Arithmetic Instructions

The 80x86 provides many arithmetic operations: addition, subtraction, negation, multiplication, division/modulo (remainder), and comparing two values. The instructions
that handle these operations are add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg, and some miscellaneous conversion instructions: aaa, aad, aam, aas, daa, and das.
The following sections describe these instructions in detail.
The generic forms for these instructions are

```
add dest, src dest := dest + src
adc dest, src dest := dest + src + C
SUB dest, src dest := dest - src
sbb dest, src dest := dest - src - C
mul src acc := acc * src
imul src acc := acc * src
imul dest, src1, imm_src dest := src1 * imm_src
imul dest, imm_src dest := dest * imm_src
imul dest, src dest := dest * src
div src acc := xacc /-mod src
idiv src acc := xacc /-mod src
cmp dest, src dest - src (and set flags)
neg dest dest := - dest
inc dest dest := dest + 1
dec dest dest := dest - 1
xadd dest, src (see text)
cmpxchg operand1, operand2 (see text)
cmpxchg8ax, operand (see text)
aaa (see text)
aad (see text)
aam (see text)
aas (see text)
daa (see text)
das (see text)
```

6.5.1 The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA

These instructions take the forms:

```
add reg, reg
add reg, mem
add mem, reg
add reg, immediate data
add mem, immediate data
add eax/ax/al, immediate data
adc forms are identical to ADD.
inc reg
inc mem
inc reg16
xadd mem, reg
xadd reg, reg
aaa
daa
```

Note that the aaa and daa instructions use the implied addressing mode and allow no operands.

6.5.1.1 The ADD and ADC Instructions

The syntax of add and adc (add with carry) is similar to mov. Like mov, there are special forms for the ax/eax register that are more efficient. Unlike mov, you cannot add a value to a segment register with these instructions.
The add instruction adds the contents of the source operand to the destination operand. For example, add ax, bx adds bx to ax leaving the sum in the ax register. Add computes dest :=dest+source while adc computes dest :=dest+source+C where C represents

the value in the carry flag. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.

Both instructions affect the flags identically. They set the flags as follows:

• The overflow flag denotes a signed arithmetic overflow.

• The carry flag denotes an unsigned arithmetic overflow.

• The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).

• The zero flag is set if the result of the addition is zero.

• The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.

• The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there are an even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote *even parity*). If there are an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote *odd parity*).

The add and adc instructions do not affect any other flags.

The add and adc instructions allow eight, sixteen, and (on the 80386 and later) thirty-two bit operands. Both source and destination operands must be the same size. See Chapter Nine if you want to add operands whose size is different.

Since there are no memory to memory additions, you must load memory operands into registers if you want to add two variables together. The following code examples demonstrate possible forms for the add instruction:

```
; J:= K + M
mov ax, K
add ax, M
mov J, ax
```

If you want to add several values together, you can easily compute the sum in a single register:

```
; J := K + M + N + P
mov ax, K
add ax, M
add ax, N
add ax, P
mov J, ax
```

If you want to reduce the number of hazards on an 80486 or Pentium processor, you can use code like the following:

```
mov bx, K
mov ax, M
add bx, N
add ax, P
add ax, bx
mov J, ax
```

One thing that beginning assembly language programmers often forget is that you can add a register to a memory location. Sometimes beginning programmers even believe that both operands have to be in registers, completely forgetting the lessons from Chapter Four. The 80x86 is a CISC processor that allows you to use memory addressing modes with various instructions like add. It is often more efficient to take advantages of the 80x86's memory addressing capabilities

```
; J := K + J
mov ax, K ;This works because addition is
add J, ax ; commutative!
; Often, beginners will code the above as one of the following two sequences.
; This is unnecessary!
```

```
mov ax, J ;Really BAD way to compute
mov bx, K ; J := J + K.
add ax, bx
mov J, ax
mov ax, J ;Better, but still not a good way to
add ax, K ; compute J := J + K
mov J, ax
```

Of course, if you want to add a constant to a memory location, you only need a single instruction. The 80x86 lets you directly add a constant to memory:

```
; J := J + 2
add J, 2
```

There are special forms of the add and adc instructions that add an immediate constant to the al, ax, or eax register. These forms are shorter than the standard add reg, immediate instruction. Other instructions also provide shorter forms when using these registers; therefore, you should try to keep computations in the accumulator registers (al, ax, and eax) as much as possible.

```
add bl, 2 ;Three bytes long
add al, 2 ;Two bytes long
add bx, 2 ;Four bytes long
add ax, 2 ;Three bytes long
etc.
```

Another optimization concerns the use of small signed constants with the add and adc instructions. If a value is in the range -128,,+127, the add and adc instructions will sign extend an eight bit immediate constant to the necessary destination size (eight, sixteen, or thirty-two bits). Therefore, you should try to use small constants, if possible, with the add and adc instructions.

6.5.1.2 The INC Instruction

The inc (increment) instruction adds one to its operand. Except for the carry flag, inc sets the flags the same way as add operand, 1 would.

Note that there are two forms of inc for 16 or 32 bit registers. They are the inc reg and inc reg16 instructions. The inc reg and inc mem instructions are the same. This instruction consists of an opcode byte followed by a mod-reg-r/m byte (see Appendix D for details). The inc reg16 instruction has a single byte opcode. Therefore, it is shorter and usually faster.

The inc operand may be an eight bit, sixteen bit, or (on the 80386 and later) thirty-two bit register or memory location.

The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction. Indeed, the inc reg16 instruction is one byte long, so it turns out that *two* such instructions are shorter than the comparable add reg, 1 instruction; however, the two increment instructions will run slower on most modern members of the 80x86 family.

The inc instruction is very important because adding one to a register is a very common operation. Incrementing loop control variables or indices into an array is a very common operation, perfect for the inc instruction. The fact that inc does not affect the carry

flag is very important. This allows you to increment array indices without affecting the result of a multiprecision arithmetic operation (see “Arithmetic and Logical Operations” on page 459 for more details about multiprecision arithmetic).

6.5.1.3 The XADD Instruction

Xadd (Exchange and Add) is another 80486 (and later) instruction. It does not appear on the 80386 and earlier processors. This instruction adds the source operand to the destination operand and stores the sum in the destination operand. However, just before storing the sum, it copies the original value of the destination operand into the source

operand. The following algorithm describes this operation:

```
xadd dest, source
```

```
temp := dest
dest := dest + source
source := temp
```

The xadd sets the flags just as the add instruction would. The xadd instruction allows eight, sixteen, and thirty-two bit operands. Both source and destination operands must be the same size.

6.5.1.4 The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. Beyond this chapter, this text will not cover BCD or ASCII arithmetic since it is mainly for controller applications, not general purpose programming applications. BCD values are decimal integer coded in binary form with one decimal digit (0..9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero.

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add them as though they were binary numbers and then execute the daa instruction afterwards to correct the results. Likewise, you can use the aaa instruction to adjust the result of an ASCII addition after executing an add instruction. Please note that these two instructions assume that the add operands were proper decimal or ASCII values. If you add binary (non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

The choice of the name "ASCII arithmetic" is unfortunate, since these values are not true ASCII characters. A name like "unpacked BCD" would be more appropriate. However, Intel uses the name ASCII, so this text will do so as well to avoid confusion. However, you will often hear the term "unpacked BCD" to describe this data type.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC5 =1) ) then
if (8088 or 8086)6 then
al := al + 6
else
ax := ax + 6
endif
ah := ah + 1
AuxC := 1 ;Set auxilliary carry
Carry := 1 ; and carry flags.
else
AuxC := 0 ;Clear auxilliary carry
Carry := 0 ; and carry flags.
endif
al := al and 0Fh
```

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers. This text will not deal with BCD or ASCII numeric strings, so you can safely ignore this instruction for now. Of course, you can use the aaa instruction any time you need to use the algorithm above, but that would probably be a rare situation.

The daa instruction functions like aaa except it handles packed BCD (binary code decimal) values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

```
if ( (AL and 0Fh) > 9 or (AuxC = 1)) then
al := al + 6
AuxC := 1 ;Set Auxilliary carry.
endif
if ( (al > 9Fh) or (Carry = 1)) then
```

```
al := al + 60h
Carry := 1; ;Set carry flag.
endif
```

6.5.2 The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS

The sub (subtract), sbb (subtract with borrow), dec (decrement), aas (ASCII adjust for subtraction), and das (decimal adjust for subtraction) instructions work as you expect. Their syntax is very similar to that of the add instructions:

```
sub reg, reg
sub reg, mem
sub mem, reg
sub reg, immediate data
sub mem, immediate data
sub eax/ax/al, immediate data
```

5. AuxC denotes the *auxiliary carry* flag in the flags register.

6. The 8086/8088 work differently from the later processors, but for all valid operands all 80x86 processors produce correct results.

```
sbb forms are identical to sub.
dec reg
dec mem
dec reg16
aas
das
```

The sub instruction computes the value dest := dest - src. The sbb instruction computes dest := dest - src - C. Note that subtraction is not commutative. If you want to compute the result for dest := src - dest you will need to use several instructions, assuming you need to preserve the source operand).

One last subject worth discussing is how the sub instruction affects the 80x86 flags register7. The sub, sbb, and dec instructions affect the flags as follows:

• They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.

• These instructions set the sign flag if the result is negative.

• These instructions set the overflow flag if signed overflow/underflow occurs.

• They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.

• They set the parity flag according to the number of one bits appearing in the result value.

• The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0..9) per byte. You would use this instruction after a sub or sbb instruction on the ASCII value. This instruction uses the following algorithm:

```
if ( (al and 0Fh) > 9 or AuxC = 1) then
al := al - 6
ah := ah - 1
AuxC := 1 ;Set auxilliary carry
Carry := 1 ; and carry flags.
else
AuxC := 0 ;Clear Auxilliary carry
Carry := 0 ; and carry flags.
endif
al := al and 0Fh
```

The das instruction handles the same operation for BCD values, it uses the following algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC = 1)) then
```

```
al := al -6
AuxC = 1
endif
if (al > 9Fh or Carry = 1) then
al := al - 60h
Carry := 1 ;Set the Carry flag.
endif
```

Since subtraction is not commutative, you cannot use the sub instruction as freely as the add instruction. The following examples demonstrate some of the problems you may encounter.

```
; J := K - J
mov ax, K ;This is a nice try, but it computes
sub J, ax ; J := J - K, subtraction isn't
; commutative!
```

7. The SBB instruction affects the flags in a similar fashion, just don't forget that SBB computes dest-source-C.

```
mov ax, K ;Correct solution.
sub ax, J
mov J, ax
; J := J - (K + M) -- Don't forget this is equivalent to J := J - K - M
mov ax, K ;Computes AX := K + M
add ax, M
sub J, ax ;Computes J := J - (K + M)
mov ax, J ;Another solution, though less
sub ax, K ;Efficient
sub ax, M
mov J, ax
```

Note that the sub and sbb instructions, like add and adc, provide short forms to subtract a constant from an accumulator register (al, ax, or eax). For this reason, you should try to keep arithmetic operations in the accumulator registers as much as possible. The sub and sbb instructions also provide a shorter form when subtracting constants in the range -128..+127 from a memory location or register. The instruction will automatically sign extend an eight bit signed value to the necessary size before the subtraction occurs. See Appendix D for the details.

In practice, there really isn't a need for an instruction that subtracts a constant from a register or memory location – adding a negative value achieves the same result. Nevertheless, Intel provides a subtract immediate instruction.

After the execution of a sub instruction, the condition code bits (carry, sign, overflow, and zero) in the flags register contain values you can test to see if one of sub's operands is equal, not equal, less than, less than or equal, greater than, or greater than or equal to the other operand. See the cmp instruction for more details.

6.5.3 The CMP Instruction

The cmp (compare) instruction is identical to the sub instruction with one crucial difference – it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is

```
cmp dest, src
```

The specific forms are

```
cmp reg, reg
cmp reg, mem
cmp mem, reg
cmp reg, immediate data
cmp mem, immediate data
cmp eax/ax/al, immediate data
```

The cmp instruction updates the 80x86's flags according to the result of the subtraction operation (dest - src). You can test the result of the comparison by checking the appropriate flags in the flags

register. For details on how this is done, see “The “Set on Condition” Instructions” on page 281 and “The Conditional Jump Instructions” on page 296.

Usually you’ll want to execute a conditional jump instruction after a cmp instruction. This two step process, comparing two values and setting the flag bits then testing the flag bits with the conditional jump instructions, is a very efficient mechanism for making decisions in a program.

Probably the first place to start when exploring the cmp instruction is to take a look at exactly how the cmp instruction affects the flags. Consider the following cmp instruction:

```
cmp ax, bx
```

This instruction performs the computation ax-bx and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative. At first glance, you might think that this flag would be set if ax is less than bx but this isn’t always the case. If ax=7FFFh and bx=-1 (0FFFFh) subtracting ax from bx produces 8000h, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn’t contain the proper status. For unsigned operands, consider ax=0FFFFh and bx=1. Ax is greater than bx but their difference is 0FFFEh which is still negative. As it turns out, the sign flag and the overflow flag, taken together, can be used for comparing two signed values.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflow or underflow. As mentioned above, the sign flag and the overflow flag are both used when performing signed comparisons.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow. This occurs only when ax is less than bx where ax and bx are both unsigned values.

The cmp instruction also affects the parity and auxiliary carry flags, but you’ll rarely test these two flags after a compare operation. Given that the cmp instruction sets the flags in this fashion, you can test the comparison of the two operands with the following flags:

```
cmp Oprnd1, Oprnd2
```

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:

If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Oprnd1 < Oprnd2 when using a signed comparison.

If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Oprnd1 >= Oprnd2 when using a signed comparison.

To understand why these flags are set in this manner, consider the following examples:

```
Oprnd1 minus Oprnd2 S O
------ ------ - -
0FFFF (-1) - 0FFFE (-2) 0 0
08000 - 00001 0 1
0FFFE (-2) - 0FFFF (-1) 1 0
07FFF (32767) - 0FFFF (-1) 1 1
```

Remember, the cmp operation is really a subtraction, therefore, the first example above computes (-1)-(-2) which is (+1). The result is positive and an overflow did not occur so both the S and O flags are zero. Since (S xor O) is zero, Oprnd1 is greater than or equal to Oprnd2.

In the second example, the cmp instruction would compute (-32768)-(+1) which is (-32769). Since a 16-bit signed integer cannot represent this value, the value wraps around to 7FFFh (+32767) and sets the overflow flag. Since the result is positive (at least within the confines of 16 bits) the sign flag is cleared. Since (S xor O) is one here, Oprnd1 is less than Oprnd2.

In the third example above, cmp computes (-2)-(-1) which produces (-1). No overflow occurred so the O flag is zero, the result is negative so the sign flag is one. Since (S xor O) is one, Oprnd1 is less than Oprnd2.

The Multiplication Instructions: MUL, IMUL, and AAM

The multiplication instructions provide you with your first taste of irregularity in the 8086's instruction set. Instructions like add, adc, sub, sbb, and many others in the 8086 instruction set use a mod-reg-r/m byte to support two operands. Unfortunately, there aren't enough bits in the 8086's opcode byte to support all instructions, so the 8086 uses the reg bits in the mod-reg-r/m byte as an opcode extension. For example, inc, dec, and neg do not require two operands, so the 80x86 CPUs use the reg bits as an extension to the eight bit opcode. This works great for single operand instructions, allowing Intel's designers to encode several instructions (eight, in fact) with a single opcode.

Unfortunately, the multiply instructions require special treatment and Intel's designers were still short on opcodes, so they designed the multiply instructions to use a single operand. The reg field contains an opcode extension rather than a register value. Of course, multiplication *is* a two operand function. The 8086 always assumes the accumulator (al, ax, or eax) is the destination operand. This irregularity makes using multiplication on the 8086 a little more difficult than other instructions because one operand has to be in the accumulator. Intel adopted this unorthogonal approach because they felt that programmers would use multiplication far less often than instructions like add and sub.

One problem with providing only a mod-reg-r/m form of the instruction is that you cannot multiply the accumulator by a constant; the mod-reg-r/m byte does not support the immediate addressing mode. Intel quickly discovered the need to support multiplication by a constant and provide some support for this in the 80286 processor[10]. This was especially important for multidimensional array access. By the time the 80386 rolled around, Intel generalized one form of the multiplication operation allowing standard mod-reg-r/m operands.

There are two forms of the multiply instruction: an unsigned multiplication (mul) and a signed multiplication (imul). Unlike addition and subtraction, you need separate instructions for these two operations.

The multiply instructions take the following forms:

10. On the original 8086 chip multiplication by a constant was always faster using shifts, additions, and subtractions. Perhaps Intel's designers didn't bother with multiplication by a constant for this reason. However, the 80286 multiply instruction was faster than the 8086 multiply instruction, so it was no longer true that multiplication was slower and the corresponding shift, add, and subtract instructions.

Unsigned Multiplication:

```
mul reg
mul mem
```

Signed (Integer) Multiplication:

```
imul reg
imul mem
imul reg, reg, immediate (2)
imul reg, mem, immediate (2)
imul reg, immediate (2)
imul reg, reg (3)
imul reg, mem (3)
```

BCD Multiplication Operations:

```
aam
2- Available on the 80286 and later, only.
3- Available on the 80386 and later, only.
```

As you can see, the multiply instructions are a real mess. Worse yet, you have to use an 80386 or later processor to get near full functionality. Finally, there are some restrictions on these instructions not obvious above. Alas, the only way to deal with these instructions is to memorize their operation.

Mul, available on all processors, multiplies unsigned eight, sixteen, or thirty-two bit operands. Note that when multiplying two n-bit values, the result may require as many as

2*n bits. Therefore, if the operand is an eight bit quantity, the result will require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits for the result.

The mul instruction, with an eight bit operand, multiplies the al register by the operand and stores the 16 bit result in ax. So

```
mul operand8
or imul operand8
```

computes:

```
ax := al * operand8
```

"*" represents an unsigned multiplication for mul and a signed multiplication for imul.

If you specify a 16 bit operand, then mul and imul compute:

```
dx:ax := ax * operand16
```

"*" has the same meanings as above and dx:ax means that dx contains the H.O. word of the 32 bit result and ax contains the L.O. word of the 32 bit result.

If you specify a 32 bit operand, then mul and imul compute the following:

```
edx:eax := eax * operand32
```

"*" has the same meanings as above and edx:eax means that edx contains the H.O. double word of the 64 bit result and eax contains the L.O. double word of the 64 bit result.

If an 8x8, 16x16, or 32x32 bit product requires more than eight, sixteen, or thirty-two bits (respectively), the mul and imul instructions set the carry and overflow flags.

Mul and imul scramble the A, P, S, and Z flags. Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions.

Imul (integer multiplication) operates on signed operands. There are many different forms of this instruction as Intel attempted to generalize this instruction with successive processors. The previous paragraphs describe the first form of the imul instruction, with a single operand. The next three forms of the imul instruction are available only on the 80286 and later processors. They provide the ability to multiply a register by an immediate value. The last two forms, available only on 80386 and later processors, provide the ability to multiply an arbitrary register by another register or memory location. Expanded to show allowable operand sizes, they are

```
imul operand1, operand2, immediate ;General form
imul reg16, reg16, immediate8
imul reg16, reg16, immediate16
imul reg16, mem16, immediate8
imul reg16, mem16, immediate16
imul reg16, immediate8
imul reg16, immediate16
imul reg32, reg32, immediate8 (3)
imul reg32, reg32, immediate32 (3)
imul reg32, mem32, immediate8 (3)
imul reg32, mem32, immediate32 (3)
imul reg32, immediate8 (3)
imul reg32, immediate32 (3)
3- Available on the 80386 and later, only.
```

The imul reg, immediate instructions are a special syntax the assembler provides. The encodings for these instructions are the same as imul reg, reg, immediate. The assembler simply supplies the same register value for both operands.

These instructions compute:

```
operand1 := operand2 * immediate
operand1 := operand1 * immediate
```

Besides the number of operands, there are several differences between these forms and the single operand mul/imul instructions:

• There isn't an 8x8 bit multiplication available (the immediate8 operands

simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).

• These instructions do not produce a 2*n bit result. That is, a 16x16 multiply produces a 16 bit result. Likewise, a 32x32 bit multiply produces a 32 bit result. These instructions set the carry and overflow flags if the result does not fit into the destination register.

• The 80286 version of imul allows an immediate operand, the standard mul/imul instructions do not.

The last two forms of the imul instruction are available only on 80386 and later processors. With the addition of these formats, the imul instruction is *almost* as general as the add instruction[11]:

```
imul reg, reg
imul reg, mem
```

These instructions compute

```
reg := reg * reg
and reg := reg * mem
```

Both operands must be the same size. Therefore, like the 80286 form of the imul instruction, you must test the carry or overflow flag to detect overflow. If overflow does occur, the CPU loses the H.O. bits of the result.

Important Note: Keep in mind that the zero flag contains an indeterminate result after executing a multiply instruction. You cannot test the zero flag to see if the result is zero after a multiplication. Likewise, these instructions scramble the sign flag. If you need to check these flags, compare the result to zero after testing the carry or overflow flags.

The aam (ASCII Adjust after Multiplication) instruction, like aaa and aas, lets you adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81, the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al:

11. There are still some restrictions on the size of the operands, e.g., no eight bit registers, you have to consider.

```
ah := ax div 10
al := ax mod 10
```

Unlike the other decimal/ASCII adjust instructions, assembly language programs regularly use aam since conversion between number bases uses this algorithm.

Note: the aam instruction consists of a two byte opcode, the second byte of which is the immediate constant 10. Assembly language programmers have discovered that if you substitute another immediate value for this constant, you can change the divisor in the above algorithm. This, however, is an undocumented feature. It works in all varieties of the processor Intel has produced to date, but there is no guarantee that Intel will support this in future processors. Of course, the 80286 and later processors let you multiply by a constant, so this trick is hardly necessary on modern systems.

There is no dam (decimal adjust for multiplication) instruction on the 80x86 processor.

Perhaps the most common use of the imul instruction is to compute offsets into multidimensional arrays. Indeed, this is probably the main reason Intel added the ability to multiply a register by a constant on the 80286 processor. In Chapter Four, this text used the standard 8086 mul instruction for array index computations. However, the extended syntax of the imul instruction makes it a much better choice as the following examples demonstrate:

```
MyArray word 8 dup ( 7 dup ( 6 dup (?))) ;8x7x6 array.
J word ?
K word ?
M word ?
```

```
.
.
.
; MyArray [J, K, M] := J + K - M
mov ax, J
add ax, K
sub ax, M
mov bx, J ;Array index :=
imul bx, 7 ; ((J*7 + K) * 6 + M) * 2
add bx, K
imul bx, 6
add bx, M
add bx, bx ;BX := BX * 2
mov MyArray[bx], ax
```

Don't forget that the multiplication instructions are very slow; often an order of magnitude slower than an addition instruction. There are faster ways to multiply a value by a constant. See "Multiplying Without MUL and IMUL" on page 487 for all the details.

6.5.7 The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16 division or a 16/8 division. These instructions take the form:

```
div reg For unsigned division
div mem
idiv reg For signed division
idiv mem
aad ASCII adjust for division
```

The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in . With 32 bit operands (on the 80386 and later) div divides the 64 bit value in edx:eax by the operand leaving the quotient in eax and the remainder in edx.

You cannot, on the 80x86, simply divide one eight bit value by another. If the denominator is an eight bit value, the numerator must be a sixteen bit value. If you need to divide one unsigned eight bit value by another, you must zero extend the numerator to sixteen bits. You can accomplish this by loading the numerator into the al register and then moving zero into the ah register. Then you can divide ax by the denominator operand to produce the correct result. *Failing to zero extend al before executing div may cause the 80x86 to produce incorrect results!*

When you need to divide two 16 bit unsigned values, you must zero extend the ax register (which contains the numerator) into the dx register. Just load the immediate value zero into the dx register[12]. If you need to divide one 32 bit value by another, you must zero extend the eax register into edx (by loading a zero into edx) before the division.

When dealing with signed integer values, you will need to sign extend al to ax, ax to dx or eax into edx before executing idiv. To do so, use the cbw, cwd, cdq, or movsx instructions. If the H.O. byte or word does not already contain significant bits, then you must sign extend the value in the accumulator (al/ax/eax) before doing the idiv operation. Failure to do so may produce incorrect results.

There is one other catch to the 80x86's divide instructions: you can get a fatal error when using this instruction. First, of course, you can attempt to divide a value by zero. Furthermore, the quotient may be too large to fit into the eax, ax, or al register. For example, the 16/8 division "8000h / 2" produces the quotient 4000h with a remainder of zero. 4000h will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *int 0* trap. This usually means BIOS will print "division by zero" or

“divide error” and abort your program. If this happens to you, chances are you didn’t sign or zero extend your numerator before executing the division operation. Since this error will cause your program to crash, you should be very careful about the values you select when using division.

The auxiliary carry, carry, overflow, parity, sign, and zero flags are undefined after a division operation. If an overflow occurs (or you attempt a division by zero) then the 80x86 executes an INT 0 (interrupt zero).

Note that the 80286 and later processors do not provide special forms for idiv as they do for imul. Most programs use division far less often than they use multiplication, so Intel’s designers did not bother creating special instructions for the divide operation. Note that there is no way to divide by an immediate value. You must load the immediate value into a register or a memory location and do the division through that register or memory location.

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation. It splits apart unpacked binary coded decimal values before an ASCII division operation. Although this text will not cover BCD arithmetic, the aad instruction is useful for other operations. The algorithm that describes this instruction is

```
al := ah*10 + al
ah := 0
```

This instruction is quite useful for converting strings of digits into integer values (see the questions at the end of this chapter).

The following examples show how to divide one sixteen bit value by another.

```
; J := K / M (unsigned)
mov ax, K ;Get dividend
mov dx, 0 ;Zero extend unsigned value in AX to DX.
< In practice, we should verify that M does not contain zero here >
div M
mov J, ax
; J := K / M (signed)
```