Inline Assembler

Willi-Hans Steeb and Yorick Hardy

International School for Scientific Computing
e-mail: steebwilli@gmail.com

Abstract

We provide a collection of inline assembler programs.

1 Using the EAX register

The `EAX` register is also known as the *accumulator*. The `EAX` register is always involved when we perform multiplication and division. It is also the most efficient register to use for some arithmetic, logical, and data-movement operations.

The lower 16 bits of the register can be referenced using the name `AX`. The lower 8 bits of the `AX` register are also known as the `AL` register (for A-Low), and the upper 8 bits of the `AX`-register are also known as the `AH` register (for A-High). This can be convenient for handling byte-sized data, since it allows `AX` to serve as two separate registers.

The `LAHF` transfers the low byte of the flags word to `AH`. The bits, from MSB to LSB, are sign, zero, indeterminate, auxiliary carry, indeterminate, parity, indeterminate, and carry.

```
// EAX.cpp

#include <iostream>
using namespace std;

int main()
{
   unsigned char r;

   _asm
   {
   XOR EAX, EAX
   XOR EBX, EBX
   XOR ECX, ECX
   MOV EBX, 3
   MOV ECX, -4
   ADD EBX, ECX
   LAHF
   MOV r, AH
   }
   cout << "r = " << (int) r << endl; // 134 in binary 10000110
   return 0;
}
```

2 Using the EBX register

The EBX *register* is a general purpose register so it can be used for logical and arithmetic operations. Furthermore, the EBX register can point to memory locations. A 32-bit value stored in EBX can be used as a part of the address of a memory location to be accessed. The lower 16 bits of the register can be referenced using the name BX. The lower 8 bits of the BX register are also known as the BL register (for *B*-Low), and the upper 8 bits of the BX-register are also known as the BH register (for *B*-High). This can be convenient for handling byte-sized data, since it allows BX to serve as two separate registers.

By default, when EBX is used as a memory pointer, it points relative to the DS segment register (DS = Data Segment).

It is square bracket [EBX} that indicates that the memory location pointed to by EBX, rather than EBX itself, should be the source operand.

```
// EBX.cpp
// pointers

#include <iostream>
using namespace std;

int main()
{
   unsigned int value = 81;
   unsigned int r;

   _asm
   {
   LEA EBX, value
   MOV EDX, [EBX]
   ADD EDX, [EBX]
   MOV r, EDX
   }
   cout << "r = " << r << endl; // 81+81 = 162
   return 0;
}
```

3 Using the ECX register

The `ECX` *register* is a general purpose register so it can be used for logical and arithmetic operations. It is also called the *counting register*. The `ECX` register's speciality is counting and looping. Do not use the `ECX` register for other purposes when loops are involved. Counting down and looping is a frequently used program element, so the Pentium provides a special instruction to make loops faster and more compact. Not surprisingly, that instruction is called `LOOP`. The `LOOP` instruction decrements the count register `ECX` without changing any of the flags. The condition is then checked whether `ECX` is not 0. Thus the `loop` instruction subtracts 1 from `ECX` and jumps if `ECX` is not 0. The loop-functions `loope`, `loopz`, `loopne`, `loopnz` impose certain conditions, for example `ZF = 1` (`ZF` = zero flag).

The lower 16 bits of the register can be referenced using the name `CX`. The lower 8 bits of the `CX` register are also known as the `CL` register (for *C*-Low), and the upper 8 bits of the `CX`-register are also known as the `CH` register (for *C*-High). This can be convenient for handling byte-sized data, since it allows `CX` to serve as two separate registers.

```
// ECX.cpp

#include <iostream>
using namespace std;

int main(void)
{
   unsigned int sum = 0;

   _asm
   {
   XOR EBX, EBX
   MOV ECX, 20
   }
   L:
   _asm
   {
   ADD EBX, 5
   LOOP L
   MOV sum, EBX
   }
   cout << "sum = " << sum << endl; // => 100
```

```
   return 0;
}
```

4 Using the EDX register

The EDX *register* is a general purpose register, so it can be used for logical and arithmetic operations. The EDX register is the only register that can be used as an I/O address pointer with the IN and OUT instructions. IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. The other unique properties of EDX relate to division and multiplication.

The function MUL performs unsigned multiplication. A doubleword operand is multiplied by EAX and the result is left in the register pair

```
EDX:EAX
```

EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise they are set to 1.

The function DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand.

The function CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX.

```
// EDX.cpp

#include <iostream>
using namespace std;

int main(void)
{
   unsigned long value = 171;
   unsigned long r1;
   unsigned long r2;

   _asm
```

```
   {
   MOV EAX, value
   XOR EBX, EBX
   XOR EDX, EDX
   MOV EBX, 7
   DIV EBX
   MOV r1, EAX
   MOV r2, EDX
   }
   cout << "r1 = " << r1 << endl; // => 24
   cout << "r2 = " << r2 << endl; // => 3 (remainder)
   return 0;
}
```

5 Using the EDI register

The `ESI` *register* is also the called source-index register. Like the `EBX` register, the `ESI` register can be used as a memory pointer. This can be very effective when accessing a sequential series of memory locations, such as a text string. Better still, the string instructions can be made to automatically repeat their actions any number of times, so a single instruction can perform hundreds or even thousands of actions. The lower 16 bits of the register can be referenced using the name `SI`.

The function `LOADS` loads the `AL`, `AX`, or `EAX` register with the memory byte, word, or doubleword at the location pointed to by the source-index register `ESI`. The functions are `LODSB`, `LODSW` and `LODSD`.

The `EDI` *register* is also called the extended destination register. The `EDI` register is much like the `ESI` register in that it can be used as a memory pointer and has special properties when used with the powerful string instructions. The lower 16 bits of the register can be referenced using the name `DI`.

The instruction `MOVS` copies the byte or word at `[(E)SI]` to the byte or word at `ES:[(E)DI]`. This means, `MOVSB` copies the byte at `DS:[ESI]` to `ES:[EDI]`, `MOVSD` copies the word at `DS:[ESI]` to `ES:[EDI]`, `MOVSD` copies the doubleword at `DS:[ESI]` to `ES:[EDI]`. After the data are moved, both `(E)SI` and `(E)DI` are advanced automatically. `MOVSB`, `MOVSW`, and `MOVSD` are synonyms for the byte, word, and doubleword `MOVS` instructions. If the direction flag is 0 (`CLD` was executed),

the registers are incremented; if the direction flag is 1 (STD was executed), the registers are decremented. Bit 10 of the status register dictates the direction (forward or backward) in which the Pentium processes a string. The setting of this flag causes the index register to be incremented or decremented by one. The ES (extra segment) register points to an extra segment of memory if set up and used in the program. Many string instructions use the DS and ES register to mark the source and destination of string moves.

```
// EDI.cpp

#include <iostream>
using namespace std;

int main()
{
  unsigned int* array = NULL;
  array = new unsigned int[4];
  array[0] = 45; array[1] = 56;
  array[2] = 76; array[3] = 17;
  cout << "array = " << array << endl;    // 00322C10
  cout << "&array = " << &array << endl;  // 0012FF60

  unsigned int* p;
  unsigned int* q;
  unsigned int sum = 0;

  _asm
  {
  LEA EDI, array
  MOV EBX, [EDI]
  MOV q, EDI
  MOV p, EBX

  MOV ECX, [EBX]
  ADD ECX, [EBX+4]
  ADD ECX, [EBX+8]
  ADD ECX, [EBX+12]
  MOV sum, ECX
  }
  cout << "p = " << p << endl;     // 00322C10
  cout << "q = " << q << endl;     // 0012FF60
```

```
  cout << "sum = " << sum << endl; // 194
  delete[] array;
  return 0;
}
```

The ASCII table is utilized in the next program.

```
// charadd.cpp

#include <iostream>
using namespace std;

int main()
{
  char str[5] = { 's', 'u', 's', 'i', '\0' };

  unsigned int sum;

  _asm
  {
  LEA ECX, str
  XOR EDI, EDI

  MOVSX EAX, BYTE PTR [ECX]
  MOVSX EDX, BYTE PTR [ECX+1]
  ADD EAX, EDX
  MOVSX EDX, BYTE PTR [ECX+2]
  ADD EAX, EDX
  MOVSX EDX, BYTE PTR [ECX+3]
  ADD EAX, EDX
  MOV EDX, EAX
  MOV sum, EDX
  }
  cout << "sum = " << sum; // 115+117+115+105=452
  return 0;
}
```

The command `CLD` clears the direction flag. No other flags are affected. After `CLD` is executed, string operations will increment the index registers (`SI` or `DI`) that they use. The command `REPNE` (repeat while not equal) is applied to string operations. The command `SCASB` (compare string data) subtracts the memory

byte at the destination register from the AL register. The result is discarded; only the flags are set.

```
// stos.cpp

#include <iostream>
using namespace std;

int main()
{
   char str[5] = { 's', 'u', 's', 'i', '\0' };

   _asm
   {
   LEA EDI, str
   CLD
   REPNE SCASB
   DEC EDI ; decrement pointer by 1
   MOV AL, 'o'
   STOS str
   }
   cout << str << endl; // ousi
   return 0;
}
```

6 Using the ESP register

The ESP *register*, also known as the *stack pointer*, is the least general of the general-purpose registers, for it is almost always dedicated to a specific purpose: maintaining the stack. The *stack* is an area of memory into which values can be stored and from which they can be retrieved on a last-in, first-out basis; that is, the last value stored onto the stack is the first value we will get when we read a value from the stack. The classic analogy for the stack is that of a stack of dishes. Since we can only add plates at the top of the stack and remove them from the top of the stack, it stands to reason that the first plate we put on the stack will be the last plate we can remove.

The ESP register points to the top of the stack at any given time; as with the stack of dishes, the top of the stack is the location at which the next value placed on the stack will be stored. The action of placing a value on the stack is known as

pushing a value on the stack, and, indeed, the `PUSH` instruction is used to place values on the stack. Thus execution of push causes the following to happen: a copy a the source content is moved to the address specified by `SS:ESP`. The source is unchanged. `ESP` is decreased by 4 Similarly, the action of retrieving a value from the stack is known as *popping* a value from the stack, and the `POP` instruction is used to retrieve values from the stack. Thus execution of `POP` causes this to happen: the content of `SS:ESP` (the top of the stack) is moved to the destination. `ESP` is increased by 4

Stack operations are supported by three registers:

Stack Segment (`SS`) Register. Stacks reside in memory. The number of stacks in a system is limited only by the maximum number of segments. A stack can be up to 4 gigabytes long, the maximum size of a segment. One stack is available at a time – the stack whose segment selector is held in the `SS` register. This is the current stack, often referred to simply as "the" stack. The `SS` register is used automatically by the processor for all stack operations.

Stack Pointer (`ESP`) Register. The `ESP` register holds an offset to the top-of-stack (TOS) in the current stack segment. It is used by `PUSH` and `POP` operations, subroutine calls and returns, exceptions, and interrupts. When an item is pushed onto the stack, the processor decrements the `ESP` register, then writes the item at the new TOS. When an item is popped off the stack, the processor copies it from the TOS, then increments the `ESP` register. In other words, the stack grows down in memory toward lesser addresses.

Stack-Frame Base Pointer (`EBP`) Register. The `EBP` register typically is used to access data structures passed on the stack. For example, on entering a subroutine the stack contains the return address and some number of data structures passed to the subroutine. The subroutine adds to the stack whenever it needs to create space for temporary local variables. As a result, the stack pointer gets incremented and decremented as temporary variables are pushed and popped. If the stack pointer is copied into the base pointer before anything is pushed on the stack, the base pointer can be used to reference data structures with fixed offsets. If this is not done, the offset to access a particular data structure would change whenever a temporary variable is allocated or de-allocated. When the `EBP` register is used to address memory, the current stack segment is referenced (i.e., the `SS`

segment). Because the stack segment does not have to be specified, instruction encoding is more compact. The EBP register also can be used to address the segments. Instructions, such as the ENTER and LEAVE instructions, are provided which automatically set up the EBP register for convenient access to variables.

ENTER creates the stack frame required by most block-structured high-level language, for example ENTER 12, 0. The first operand specifies the number of bytes of dynamic storage allocated on the stack for the routines being entered. The second operand gives the lexical nesting level (0 to 31) of the routine within the high-level language source code. LEAVE reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer (set ESP to EBP), LEAVE releases the stack space used by a procedure for its local variables.

```
// ESP.cpp

#include <iostream>
using namespace std;

int main()
{
   int a = 3;
   int b = 4;
   int c = 5;

   _asm
   {
   MOV EAX, a
   MOV EBX, b
   MOV ECX, c
   PUSH EAX
   PUSH EBX
   PUSH ECX
   POP EAX
   POP EBX
   POP ECX
   MOV a, EAX
   MOV b, EBX
   MOV c, ECX
   }
   cout << "a = " << a << endl; // => 5
   cout << "b = " << b << endl; // => 4
```

```
   cout << "c = " << c << endl; // => 3

   int x = 6;
   int y = 7;
   int z = 8;

   _asm
   {
   MOV EAX, x
   MOV EBX, y
   MOV ECX, z
   SUB ESP, 12 ; 3*4 bytes = 12 bytes
   MOV [ESP+8], EAX
   MOV [ESP+4], EBX
   MOV [ESP], ECX
   POP EAX
   POP EBX
   POP ECX
   MOV x, EAX
   MOV y, EBX
   MOV z, ECX
   }
   cout << "x = " << x << endl; // => 8
   cout << "y = " << y << endl; // => 7
   cout << "z = " << z << endl; // => 6
   return 0;
}

// inc.cpp

#include <iostream>
using namespace std;

int main()
{
  unsigned int j = 1;

  _asm
  {
  SUB ESP, 8
  MOV DWORD PTR [ESP+4], 5
  MOV DWORD PTR [ESP], 6
  INC DWORD PTR [ESP+4]
```

```
  INC DWORD PTR [ESP]
  MOV EDX, [ESP]
  ADD EDX, [ESP+4]
  MOV j, EDX
  }
  cout << "j = " << j << endl; // => 6+7=13
  return 0;
}
```

7 Conditional and Unconditional Jumps

The command `JMP` transfers control to a different point in the instruction stream without recording return information.

Command `JGE`: Jump if greater or equal.

Command `JBE`: Jump if below or equal.

Command `CMP`: Compare two operands. `CMP` subtracts the second operand from the first but does not store the result; only the flags are changed. `CMP` is used in conjunction with conditional jumps.

```
// sroot.cpp

#include <iostream>
using namespace std;

unsigned int isqrt(unsigned int square)
{
  unsigned int r;

  _asm
  {
  MOV ECX, square
  MOV EBX, ECX
  MOV EAX, 1
  JMP SHORT zero_chk
  }
  loop1:
  _asm
  {
  SUB ECX, EAX
```

```
  ADD EAX, 2
  }
  zero_chk:
  _asm
  {
  OR ECX, ECX
  JGE loop1
  SAR EAX, 1
  MOV ECX, EAX
  IMUL ECX
  SUB EAX, ECX
  INC EAX
  CMP EAX, EBX
  JBE finish
  DEC ECX
  }
  finish:
  _asm
  {
  MOV EAX, ECX
  MOV r, EAX
  }
  return r;
}

int main()
{
  unsigned int j;
  j = isqrt(360001);
  cout << "j = " << j << endl;
  return 0;
}

// whileif.cpp

#include <iostream>
using namespace std;

int main()
{
  unsigned long r = 65;
  unsigned long m = 16;
```

```
  while(r >= m) { r = r - m; }
  cout << "r = " << r << endl;  // => 1

  unsigned long s = 65;
  if(s >= m) { s = s - m; }
  cout << "s = " << s << endl;  // => 49

  return 0;
}
```

The `while` and `if` can be implemented in embedded assembler as follows:

```
// whileif1.cpp

// JA jump if above
// JA causes program execution to branch to the
// operand address if both the carry and zero flags are clear.
// JBE jump if below or equal
// JBE causes program execution to branch to the
// operand address if either the carry or zero flag is set.

#include <iostream>
using namespace std;

int main()
{
   unsigned long r = 65;
   unsigned long m = 16;

   _asm
   {
   MOV ECX, m
   MOV EDX, r
   CMP EDX, ECX
   JBE L1
   }
   L2:
   _asm
   {
   SUB EDX, ECX
   MOV r, EDX
   CMP EDX, ECX
   JA L2
```

```
   }
   L1:

   cout << "r = " << r << endl; // => 1

   unsigned long s = 65;
   _asm
   {
   MOV ECX, m
   MOV EDX, s
   CMP EDX, ECX
   JBE L3
   SUB EDX, ECX
   MOV s, EDX
   }
   L3:

   cout << "s = " << s << endl; // => 49
   return 0;
}
```

8 Using XOR operation

`XOR` computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

```
// swap.cpp

#include <iostream>
using namespace std;

int main()
{
   int x = 8;
   int y = -17;

   _asm
   {
   MOV EBX, x
   MOV ECX, y
   XOR EBX, ECX
   XOR ECX, EBX
```

```
   XOR EBX, ECX
   MOV x, EBX
   MOV y, ECX
   }
   cout << "x = " << x << endl; // -17
   cout << "y = " << y << endl; // 8
   return 0;
}
```

9 Pointers

Pointer. A value that indicates the storage location (address) of a item of data. Thus a pointer has an address and contains an address.

```
// pointer1.cpp

#include <iostream>
using namespace std;

int main(void)
{
   unsigned int a = 14;
   unsigned int* p = NULL;

   p = &a;  // address of a is assigned to p
   *p = 23; // * is the dereference operator
            // thus a is assigend to 23, i.e. 14 is overridden

   cout << "a = " << a << endl;
   return 0;
}
```

With inline assembler language the program would look as follows

```
// pointer1a.cpp

#include <iostream>
using namespace std;

int main(void)
{
   unsigned int a = 14;
   unsigned int* p = NULL;
```

```
   _asm
   {
   LEA EBX, a    ; p = &a
   MOV p, EBX

   MOV EBX, p    ; *p = 23
   MOV [EBX], 23
   }
   cout << "a = " << a << endl;
   return 0;
}
```

Pointers to pointers are applied as follows

```
// pointer2.cpp

#include <iostream>
using namespace std;

int main(void)
{
   int v = 9;
   int* p = NULL;   // p is a pointer to int
   p = &v;          // address of v is assigned to p
   int** pp = NULL; // pp is a pointer to pointer
   pp = &p;         // address of p is assigned to pp
   **pp = -11;      // dereference pp twice
   cout << "v = " << v << endl; // => -11
   return 0;
}
```

With inline assembler language the program would look as follows

```
// pointer2a.cpp

#include <iostream>
using namespace std;

int main(void)
{
   int v = 9;
```

```
    _asm
    {
     PUSH EDI                   ; cannot use EBP, since
                                ; it already describes the stack frame of main()
                                ; which is used to find v
     PUSH EBX
     MOV  EDI, ESP

     SUB  ESP, 4                ; int *p
                                ; p == [EDI]

     LEA  EBX, v                ; p = &v
     MOV  [EDI], EBX            ;

     SUB  ESP, 4                ; int **PP = 0 (NULL)
                                ; pp == [EDI-4]

     LEA  EBX, [EDI]            ; pp = &p
     MOV  [EDI-4], EBX          ;

     MOV  EBX, [EDI-4]          ; **pp = -11
     MOV  EBX, [EBX]            ;
     MOV  DWORD PTR [EBX], -11 ;

     ADD  ESP, 8                ; free pp, free p
     POP  EBX                   ; restore EBX
     POP  EDI                   ; restore EDI
    }
    cout << "v = " << v << endl; // => -11
    return 0;
}
```

10 Floating Point Instructions

Floating point numbers can be stored as `float` (32 bits) or `double` (64 bits). `ST` is the top of the stack; the register currently at the top of the stack. `ST(1)` is a register in the stack $i\,(0 \leq j \leq 7)$ stack elements from the top. `ST(1)` is the next-on-stack register, `ST(2)` is below `ST(1)`, etc.

The command `FLD` stands for load real. The command `FADD` stands for add real with

```
FADD destination, source
```

The command `FSTP` stands for store real and pop.

```
// stack.cpp

#include <iostream>
using namespace std;

int main()
{
   float a=4.0;
   float b=5.0;
   float c=6.0;
   float d=7.0;
   float result;

   _asm
   {
   FLD a
   FLD b
   FADD
   FLD c
   FADD
   FLD d
   FADD
   FSTP result
   }
   cout << "result = " << result; // => 22.0
   return 0;
}
```

The command `FLDPI` stands for load pi. The command `FMUL` stands for multiply real.

```
// fpi.cpp

#include <iostream>
using namespace std;

double pimul(double x,double y)
{
   _asm
   {
   FLDPI
   FMUL x
   FMUL y
   FSTP y
   }
   return y;
}

int main()
{
  double a = 2.0;
  double b = 4.5;
  double result = pimul(a,b); // 4.5 x 2.0 x 3.14159
  cout << "result = " << result << endl; // 28.2743
  return 0;
}
```

The command `FCOMP` stands for compare real and pop. The command `FSTSW` stands for store status word. This instruction is commonly used for conditional branching. The command `SAHF` stands for store register `AH` into flags (opposite of `LAHF`).

```
// fcomp2.cpp

#include <iostream>
using namespace std;

int fcomp(float x,float y)
{
   int r;

   _asm
   {
   FLD x
```

```
   FCOMP y
   FSTSW AX
   SAHF
   JE short L1
   XOR EAX, EAX
   JMP L2
   }
   L1:
   _asm
   {
   MOV EAX, 1
   }
   L2:
   _asm
   {
   MOV r, EAX
   }
   return r;
}

int main()
{
   float a = 3.3;
   float b = 3.2;
   int result = fcomp(a,b);
   cout << "result = " << result << endl; // => 0
   return 0;
}
```

11 Inline assembler under the GNU compiler g++

The following small example shows how inline assembler under the GNU compiler is used (AT & T style). Register names are prefixed with %. Source/destination ordering: The source is always on the left and the destination is always on the right. Compile and link with

```
g++ -fasm -o gnuasm gnuasm.cpp
```

```
// gnuasm.cpp

#include <iostream>
using namespace std;

int main(void)
{
  int v = 11, s;
  asm ("movl %1, %%eax\n"
       "movl %%eax, %0\n"
       "incl %0\n"
       "negl %0\n"
       : "=r"(s)  /* s is output operand */
       : "r"(v)   /* v is input operand */
       : "%eax"); /* %eax is clobbered register */
  cout << "s = " << s << endl; // => -12

  int w = 7, z = 15;
  asm ("addl %%ecx, %%eax\n"
       : "=a"(w)
       : "a"(w), "c"(z)
       );
  cout << "w = " << w << endl; // => 22
  cout << "z = " << z << endl; // => 15

  int count = 0;
  asm ("movl %%ecx, %%ecx\n"
       "incl %%ecx\n"
      : "=c"(count)
      : "c"(count)
      );
  cout << "count = " << count << endl;
```

```
  return 0;
}
```