CSN08101, TUTORIAL 3: PENTIUM ASSEMBLY LANGUAGE

INTRODUCTION

This tutorial complements the practical lab exercise on the same subject area. It is intended to introduce assembly language programming at a level that will help you understand the way in which a processor works. The tutorial is divided into two sections. Due to the vagaries of lab scheduling, I do not know if you will have done the lab by the time you do this tutorial. So, section A is introductory, and repeats much of the lab material. If you have not yet done the lab, you will need to work through this section A. If you have done the lab, and are happy that you understand that, then you can skip to section B.

SECTION A: ELEMENTS OF ASSEMBLY LANGUAGE

Assembly Language Instructions

These are written in short form using 'mnemonics': abbreviations that are supposed to remind you of the instruction's purpose. Some (MOV, MUL, ADD) are fairly obvious in meaning. Some are less obvious, but are usually an abbreviation for something (e.g. JGE means **J**ump if **G**reater than or **E**qual to). They vary for different processor families, and I am going to refer to the instructions for the Pentium family (essentially these are the same for every member of the family from the 80386 through all '486, P1,,P4).

Registers

These are used to store data values that are currently being worked on. The Pentium family has a particularly complicated register structure due mostly to inherited 80x86 features. For instance, the main accumulator can be referred to as AL (the bottom 8 bits), AX (the bottom 16) or EAX, (all 32). The full EAX register can hold 32 bits, usually written as 8 hex digits (preceded by 0x to denote hexadecimal). What would the contents of the Al register be after the following?

```
MOV   AL,0X80
ADD   AL,0X1F
```

What would the contents of the EAX register be after the following?

```
MOV   EAX,0X1234567
MOV   BL,0X10
ADD   AL,BL
```

Source & Destination

An instruction, such as ADD AL,BL takes numbers from two source registers, manipulates them, then stores the result in the destination register. The Pentium always has the destination as the first register after the instruction (e.g. MOV dest, src). Note that the original contents of the first register are lost. Other processors may list the destination second. A recent trend (e.g. Itanium IA-64, Power PC) is to specify three registers: 2 source, and one destination. This leaves the contents of the source register unchanged. ADD AX,0X1234 is a valid instruction, so what would be wrong with this instruction?

```
ADD 0X1234,AX
```

Addressing Modes

For the Pentium family, data moves always involve at least one register. However, the source or destination of the data could be from different places , accessed in a variety of ways called addressing modes. You have already seen examples of addressing modes. In an instruction such as MOV AX, 0XFFFF, the data that is to be moved into the register is contained in the instruction itself (this is known as immediate addressing). It can only be used for data sources. A second mode of addressing is to use register-to-register addressing: MOV AL,BL. The simplest way of getting data from variables in RAM is to give their address in square brackets: MOV AX,[0X9988] means go to location 9988, fetch the contents, and put them in AX. Subtle point: AX is the name of a 16 bit register, so this instruction fetches 16 bits by getting 8 bits from location 9988, then another 8 bits from 9989. Likewise, MOV [0X9988],AL would take the 8 bits from the AL register and store them in RAM at location [0X9988]. The addresses in RAM can be large (8 hex digits). I will often use simple examples where I have omitted leading zeroes. The previous address would actually come out of the processor as 0X00009988, or (in binary) 00000000000000001001100110001000. Data locations in RAM can also be given names; this is essentially what a variable is. So, a valid instruction could be: MOV EBX,total where 'total' is presumably some location in RAM.

What would be the effect of the instruction: MOV [0XABCDEF00],EAX

Conditional branches

A section of code can always jump to another bit using the JMP instruction (e.g. JMP End, where End is a label given to another bit of code). However, a processor has to be able to make decisions, and this is usually done using conditional branches or jumps. This is done in two stages. First something is done that will affect the flags (a collection of bits that are set or cleared under various conditions). Then a conditional branch is inserted. This tests the value of a particular flag, for instance:

```
CMP  EAX,EBX     ;Compare the contents of the two registers
JE  Same         ;Jump if Equal to the code labelled Same
  ….             ;Continue here if not the same
```

Note that if the contents of EAX and EBX were not the same, execution would continue with the instruction following the JE instruction. The Pentium has a number of conditional jumps, examples are: JE, JNE, JG, JLE (Jump if Equal, Jump if Not Equal, Jump if Greater than, Jump if Less than or Equal). There are several flags. A couple of important ones are the Zero flag (set to 1 if the result is zero), and the Carry flag (set to one if the result overflows the register, meaning that there is a bit to be carried). For instance:

```
SUB  EAX,0X10    ;Subtract 10 from EAX
JZ  Next         ;Jump to Next if the result is zero
```

Looping

On many processors, this is done by doing a conditional jump back in the code. The Pentium is unusual in having a specific instruction, LOOP, that uses the Count register (ECX) as a loop counter:

```
       MOV AL, 0X00     ;Clear AL
       MOV ECX, 0X08    ;Set the counter to 8
Here:  ADD AL, 0X02     ;Add 2 to AL
       LOOP Here        ;Loop back
```

Each time the LOOP instruction is encountered, the contents of ECX are decremented. If the result is not zero, the LOOP goes back to the label. Once ECX reaches zero, the LOOP is ignored, and execution continues on the following line. This is very similar to a 'for' loop in a High Level Language. What is the contents of AL after the above instructions are executed?

SECTION B

1. What will be the contents of the al and bl registers after the following code has executed:

```
mov  al,0x10
mov  bl,0x08
Sub  al,bl
```

A. al = 0x02; bl = 0x08;
B. al = 010x; bl = 0x02;
C. al = 0x10; bl = 0x08;
D. al = 0x08; bl = 0x10;
E. al = 0x08; bl = 0x08

2. What will be the contents of the al and bl registers, and the Zero Flag (ZR) after the following code has executed.

```
mov  al,0xff
mov  bl,0xff
sub  al,bl
```

A. al = 0x00; bl = 0xff; ZR = 1;
B. al = 0x00; bl = 0x00; ZR = 0;
C. al = 0xff; bl = 0x00; ZR = 0;
D. al = 0xff; bl = 0x00; ZR = 1;
E. al = 0x00; bl = 0xff; ZR = 0;

3. What will happen after the following code has executed.

```
mov  al,0xff
mov  bl,0xff
sub  al,bl
jz   Next_bit
```

A. bl and al will contain 0xff, the conditional jump will take place
B. The zero flag will be cleared to 0, the conditional jump will take place
C. The zero flag will be set to 1, the conditional jump will not take place
D. The zero flag will be cleared to 0, the conditional jump not will take place
E. al will contain 0x00, the conditional jump will occur

4. What will al contain after the following code has executed.

```
mov  al,0x10
mov  ecx,0x0003
```

```
Here:   add     al,0x05
        loop    Here
```

A. 0x15
B. 0x25
C. 0x0f
D. 0x1f
E. 0x10

5. What will the contents of ax be after the following code has executed:

```
mov ax,[2000]
mov bx,[1000]
add bx,ax
```

A. 2000
B. 3000
C. The same as the contents of memory location 2000
D. Twice the contents of memory location 2000

6. Explain what is happening in each of the following lines of code:

```
mov     ax,0xf1aa
mov     bx,0x2b03
sub     ax,bx
jz      finish
```

7. In the following lines of code, explain what size of numbers are being manipulated, and what happens at the conditional jump instruction.

```
Line1:  sub     rax,rbx
        jz Line1
Line2:  jmp last_section
```

SECTION C. ADVANCED

The next section shows some examples where I have taken the output of the C++ compiler to show the assembly language that is produced by the compiler. The lines of C++ code are shown in bold, the assembly language in ordinary font. This is here purely for interest. I have added comments: see if you can follow what is going on.

1. Initialising the variables at the start of the program

```
    signed int Var1 = 11, Var2 = 18, Result = 0;
00401009  mov    dword ptr [Var1],0Bh     /* Put value into 32 bit (dword) location pointed to by Var1*/
00401010  mov    dword ptr [Var2],12h     /* Note the instructions are quite long: 7 bytes */
00401017  mov    dword ptr [Result],0
```

2. Using a system call

```
   cookie ^= GetTickCount();                                 /* Call a System function  */
00401402  call   dword ptr [__imp__GetTickCount@0 (43A010h)]   /* Result will be returned in EAX*/
00401408  xor    eax,dword ptr [cookie]                        /* XOR the value with the value in cookie*/
0040140B  mov    dword ptr [cookie],eax                        /* Store the result in the variable cookie*/
```

3. building up the two outcomes of an if statement using jumps:

```
    if ( initret != 0 )
004014B7  cmp    dword ptr [initret],0               /* Compare the value in the variable with 0 */
004014BB  je     _cinit+52h (4014C2h)                /* If the variable was 0, skip to 4014C2h*/
        return initret;                              /* If the value was not 0, do the next bit*/
004014BD  mov    eax,dword ptr [initret]             /*Put the value into EAX so it will be returned*/
004014C0  jmp    _cinit+99h (401509h)                /*Jump to the end of the routine*/
004014C2  push   offset _RTC_Terminate (402B20h)     /*Code to do if we don't return yet*/
…………
00401509  ret                                        /*End of routine, return to caller*/
```