Writing an 80386 C Compiler for the SAS® System

Glenn Musial, SAS Institute Inc., Cary, NC

INTRODUCTION

This paper discusses the development of a C compiler for the Intel 80386 microprocessor. A brief discussion of the evolution of the 80386 is given, followed by a description of the 80386 register and instruction sets. From there, the C compiler's view of the 80386 is presented with emphasis on the features of the chip that make it attractive to a large-scale application such as the SAS system. In addition, comments on the future of the 80386, including operating system targets and processor evolution, are presented.

EVOLUTION OF THE 80386

The first member of the 80x86 family to gain wide acceptance was the 8088/8086. Presented as a 16-bit processor, the 8086 used a segmented memory architecture to maintain source level consistency with its 8-bit forerunner, the 8080A (for example, every 8080A instruction could be translated into an 8086 instruction). While this compatibility eased the transition from the 8-bit to the 16-bit PC, the segmentation concept proved to be a fundamental and limiting flaw in both the 8086 and its successor, the 80286.

Some major drawbacks of the 8086 chip were addressed in the 80286. While the 80286, a programmatic superset of the 8086, could be run in 8086 compatibility mode, it also introduced a privileged mode of operation that provided for memory protection and process control management. By using a virtual-to-physical memory mapping scheme, the 80286 running in privileged (or *protected*) mode could exceed the 1 megabyte memory limit of the 8086 and directly address up to 16 megabytes of memory.

However, the software overhead required by the segmented architecture of the 80286 proved to be a significant obstacle. The cost of tracking, manipulating, storing and loading segment registers was too high to perform a truly quality implementation of large-scale software systems (including operating systems).

This leads us to the 80386. Although Intel has once again attempted to maintain a level of compatibility with it predecessors, the power of the 80386 lies in its *incompatibilities*. In native 32-bit unsegmented *flat model* mode, the 80386 is a 32-bit processor capable of efficiently performing operations on 8-bit, 16-bit, and 32-bit quantities. It is also capable of efficiently addressing large areas of memory (up to 4 gigabytes). In addition, new addressing modes allow for increased flexibility in memory access.

It must be pointed out that the 80386 is capable of running in a 48-bit segmented model, allowing the direct access of up to 64 terabytes of virtual memory. This paper shall focus on an efficient implementation in the 32-bit flat model mode (*no* segment registers are used); the only currently viable 48-bit 80386 implementations are in the embedded systems area.

REGISTER AND INSTRUCTION SET

Consider the register set of the 80386 (see Figure 1). The shaded regions represent a direct mapping of the 80286 16-bit register set. It is worth noting that the high order 16 bits of the 32-bit registers are **not** available as distinct 16-bit entities.

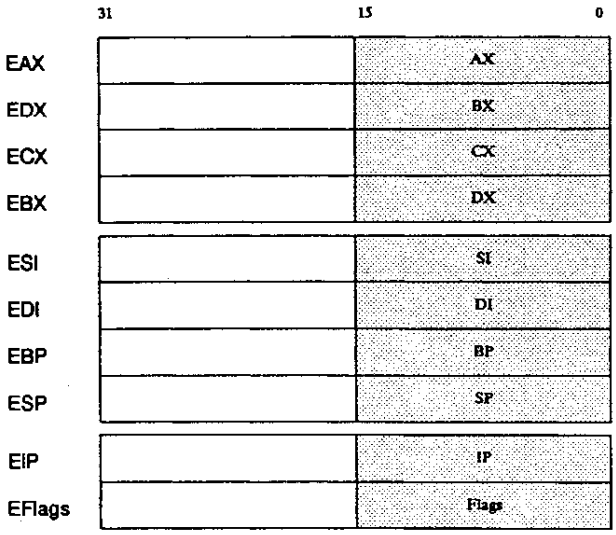

Figure 1: 80386 Flat Model Register Set

The instruction set for the 80386 will be very familiar to developers used to working with the 80286. Opcodes from the 80286 translate directly into 80386 opcodes. The 80286 8-bit manipulation instructions remain functionally unchanged and 16-bit manipulation instructions map to 32-bit manipulation instructions. The 80386 provides two new opcode prefix bytes, the operand-size prefix byte and the address-size prefix byte, to allow for the transformation of 32-bit instructions back to their 16-bit counterparts.

Instructions without immediate values use the same opcodes to map from 16 bit sequences to 32 bit sequences (8-bit instructions remain the same). Figure 2 demonstrates this, along with an example of a 16 bit instruction generated with an operand-size prefix byte.

OPCODES	80286	80386
8A C2	MOV AL,DL	MOV AL, DL
8B C2	MOV AX,DX	MOV EAX,EDX
66 8B C2		MOV AX, DX
AA	STOSB (AL->[DI])	STOSB (AL->[EDI])
67 AA		STOSB (AL->[DI])

Figure 2: 80386 Instructions Without Immediate Values

instructions which make use of immediate values use similar op-code sequences with 16-bit quantities changed to 32-bit quantities. Once again, 8-bit instructions remain the same (see Figure 3).

OPCODES	80286	80386
B0 01	MOV AL, 1	MOV AL, 1
B8 00 01	MOV AX, 1	
66 B8 00 01		MOV AX, 1
B8 00 00 00 01		MOV EAX, 1
8A 46 12	MOV AL,[SI+12]	MOV AL,[ESI+12]
8A 86 34 12	MOV AL,[SI+1234]	
8A 8634 12 00 00		MOV AL,[ESI+1234]
67 8A 86 34 12		MOV AL,[SI+1234]

Figure 3: 80386 Instructions With Immediate Values

An important improvement of the 80386 is in the area of memory addressing. All general purpose registers can now be used as index registers. Also, a broad, new range of addressing modes has been introduced. These changes are summarized in Figure 4.

[REG1 [+REG2[*SCALE]] [+OFFSET]]	
REG1	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
REG2 (optional)	EAX, EBX, ECX, EDX, ESI, EDI, EBP
SCALE (optional)	1, 2, 4, 8
OFFSET (optional)	8 or 32 bit quantity

Figure 4: 80386 Addressing Modes

Another significant improvement in the 80386 is addition of several new instructions which are particularly useful for compiler writers. These are summarized in Figure 5.

MNEMONIC	DESCRIPTION
MOVSX	Move of 8 and 16-bit quantities with sign extend
MOVZX	Move of 8 and 16-bit quantities with zero extend
SETcc	Byte set on condition
Jcc	Jump long on condition
SHxD	Double precision shift left or right (used in support routines)
BSR	Bit Scan reverse (used in support routines)
BT(x)	Bit test (and modify) (used in support routines)

Figure 5: New 80386 Instructions

THE C COMPILER'S PERSPECTIVE

This section describes ways in which an 80386 compiler can take advantage of the power of the processor. Remember when reviewing the examples that 80386 instructions are typically longer than their 80286 counterparts due to 16 vs. 32 bit offset differences. The true efficiency of generated code must be measured by speed as well as size. In each of the following examples, a significant speed increase is achieved in generated 80386 code.

All 80286 examples are compiled using 32-bit pointer model, to allow for a better functional comparison.

32-bit Linear Address Space

The greatest benefit of the 80386 is that it allows the compiler to view memory as a 32-bit linear address space. Without the overhead of segmentation, the compiler can generate faster and smaller code in the areas of pointer arithmetic and memory access. Example 1 demonstrates the improvements in generated code that can be achieved in a linear address space.

Also, the (somewhat artificial) limit of 64K of default static data is removed because of the linear nature of the address space.

In general, it is no longer neccessary for the application programmer to assist the compiler in the code generation task by using extended keywords such as *huge, near* and *far*.

80286

```
for ( i = 0; i < j; i++ )
0000    C7 06 04 00-02 00 00   MOV   WORD PTR [i],0000
0005    EB 2F                  JMP   0037
  *p++ = getp();
0008    C4 3E 00 00-02         LES   DI,[p]
000C    8C 46 04               MOV   [BP+04],ES        ; pointer
000F    A1 02 00-02            MOV   AX,[p_hi]         ;  arithmetic
0012    8B 1E 00 00-02         MOV   BX,[p_lo]         ; *
0016    B9 01 00               MOV   CX,0001           ; *
0019    9A 00 00 00 00-XX      CALL  (FAR) _CXIX1      ; *
001E    A3 02 00-02            MOV   [p_hi],AX         ; *
0021    89 1E 00 00-02         MOV   [p_lo],BX         ; *
0025    89 7E 02               MOV   [BP+02],DI
0028    9A 00 00 00 00-XX      CALL  (FAR) getp
002D    C4 7E 02               LES   DI,[BP+02]        ; memory
0030    26 88 05               MOV   ES:[DI],AL        ;  access
0033    FF 06 04 00-02         INC   WORD PTR [i]
0037    A1 04 00-02            MOV   AX,[i]
0038    3B 06 06 00-02         CMP   AX,[j]
003E    7C C8                  JL    0008
```

80386

```
  for ( i = 0; i < j; i++ )
00000000  C7 05 00 00 00 00-02   MOV   DWORD PTR [i],00000000
          00 00 00 00
0000000A  EB 19                  JMP   00000025
    *p++ = getp();
0000000C  8B 3D 08 00 00 00-02   MOV   EDI,[p]
00000012  FF 05 08 00 00 00-02   INC   DWORD PTR [p]   ; ptr arith
00000018  E8 00 00 00 00-XX.1    CALL  getp
0000001D  88 07                  MOV   [EDI],AL        ; mem access
0000001F  FF 05 00 00 00 00-02   INC   DWORD PTR [i]
00000025  A1 00 00 00 00-02      MOV   EAX,[i]
0000002A  3B 05 04 00 00 00-02   CMP   EAX,[j]
00000030  7C DA                  JL    0000000C
```

Example 1: Efficiencies of a Linear Address Space

32-bit default ints

Array indices are processed most effectively if default ints are the same size as memory pointers. Thus, the natural size for default ints on the 80386 is 32 bits. Note that the 80386 processes 32-bit quantities more efficiently than 16-bit quantities, so there is no performance penalty for this decision.

Another benefit of the 32-bit nature of the processor is that there is less need for out-of-line code. Because the 80286 could not efficiently perform operations on 32-bit quantities, frequent calls to

out-of-line support routines were necessary, as evidenced in Example 2.

A further consequence is that all integer and pointer return values can be returned in the single register EAX, freeing EBX for register variable use as outlined below.

80286

```
  l3 = l1 * l2;
0000    8B 0E 06 00-02      MOV   CX,[l2_hi]        ; load
0004    8B 16 04 00-02      MOV   DX,[l2_lo]        ; and
0008    A1 02 00-02         MOV   AX,[l1_hi]        ;   multiply
000B    8B 1E 00 00-02      MOV   BX,[l1_lo]        ; *
000F    9A 00 00 00 00-XX   CALL  (FAR) CXM33       ; *
0014    A3 0A 00-02         MOV   [l3_hi],AX
0017    89 1E 08 00-02      MOV   [l3_lo],BX
```

80386

```
  l3 = l1 * l2;
00000000  A1 00 00 00 00-02   MOV   EAX,[l1]          ; load and
00000005  F7 2D 04 00 00 00-02 IMUL  DWORD PTR [l2]   ;   multiply
0000000B  A3 08 00 00 00-02   MOV   [l3],EAX
```

Example 2: Processing of 32-bit Quantities

Index registers

The addition of EAX, ECX and EDX as index registers allows for better register allocation algorithms. More index registers allow for keeping values in registers for longer periods of time during memory intensive operations. Example 3 demonstrates the efficiency of an increased number of index registers.

80286

```
  foo( p1, p2, p3, p4, p5,
    *p1, *p2, *p3, *p4, *p5);
0000    C4 3E 10 00-02      LES   DI,[p5]
0004    26 FF 35            PUSH  ES:[DI]
0007    C4 1E 0C 00-02      LES   BX,[p4]
000B    26 FF 37            PUSH  ES:[BX]
000E    C4 36 08 00-02      LES   SI,[p3]
0012    26 FF 34            PUSH  ES:[SI]
0015    C4 3E 04 00-02      LES   DI,[p2]         ; out of regs!
0019    26 FF 35            PUSH  ES:[DI]
001C    C4 1E 00 00-02      LES   BX,[p1]
0020    26 FF 37            PUSH  ES:[BX]
0023    FF 36 12 00-02      PUSH  [p5_hi]         ; push from memory
0027    FF 36 10 00-02      PUSH  [p5_lo]         ; *
002B    FF 36 0E 00-02      PUSH  [p4_hi]         ; *
002F    FF 36 0C 00-02      PUSH  [p4_lo]         ; *
0033    FF 36 0A 00-02      PUSH  [p3_hi]         ; *
0037    56                  PUSH  SI
0038    FF 36 06 00-02      PUSH  [p2_hi]         ; *
003C    57                  PUSH  DI
003D    06                  PUSH  ES
003E    53                  PUSH  BX
003F    9A 00 00 00 00-XX   CALL  (FAR) foo
```

80386

```
  foo( p1, p2, p3, p4, p5,
    *p1, *p2, *p3, *p4, *p5);
00000000  A1 10 00 00 00-02   MOV   EAX,[p5]         ; new index reg
00000005  FF 30               PUSH  [EAX]
00000007  8B 0D 0C 00 00 00-02 MOV  ECX,[p4]         ; new index reg
0000000D  FF 31               PUSH  [ECX]
0000000F  8B 15 08 00 00 00-02 MOV  EDX,[p3]         ; new index reg
00000015  FF 32               PUSH  [EDX]
00000017  8B 3D 04 00 00 00-02 MOV  EDI,[p2]
0000001D  FF 37               PUSH  [EDI]
0000001F  8B 1D 00 00 00 00-02 MOV  EBX,[p1]
00000025  FF 33               PUSH  [EBX]
00000027  50                  PUSH  EAX             ; push from reg
00000028  51                  PUSH  ECX             ; push from reg
00000029  52                  PUSH  EDX             ; push from reg
0000002A  57                  PUSH  EDI
0000002B  53                  PUSH  EBX
0000002C  E8 00 00 00 00-XX.1 CALL  foo
```

Example 3: Using New Index Registers

In addition, the extra index registers allow the number of register variables to be increased from two to three. Previously, two register variables were the functional limit as one index register needed to be kept free at all times.

The natural choice for the third register variable is EBX (ESI and EDI are the first two, reflecting the choices of SI and DI for the 80286). Specialized instructions which make use of EAX, ECX and EDX discourage their use for register variables.

Example 4 shows why three register variables are not practical in the 80286 case and are of benefit in the 80386 case.

80286

```
  register int i,j,k;
  for ( i = 0; i < ilimit; i++)
0000    C7 46 0C 00 00      MOV   WORD PTR [i],0000
0005    EB 45               JMP   004C
    for ( j = 0; j < jlimit; j++)
0007    33 FF               XOR   DI,DI
0009    EB 38               JMP   0043
      for ( k = 0; k < klimit; k++ )
000B    33 F6               XOR   SI,SI
000D    EB 2D               JMP   003C
        *p++ = i+j+k;
000F    8B 46 0C            MOV   AX,[i]
0012    03 C7               ADD   AX,DI
0014    03 C6               ADD   AX,SI
0016    C4 1E 00 00-02      LES   BX,[p]
001A    26 89 07            MOV   ES:[BX],AX      ; BX used here
001D    A1 02 00-02         MOV   AX,[p_hi]       ;   so can't be
0020    8B 1E 00 00-02      MOV   BX,[p_lo]       ;   used as reg var
0024    B9 02 00            MOV   CX,0002
0027    8E C0               MOV   ES,AX
0029    03 D9               ADD   BX,CX
002B    73 07               JAE   0034
002D    2B D9               SUB   BX,CX
002F    9A 00 00 00 00-XX   CALL  (FAR) CXIX1
0034    A3 02 00-02         MOV   [p_hi],AX
0037    89 1E 00 00-02      MOV   [p_lo],BX
003B    46                  INC   SI
003C    3B 36 08 00-02      CMP   SI,[klimit]
0040    7C CD               JL    000F
0042    47                  INC   DI
0043    3B 3E 06 00-02      CMP   DI,[jlimit]
0047    7C C2               JL    000B
0049    FF 46 0C            INC   WORD PTR [i]
004C    8B 46 0C            MOV   AX,[i]
004F    3B 06 04 00-02      CMP   AX,[ilimit]
0053    7C B2               JL    0007
```

80386

```
  register int i,j,k;
  for ( i = 0; i < ilimit; i++)
00000000  33 FF               XOR   EDI,EDI
00000002  EB 30               JMP   00000034
    for ( j = 0; j < jlimit; j++)
00000004  33 F6               XOR   ESI,ESI
00000006  EB 23               JMP   0000002B
      for ( k = 0; k < klimit; k++ )
00000008  33 DB               XOR   EBX,EBX
0000000A  EB 16               JMP   00000022
        *p++ = i+j+k;
0000000C  8B C7               MOV   EAX,EDI
0000000E  03 C6               ADD   EAX,ESI
00000010  03 C3               ADD   EAX,EBX
00000012  8B 15 00 00 00 00-02 MOV  EDX,[p]
00000018  89 02               MOV   [EDX],EAX       ; EDX as index reg
0000001A  83 05 00 00 00 00-02 ADD  DWORD PTR [p],04
          04
00000021  43                  INC   EBX
00000022  3B 1D 0C 00 00 00-02 CMP  EBX,[klimit]
00000028  7C E2               JL    0000000C
0000002A  46                  INC   ESI
0000002B  3B 35 08 00 00 00-02 CMP  ESI,[jlimit]
00000031  7C D5               JL    00000008
00000033  47                  INC   EDI
00000034  3B 3D 04 00 00 00-02 CMP  EDI,[ilimit]
0000003A  7C C8               JL    00000004
```

Example 4: Three Register Variables

New Instructions

Another straightforward way to take advantage of the 80386 is to make use of the new instructions provided. Consider how the following new instructions are used:

MOVZX,MOVSX — provide signed and unsigned conversion of 8 bit quantities to 16 and 32 bit quantities and signed and unsigned conversion of 16 bit quantities to 32 bit quantities. An important distinction from the 80286 instructions CBW and CWD is their ability to convert in all registers and convert from memory.

80286

```
  unsigned char uc;
  signed char sc;
  foo(uc,sc);
0000    8A 46 FE          MOV   AL,[uc]
0003    30 E4             XOR   AH,AH          ; zero extend
0005    8A 4E FE          MOV   CL,[sc]
0008    91                XCHG  AX,CX          ; sign extend
0009    98                CBW                  ; *
000A    91                XCHG  AX,CX          ; *
000B    51                PUSH  CX
000C    50                PUSH  AX
000D    9A 00 00 00 00-XX CALL  (FAR) foo
```

80386

```
  unsigned char uc;
  signed char sc;
  foo(uc,sc);
00000000  0F B6 45 FF       MOVZX  EAX,BYTE PTR [uc]
00000004  0F BE 4D FE       MOVSX  ECX,BYTE PTR [sc]
00000008  51                PUSH   ECX
00000009  50                PUSH   EAX
0000000A  E8 00 00 00 00-XX.1 CALL  foo
```

Example 5: New MOVZX, MOVSX Instructions

Jcc — allows conditional jumps to be taken within the entire 32-bit address space, eliminating the need for the tiresome *jump around on !condition* construct.

80286

```
  if ( flag )
0000    83 7E 04 00       CMP   WORD PTR [flag],00
0004    75 03             JNZ   0009
0006    E9 86 00          JMPL  008F
```

80386

```
  if ( flag )
00000000  83 7D FC 00       CMP   DWORD PTR [flag],00
00000004  74 83             JZ    00000089
```

Example 6: New Long Jump On Condition Instructions

SETcc — these instructions allow registers or memory locations to be set or reset depending on a condition code.

80286

```
  i = j < k;
0000    8B 46 F8          MOV   AX,[k]
0003    3B 46 F4          CMP   AX,[j]
0006    BB 00 00          MOV   BX,0000
0009    7D 01             JGE   000C
000B    43                INC   BX
```

80386

```
  i = j < k;
00000000  8B 45 F8          MOV    EAX,[k]
00000003  3B 45 F4          CMP    EAX,[j]
00000006  0F 9C C1          SETL   CL
00000009  0F B6 C9          MOVZX  ECX,CL
```

Example 7: New Set On Condition Instructions

New Addressing Modes

Taking advantage of the new addressing modes of the 80386 is a challenging problem for the compiler designer. The compiler must be able to plan ahead, keeping strategic values in registers for their subsequent use in complex addressing instructions.

Example 8 is a simple demonstration of an 80386 addressing mode. Two registers are used to perform a memory access. The power of the 80386 is better shown by Example 9, where the new scaled index addressing mode is used. The scaled index addressing mode allows for the a scale factor of 1, 2, 4, or 8. Of course, this is very convenient for C data types!

80286

```
  char *p;
  int i;
  p[i] = 'A';
0000    8B 4E 0A          MOV   CX,[i]
0003    8B 46 08          MOV   AX,[p_hi]
0006    8B 5E 06          MOV   BX,[p_lo]
0009    9A 00 00 00 00-XX CALL  (FAR) _CHIX1
000E    26 C6 07 41       MOV   BYTE PTR ES:[BX],'A'
```

80386

```
  char *p;
  int i;
  p[i] = 'A';
00000000  8B 45 FC          MOV   EAX,[i]
00000003  8B 4D F8          MOV   ECX,[p]
00000006  C6 04 08 41       MOV   BYTE PTR [EAX+ECX],'A'
end parasp
```

Example 8: Simple 80386 Addressing Mode

80286

```
  int *int ptr, i, j;
  int_ptr[i] = j;
0000    8B 4E 0A          MOV   CX,[i]
0003    8B 46 08          MOV   AX,[int_ptr_hi]
0006    8B 5E 06          MOV   BX,[int_ptr_lo]
0009    9A 00 00 00 00-XX CALL  (FAR) _CHIX2
000E    8B 4E 0C          MOV   CX,[j]
0011    26 89 0F          MOV   ES:[BX],CX
```

80386

```
  int *int ptr, i, j;
  int_ptr[i] = j;
00000000  8B 45 F4          MOV   EAX,[j]
00000003  8B 4D FC          MOV   ECX,[int ptr]
00000006  8B 55 F8          MOV   EDX,[i]
00000009  89 04 91          MOV   [ECX+EDX*4],EAX
```

Example 9: 80386 Scaled Index Addressing Mode

FUTURE OF THE INTEL 80x86 FAMILY

Intel has stated that the evolution of the 80x86 product line will occur in a truly upwardly compatible fashion with the 80386. Long range plans call for processors through the 80686 to be developed to improve performance only. Thus, programs written for the 80386 today will remain viable across the new Intel line for several years.

The 80486 remains faithful to this goal. Though it has added an onboard numeric coprocessor, there are no new processing modes requiring extensive code modifications to take full advantage of the chip. Programs written for the 80386 run with full power on the 80486.

At the present time, Microsoft Windows/386 provides the best example of a PC-type operating system running in 386 protected mode. The high performance Windows/386 implementation provides an appealing target for large scale applications which find themselves restrained by the memory and co-processing limitations of MSDOS.

It seems inevitable that an 80386 native implementation of OS/2 will be introduced. It is only with the power of the 80386 that OS/2 will have its chance to demonstrate whether it can live up to its hype.

As far as UNIX-type operating systems, AIX 386, Sun386i UNIX and SCO UNIX System V/386 are examples of native 386 operating systems. These Workstation/PC environments seem positioned to challenge OS/2 for dominance of the 80386 operating system market.

CONCLUSION

When viewed as a 32-bit processor, able to directly address 4 gigabytes in a flat address space, the 80386 shows why it is well positioned to be the fundamental microprocessor design for a new generation of personal computers. Without the need for segmentation, large scale applications will move smoothly to these machines without the performance penalties associated with previous 80x86 generations.

The Institute's 80386 C compiler has been created with the SAS System in mind and promises exciting new implementations in the near future.

REFERENCES

Intel Corporation (1986), iAPX 86/88, 186/188 User's Manual, Santa Clara, CA

Intel Corporation (1985), iAPX 286 Programmer's Reference Manual, Santa Clara, CA

Intel Corporation (1986), 80386 Programmer's Reference Manual, Santa Clara, CA