PRACTICAL 3: PENTIUM ASSEMBLY LANGUAGE

LEARNING OUTCOME

On completion of this exercise you should have gained an understanding of the key features of a Pentium processor by observing the execution of an assembly language program, and the effect this has on the registers. This will also help you understand how a processor can make decisions using the flags and conditional instructions.

INTRODUCTION

The purpose of this lab is to introduce the way in which instructions are executed inside a Pentium microprocessor. The internal operation of the processor will be demonstrated by stepping through a low-level program, while observing the contents of the registers. This will give an understanding of the basic operation of the microprocessor, and its abilities to do arithmetic operations, make decisions, and control the flow of program execution.

BACKGROUND

In order to be able to see what happens when a program executes inside a processor, it is necessary to have software that will display the contents of registers, and allow you to step through a program one line at a time. These features are provided by a debugger, a program designed to help debug programs as they are being developed. This handout has been written assuming that you will be using Visual C++, which has a suitable debugger built in. There are many other possible debuggers; for instance, Borland produce a stand-alone debugger (Turbo debugger) that could have been used instead of the Visual C++ system.

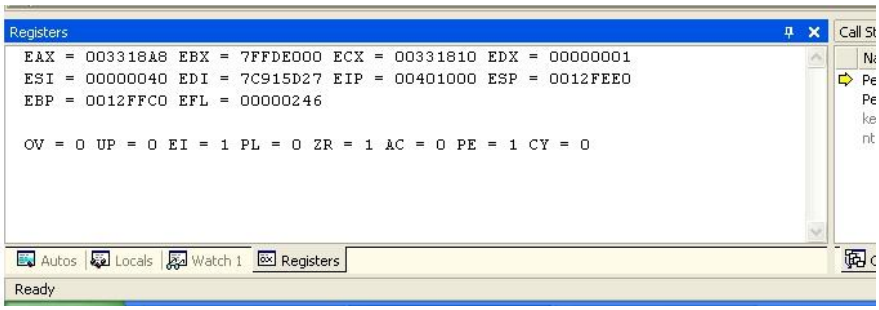
PROGRAM

I have written a single program for you to step through in order to see what is happening inside the microprocessor. In fact, the program consists of a number of separate sections, which will be explained as you work through the exercise. I have combined them into a single program, just to keep things simple (the program is listed at the back of this handout).

PROCEDURE

In principle the procedure is straightforward:

- Start Visual C++
- Load the program and build it
- Start the debugger
- Step through the program a line at a time, following the description in this worksheet.



In practice there is a bit more to it than that. The main thing to remember is that it is quite possible to step through the program quickly. You will only learn what is going on if you work through it slowly, checking carefully to see what happens to the registers as you go.

REGISTERS

The debugger shows the registers in a window like the one shown. If it does not, you may have to add the register window to the view once you have started debugging. Any value that has just changed is highlighted in red. The 4 general purpose registers (EAX, EBX, ECX & EDX) are shown first. These will be the main ones to watch in this lab. Note also the Extended Instruction Pointer (EIP). This increases in value as a program is stepped through. The Extended Flags register (EFL) is also shown. For increased visibility, individual flags are 'broken out' from this: for instance, ZR is the Zero flag, CY is the Carry flag.

RUNNING THE PROGRAM

To run the program, you need to start Visual C++ (The lab demonstrator will give you specific instructions: these depend on the lab you are in). You don't need to know anything about C++, it is just being used as a way to get a debugger running. The program has been saved as PentiumProg.cpp. There are a few files that are needed, so copy across the whole of the PentiumProg folder to your local C: drive if it is not already on your PC. Again, the demonstrator will give specific instructions for this.

Having done this, it is now possible to compile the program. The quickest way is to use the **Debug..Step Into** option on the menu. This compiles the program, and executes one line of code. There is a short-cut key: **F11**. Pressing this will execute one line of code, specifically, the line that the cursor is at the start of. The only thing you might have to do, is to get the registers visible using **View..Debug Window > Registers. I**f you do not see the individual flags, you may have to right-click on the registers window and select **flags** from the optional registers that can be viewed.

Now, all you need to do is to go through the program pressing F11 to execute a line of code. Use the notes below to guide you through what is happening. If you want to go back to the start of the program, use **Debug..Restart** from the main menu. If you want to quickly jump ahead through a section, insert the cursor where you want to go (move the

cursor to the insertion point, then left-click it), then use **Debug..Run to Cursor**. You can run through the program as many times as you want.

SECTIONS OF THE PROGRAM

The program is organised into a number of sections, separated by NOPs. NOP is short for No Operation, and just means that the processor idles for a cycle. I have just used these as spacers to delimit the sections of the program.

SECTION 1: REGISTERS.

The first section shows how the registers work; in particular how 8, 16 or 32 bit quantities can be manipulated. All the instructions use the 'accumulator', or 'a' register

```
1  mov eax, 0x0          //1: registers
2  mov al,0x11
3  mov ah,0x22
4  mov ax, 0x3333
5  mov eax, 0x1234abcd
```

The first line puts a 32 bit value into eax. In hexadecimal, a 32 bit number contains 8 digits, so I could have written mov eax, 0x00000000, but the compiler is smart enough to allow leading zeroes to be omitted. Notice the prefix 0x that denotes a hexadecimal number. You should see the contents of eax change to 00000000. The second line puts an 8-bit value in al, the lower of the two 8-bit sections of the accumulator. You should see the bottom section of eax change. Line 3 does a similar thing with ah, the higher of the two 8-bit sections. The 4th line use ax, which is just the two 8-bit registers grouped into one. The strange naming scheme is a relic of the early days of this family: al, ah and ax were the three registers that were available. All the processors from the 80386 onwards also contain eax, which extends the accumulator to 32 bits. Line 5 shows a number being put into this extended 32-bit register. Once you get to the end of section 1, there are a couple of nops to execute before getting to the next section. As you execute these, the only thing that will change is eip, the extended instruction pointer.

SECTION 2: ARITHMETIC

This section sows how a few simple arithmetic instructions work. There are more complicated ones for multiplication, division, and working with floating point numbers, but these are beyond the scope of this course.

```
1  mov eax, 0x90         //2: arithmetic
2  mov ebx, 0x10100088
3  add eax, ebx
4  mov ecx, 0x20000000
o  sub ecx, eax
```

Lines one and two put 32 bit numbers into eax and ebx. Line 3 takes the contents of the two registers and adds them together. The answer goes into the first register specified (i.e. eax). You should see that after line 3, the contents of eax has changed, but not the contents of ebx. (Aside: 90 in hex is the same as 144 in decimal. Adding this to 0x10100088, which is 269484168 in decimal, should give 0x10100118, i.e. 269484312 in decimal).

Line 4 moves another number into ecx, then line 5 does a subtraction. Note that for an instruction of the form: *sub m,n* the number in *n* is subtracted from the number in *m* and the result is placed in *m*. The result is of course given in hexadecimal. (If you want to convert this number to decimal, the easiest way is to use the scientific version of the Windows calculator). Inside the microprocessor, the number is held as a 32-bit binary number. This is almost impossible to display meaningfully, so the display is given in hex (four bits make up each hex digit).

SECTION 3: VARIABLES IN MEMORY

In the previous section, all the operands were contained either directly in the instructions, or in the registers (what is called immediate or register addressing). However, for most programs, it is necessary to get numbers from variables (i.e. sections of main RAM) into and out of the processor. In the example program, I declared three variables: Var1, Var2 and Result at the beginning. The section of code actually implements the C++ statement: **Result = Var1 – Var2**;

```
1  mov eax, Var1         //2: variables in memory
2  mov ebx, Var2
3  sub eax, ebx
4  mov Result, eax
```

In the first two lines, the contents of the variables are copied from RAM memory into the registers of the processor. If you were observant you might have spotted that the numbers appear differently in the registers. This is because I put starting values in the variables right at the beginning of the program. The values I used, 11 and 18, were in decimal. Inside the processor, they are displayed in hex: 0B & 12. In line 3, ebx is subtracted from eax, and the result is put in eax. Finally, the answer is moved out of the processor's accumulator register and written to the variable in RAM. There is also a variable window that shows the contents of the variables directly (you may have to use **View..Debug Windows > Variables** to see this). The subtraction, 18 – 11, gives the correct result: -7. Do you understand why this is represented as FFFFFFF9 within the processor? If you are unsure on this, the CS1 notes covered the appropriate number systems.

SECTION 4: FLAGS

This section does a simple bit of arithmetic in order to show how the flags give information about results of operations.

```
1   mov eax, 0x90000000     //3: flags
2   mov ebx, 0xa0000000
3   add eax, ebx
4   sub eax, 0x30000000
```

Lines 1 and two load some starting numbers into registers eax and ebx. Line 3 adds the two numbers together and puts the result into eax. However, the result of adding the two numbers should be 0x130000000, which is 33 bits long, too big to fit in the register. The bottom 32 bits are held in eax, and the Carry flag is set to one to show that there is a bit to carry. This bit is quite difficult to see, and you may have missed it. If you are not sure what happened, go through it again. While you are executing some of the instructions, (such as the simple *mov* instructions) the flags do not change. When an arithmetic instruction is executed, the flags may change depending on the result of the operation. They can then be used by later sections of the program (for instance to allow for the carry).

SECTION 5: DECISIONS

I have split this part into two. The idea here is to see how decisions can be made by the microprocessor. The format is always the same: do something that will affect the flags, then branch one way or another depending on the contents of the flags. The difference between the two parts is that in one, the branch is taken, in the other, it is not.

```
1          mov al, 0x11          //5a: decisions
2          mov bl, 0x11
3          sub al, bl
4          jz _EQ
5          mov cl, 0x00
○          nop
7   _EQ:   mov cl, 0xff
```

The first two lines move numbers into registers. Line three does a subtraction. As the numbers were the same, the result (in al) is zero, so the Zero flag is set (you should see ZR =1). Line 4 is the one that makes the decision. The instruction, jz, is short for 'Jump if Zero'. As the zero flag is set, the jump is made to the label _EQ. In other words, lines 5 and 6 are skipped, and line 7 is executed after line 4. If the numbers had been different, the answer would not have been zero, and lines 5 & 6 would have been executed.

```
1   _EQ:    mov cl, 0xff
1.          nop
2.          mov al, 0x11          //5b: decisions
3.          mov bl, 0x22
4.          sub al, bl
5.          jz _EQ
6.          mov cl, 0xaa
```

The next section really starts at line 3 above. Two numbers are loaded into registers and subtracted. As the numbers were different, the result is not zero, so the zero flag is cleared. Then, in line 6, the jump is not made, and execution continues normally at line 7. If the jump had been made, the processor would have gone back to line 1 (where I inserted the label _EQ). This should give an idea of how loops can be set up. Note that there are more conditional jumps than the one I have shown (jump if not zero, jump if equal, jump if greater than, jump if equal, jump if less than..).

SECTION 6: LOOPING

The final piece of code demonstrates the use of a special instruction (loop) to perform a repeated loop, rather like a 'for' loop in a high level language.

```
1          mov eax, 0x0          //6: looping
2          mov ecx,  0x04
3   _AA:   add eax, 0x03
4          loop _AA
```

Line 1 clears the eax register. Line 2 puts a count value in ecx (note: this is the only register that can be used in this way). Line 3 is the body of the loop: every time round the loop, 3 is added to the contents of eax. Line 4 is the decision part of the loop. The special instruction 'loop' decrements ecx, then loops back if ecx hasn't yet reached zero (you should be able to see ecx decreasing as you go round the loop). After four times round, the value of ecx drops to zero, the loop terminates, and execution continues normally.

CONCLUSIONS AND FURTHER WORK

Firstly, if you have reached this far, well done (assuming of course that you have actually followed roughly what is going on!). What I have tried to do is give an idea of how a microprocessor works at a fairly low level. This has not made you an expert assembly language programmer, but that was not the intention. Very rarely do people have to write assembly language nowadays. The commonest exceptions to this are when talking to hardware, or when out to get the fastest performance (such as in games programming). Obviously, nobody would write in assembly language if they could use a high-level language. What you should have is an idea of some of the features and limitations of a modern microprocessor. For instance the fundamental limitations of register size meaning that everything has to be in 8, 16 or 32 bits, and the fact that only ecx can be used as a loop counter.

If you are interested, there is a lot you can do to adapt the program to see the effects of changing the code. If you stop the debugging, then you can easily alter the code. Once you have made a change, pressing F11 will bring up a box telling you that files are out of date, and asking if you want them re-built. Select yes, and the debugger will re-compile and start the new program. Try a simple change first, such as altering the value stored in a register, or the actual register that is used. For instance, in the last example, you can change the number of times that the loop is executed by changing the value that is stored in ecx (section 6, line2). For a more advanced challenge, try altering the decision examples so that the value from two variables is subtracted, rather than the immediate values in the code (section 5). Finally, you could see what happens if the 'jz' instruction is replaced with a 'jnz' instruction.

One last thing to think about. When stepping through the program, I asked you to make sure the registers were visible. In fact, we only looked at a small selection of the registers: the main data registers and flags. Other registers could be selected in the register window by right-clicking. Here is a brief description of some of them. Depending on the version of Pentium on your PC you may seem a slightly different selection:

Segment: These enable segmented virtual memory systems. They were used in IBM's OS-2 operating system, but are not really used in Windows or Linux, which both used paged virtual memory systems.

Floating point: These registers can be used for floating point operations.

MMX: 'MultiMedia eXtension' registers. This is actually another name for some of the floating point registers that can be used for multimedia instructions. They are large registers and can hold, for instance, several pixel values in one register.

SSE, SSE-2 etc. 'Streaming SIMD Extensions'. Here SIMD stands for: Single Instruction, Multiple Data. Similar to MMX, the idea is that a register can hold several values (pixels, sound samples…). The same instruction can then be applied to all the values. Very useful for image processing, sound processing and other multimedia applications.

EM64T: This is the Extended 64 bit register set that was first introduced by AMD and is now found on Intel processors. Registers that we have seen, such as EAX, are extended to 64 bits: RAX.

The Assembly Language Program

```
// PentiumProg.cpp

int main(int argc, char* argv[])
{
    int Var1 = 11, Var2 = 18, Result = 0;

_asm
{
        mov eax, 0x0       //1: registers
        mov al,  0x11
        mov ah,  0x22
        mov ax,  0x3333
        mov eax, 0x1234abcd
        nop
        nop
        mov eax, 0x90      //2: arithmetic
        mov ebx, 0x10100088
        add eax, ebx
        mov ecx, 0x20000000
        sub ecx, eax
        nop
        nop
        mov eax, Var1      //3: variables
        mov ebx, Var2
        sub eax, ebx
        mov Result, eax
        nop
        nop
        mov eax, 0x90000000    //4: flags
        mov ebx, 0xa0000000
        add eax, ebx
        sub eax, 0x30000000
        nop
        nop
        mov al, 0x11       //5a: decisions
        mov bl, 0x11
        sub al, bl
        jz _EQ
        mov cl, 0x00
        nop
_EQ:    mov cl, 0xff
        nop
        mov al, 0x11       //5b: decisions
        mov bl, 0x22
        sub al, bl
        jz _EQ
        mov cl, 0xaa
        nop
        nop
        mov eax, 0x0       //6: looping
        mov ecx,  0x04
_AA:    add eax, 0x03
        loop _AA

}
    return 0;
}
```