, regardless of whether the system exhibits the Pentium processor floating-point division error. This parameter is useful for testing software emulators and for working around floating-point hardware defects known to the operating system. If you select this parameter, you must restart the computer for the changes to take effect.

-o Disables forced emulation and reenables floating-point hardware if it is present. If you select this parameter, you must restart the computer for the changes to take effect.

-?|-h Displays Help for the command.

Windows 2000

??? Not yet tried but I think it's the same as the NT.

Appendix C
Using 80x86 Assembly Language within a C program

INTERFACING C WITH ASSEMBLY LANGUAGE

At first it may seem that there is no need to use assembly language in C, because C emulates so much of what assembly does. Even interrupts are used in C. There are a few advanced topics that C cannot do and that must be done in assembly. A sample is listed below:

- Assessing the stack or base pointer registers.
- Setting some segment registers.
- Performing PUSH and POP operations (particularly the flag register).
- Performing real-time operations where speed is important. This could be reading and writing to an I/O port or writing graphics to the screen.

These topics become important when writing advanced operations such as terminate and stay resident (TSR) programs, device drivers, and interrupt service routines.

IN-LINE ASSEMBLY

In-line assembly is accomplished through the _asm command. It was covered briefly in Chapter 9. You can use _asm as a one-line command:

```
_asm mov dl,4
```

or as a multiline function:

```
_asm{
mov ah,2
mov dl,2
int 21h
}
```

By the way, this small routine prints a happy face (ASCII 2) to the screen. If you are using Borland or Turbo-C, enter asm instead of _asm. Suppose you want to obtain the current address of the base of the stack. The stack pointer will provide this information. The program below returns **sp** into a variable called StackPointer.

```
#include <stdio.h>
void main()
{
unsigned StackPointer;
_asm
{
        mov StackPointer,sp
}
        printf("StackPointer = %u",StackPointer);

}
```

The small program hints at the true power of in-line assembly. You can create variables in C (such as StackPointer) and use these variables in the _asm function with 80x86 registers.

Register names of 80x86 x ³ 3

	31 ... 16	15 ... 8	7 ... 0
EAX		*AH* AX	*AL*
EDX		*DH* DX	*DL*
ECX		*CH* CX	*CL*
EBX		*BH* BX	*BL*
EBP		BP	
ESI		SI	
EDI		DI	
ESP		SP	

Bit positions marked: 31, 23, 15, 7, 0

Fig. 1-7. General registers.

Arithmetic Instructions for 80x86

Mnemonic:	**ADD** ADD *destination,source*
Description:	Adds a source to a destination. The result is stored in the destination register.
Example:	ADD AX,13 (Adds 13 to AX. Result is in AX.) ADD AL,DL (Adds DL to AL. Result is in AL.)
Flags Affected:	Carry, Overflow, Parity, Sign, Zero

Mnemonic:	**SUB** SUB *destination,source*
Description:	Subtracts a source from a destination. The result is stored in the destination register.
Example:	SUB AX,13 (Subtracts 13 from AX. Result is in AX.) SUB AL,DL (Subtracts DL from AL. Result is in AL.)
Flags Affected:	Carry, Overflow, Parity, Sign, Zero

Mnemonic:	**MUL** (Byte version) MUL *8 bit source*
Description:	Multiplies AL by a source value. The source can be a register or an address. The result is stored in AX.
	(Word version) MUL *16 bit source*
Description:	Multiplies AX by a source value. The source can be a register or an address. The high 16-bit result is stored in DX, the lower 16-bit result is stored in AX.
Example:	MUL DL (Multiplies AX by DL. Result is in AX.) MUL BX (Multiplies AX by BX. High result is in DX, low result is in AX.)
Flags Affected:	Carry, Overflow, Parity, Sign, Zero

Mnemonic:	**DIV** (Byte version) DIV *8 bit source*
Description:	Divides AX by an 8-bit source value. The source can be a register or an address. The result is stored in AX. The quotient is in AL, and the remainder is in AH. (Word version) DIV *16 bit source*
Description:	Divides AX by a source value. The source can be a register or an address. The quotient is in AX, and the remainder is in DX.
Example:	DIV DL (Divides AX by DL. Remainder is in AH, quotient is in AL.) DIV BX (Divides DX:AX by BX. Remainder is in DX, quotient is in AX.)
Flags Affected:	Carry, Overflow, Parity, Sign, Zero