Integer Arithmetic Chapter Ten

10.1 Chapter Overview

This chapter discusses the implementation of arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate (integer) arithmetic expressions and assignment statements from high level languages like Pascal and C/C++ into 80x86 assembly language.

10.2 80x86 Integer Arithmetic Instructions

Before describing how to encode arithmetic expressions in assembly language, it would be a good idea to first discuss the remaining arithmetic instructions in the 80x86 instruction set. Previous chapters have covered most of the arithmetic and logical instructions, so this section will cover the few remaining instructions you'll need.

10.2.1 The MUL and IMUL Instructions

The multiplication instructions provide you with another taste of irregularity in the 80x86's instruction set. Instructions like ADD, SUB, and many others in the 80x86 instruction set support two operands. Unfortunately, there weren't enough bits in the 80x86's opcode byte to support all instructions, so the 80x86 treats the MUL (unsigned multiply) and IMUL (signed integer multiply) instructions as single operand instructions, like the INC, DEC, and NEG instructions.

Of course, multiplication *is* a two operand function. To work around this fact, the 80x86 always assumes the accumulator (AL,AX, or EAX) is the destination operand. This irregularity makes using multiplication on the 80x86 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.

Another problem with the MUL and IMUL instructions is that you cannot multiply the accumulator by a constant using these instructions. Intel quickly discovered the need to support multiplication by a constant and added the INTMUL instruction to overcome this problem. Nevertheless, you must be aware that the basic MUL and IMUL instructions do not support the full range of operands that INTMUL does.

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

The multiply instructions take the following forms:

Unsigned Multiplication:

```
mul( reg8 );        // returns "ax"
mul( reg16 );       // returns "dx:ax"
mul( reg32 );       // returns "edx:eax"

mul( mem8 );        // returns "ax"
mul( mem16 );       // returns "dx:ax"
mul( mem32 );       // returns "edx:eax"
```

Signed (Integer) Multiplication:

```
imul( reg8 );       // returns "ax"
imul( reg16 );      // returns "dx:ax"
imul( reg32 );      // returns "edx:eax"
```



```
imul( mem8 );       // returns "ax"
imul( mem16 );      // returns "dx:ax"
imul( mem32 );      // returns "edx:eax"
```

The "returns" values above are the strings these instructions return for use with instruction composition in HLA (see "Instruction Composition in HLA" on page 558).

(I)MUL, available on all 80x86 processors, multiplies 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 could require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits to hold the result.

The (I)MUL instruction, with an eight bit operand, multiplies the AL register by the operand and leaves the 16 bit product 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're wondering why Intel didn't put the 32-bit result in EAX, just note that Intel introduced the MUL and IMUL instructions in the earliest 80x86 processors, before the advent of 32-bit registers in the 80386 CPU.

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 sign, and zero flags. **Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions**.

To help reduce some of the problems with the use of the MUL and IMUL instructions, HLA provides an extended syntax that allows the following two-operand forms:

Unsigned Multiplication:

```
mul( reg8, al );
mul( reg16, ax );
mul( reg32, eax );

mul( mem8, al );
mul( mem16, ax );
mul( mem32, eax );

mul( constant8, al );
mul( constant16, ax );
mul( constant32, eax );
```

Signed (Integer) Multiplication:


```
imul( reg8, al );
imul( reg16, ax );
imul( reg32, eax );

imul( mem8, al );
imul( mem16, ax );
imul( mem32, eax );

imul( constant8, al );
imul( constant16, ax );
imul( constant32, eax );
```

The two operand forms let you specify the (L.O.) destination register. The instructions whose first operand is a register or memory location are completely identical to the instructions above. By specifying the destination register, however, you can make your programs easier to read; therefore, it's probably a good idea to go ahead and specify the destination register. Note that just because HLA allows two operands here, you can't specify an arbitrary register. The destination operand must always be AL, AX, or EAX, depending on the source operand.

Note that HLA allows a form that lets you specify a constant. The 80x86 doesn't actually support a MUL or IMUL instruction that has a constant operand. HLA will take the constant you specify and create a "variable" in the special "const" segment in memory and initialize that variable with this value. Then HLA converts the instruction to the "(I)MUL(memory);" instruction. Generally, you won't need to use this special form since the INTMUL instruction will multiply a register by a constant.

You'll use the MUL and IMUL instructions quite a bit when you learn about extended precision arithmetic in the chapter on Advanced Arithmetic. Until you get to that chapter, you'll probably just want to use the INTMUL instruction in place of the MUL or IMUL since it is more general. However, INTMUL is not a complete replacement for these two instructions. Besides the number of operands, there are several differences between the INTMUL instruction you've learned about earlier and the MUL and IMUL instructions. Specifically for the INTMUL instruction:

- There isn't an 8x8 bit INTMUL instruction available (the $immediate_8$ operands simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).
- The INTMUL instruction does 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.

10.2.2 The DIV and IDIV Instructions

The 80x86 divide instructions perform a 64/32 division, a 32/16 division or a 16/8 division. These instructions take the form:

```
div( reg8 );              // returns "al"
div( reg16 );             // returns "ax"
div( reg32 );             // returns "eax"

div( reg8, AX );          // returns "al"
div( reg16, DX:AX );
div( reg32, EDX:EAX );

div( mem8 );              // returns "al"
div( mem16 );             // returns "ax"
div( mem32 );             // returns "eax"
```



```
div( mem8, AX );          // returns "al"
div( mem16, DX:AX );      // returns "ax"
div( mem32, EDX:EAX );    // returns "eax"

div( constant8, AX );         // returns "al"
div( constant16, DX:AX );     // returns "ax"
div( constant32, EDX:EAX );   // returns "eax"

idiv( reg8 );                 // returns "al"
idiv( reg16 );                // returns "ax"
idiv( reg32 );                // returns "eax"

idiv( reg8, AX );             // returns "al"
idiv( reg16, DX:AX );         // returns "ax"
idiv( reg32, EDX:EAX );       // returns "eax"

idiv( mem8 );                 // returns "al"
idiv( mem16 );                // returns "ax"
idiv( mem32 );                // returns "eax"

idiv( mem8, AX );             // returns "al"
idiv( mem16, DX:AX );         // returns "ax"
idiv( mem32, EDX:EAX );       // returns "eax"

idiv( constant8, AX );       // returns "al"
idiv( constant16, DX:AX );   // returns "ax"
idiv( constant32, EDX:EAX ); // returns "eax"
```

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 DX. With 32 bit operands 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. To do this, just load zero into the DX register. 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 into AX, AX into 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. Second, the quotient may be too large to fit into the EAX, AX, or AL register. For example, the 16/8 division "$8000 / 2" produces the quotient $4000 with a remainder of zero. $4000 will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *ex.DivisionError* exception or integer overflow error (*ex.IntoInstr*). This usually means your program will display the appropriate dialog box 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. Of course, you can use the TRY..ENDTRY block with the *ex.DivisionError* and *ex.IntoInstr* to trap this problem in your program.

The carry, overflow, sign, and zero flags are undefined after a division operation. Like MUL and IMUL, HLA provides special syntax to allow the use of constant operands even though these instructions don't really support them.

The 80x86 does not provide a separate instruction to compute the remainder of one number divided by another. The DIV and IDIV instructions automatically compute the remainder at the same time they compute the quotient. HLA, however, provides mnemonics (instructions) for the MOD and IMOD instructions. These special HLA instructions compile into the exact same code as their DIV and IDIV counterparts. The only difference is the "returns" value for the instruction (since these instructions return the remainder in a different location than the quotient). The MOD and IMOD instructions that HLA supports are

```
mod( reg8 );                 // returns "ah"
mod( reg16 );                // returns "dx"
mod( reg32 );                // returns "edx"

mod( reg8, AX );             // returns "ah"
mod( reg16, DX:AX );         // returns "dx"
mod( reg32, EDX:EAX );       // returns "edx"

mod( mem8 );                 // returns "ah"
mod( mem16 );                // returns "dx"
mod( mem32 );                // returns "edx"

mod( mem8, AX );             // returns "ah"
mod( mem16, DX:AX );         // returns "dx"
mod( mem32, EDX:EAX );       // returns "edx"

mod( constant8, AX );        // returns "ah"
mod( constant16, DX:AX );    // returns "dx"
mod( constant32, EDX:EAX );  // returns "edx"

imod( reg8 );                // returns "ah"
imod( reg16 );               // returns "dx"
imod( reg32 );               // returns "edx"

imod( reg8, AX );            // returns "ah"
imod( reg16, DX:AX );        // returns "dx"
imod( reg32, EDX:EAX );      // returns "edx"

imod( mem8 );                // returns "ah"
imod( mem16 );               // returns "dx"
imod( mem32 );               // returns "edx"

imod( mem8, AX );            // returns "ah"
imod( mem16, DX:AX );        // returns "dx"
imod( mem32, EDX:EAX );      // returns "edx"

imod( constant8, AX );       // returns "ah"
imod( constant16, DX:AX );   // returns "dx"
imod( constant32, EDX:EAX ); // returns "edx"
```


10.2.3 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 similar to SUB (though the operands are reversed so it reads better), the generic form is

```
cmp( LeftOperand, RightOperand );
```

This instruction computes "LeftOperand - RightOperand" (note the reversal from SUB). The specific forms are

```
cmp( reg, reg );      // Registers must be the same size (8, 16, or 32 bits)
cmp( reg, mem );      // Sizes must match.
cmp( reg, constant );
cmp( mem, constant );
```

Note that both operands are "source" operands, so the fact that a constant appears as the second operand is okay.

The CMP instruction updates the 80x86's flags according to the result of the subtraction operation (LeftOperand - RightOperand). The flags are generally set in an appropriate fashion so that we can read this instruction as "compare LeftOperand to RightOperand". You can test the result of the comparison by checking the appropriate flags in the flags register using the conditional set instructions (see the next section) or the conditional jump instructions.

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=$7FFF and BX= -1 ($FFFF) subtracting AX from BX produces $8000, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn't contain the proper status. For unsigned operands, consider AX=$FFFF and BX=1. AX is greater than BX but their difference is $FFFE 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.

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( Left, Right );
```


Table 1: Condition Code Settings After CMP

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Left < Right (C=1) Left >= Right (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

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 Left < Right when using a signed comparison.

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

Note that (S xor O) is one if the left operand is less than the right operand. Conversely, (S xor O) is zero if the left operand is greater or equal to the right operand.

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

```
Left          minus     Right          S    O
------                  ------         -    -

$FFFF (-1)      -       $FFFE (-2)     0    0
$8000           -       $0001          0    1
$FFFE (-2)      -       $FFFF (-1)     1    0
$7FFF (32767)   -       $FFFF (-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, Left is greater than or equal to Right.

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 $7FFF (+32767) and sets the overflow flag. The result is positive (at least as a 16 bit value) so the CPU clears the sign flag. (S xor O) is one here, so *Left* is less than *Right*.

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, Left is less than Right.

In the fourth (and final) example, CMP computes (+32767)-(-1). This produces (+32768), setting the overflow flag. Furthermore, the value wraps around to $8000 (-32768) so the sign flag is set as well. Since (S xor O) is zero, Left is greater than or equal to Right.

10.2.4 The SETcc Instructions

The *set on condition* (or SET*cc*) instructions set a single byte operand (register or memory location) to zero or one depending on the values in the flags register. The general formats for the SET*cc* instructions are

```
setcc( reg8 );
setcc( mem8 );
```


SET*cc* represents a mnemonic appearing in the following tables. These instructions store a zero into the corresponding operand if the condition is false, they store a one into the eight bit operand if the condition is true.

Table 2: SETcc Instructions That Test Flags

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

The SET*cc* instructions above simply test the flags without any other meaning attached to the operation. You could, for example, use SETC to check the carry flag after a shift, rotate, bit test, or arithmetic operation. You might notice the SETP, SETPE, and SETNP instructions above. They check the *parity* flag. These instructions appear here for completeness, but this text will not consider the uses of the parity flag.

The CMP instruction works synergistically with the SET*cc* instructions. Immediately after a CMP operation the processor flags provide information concerning the relative values of those operands. They allow you to see if one operand is less than, equal to, greater than, or any combination of these.

There are two additional groups of SET*cc* instructions that are very useful after a CMP operation. The first group deals with the result of an *unsigned* comparison, the second group deals with the result of a *signed* comparison.

Table 3: SETcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The corresponding table for signed comparisons is

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw and Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETNGE	Set if not greater or equal (not >=)	Sign ≠ Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The SET*cc* instructions are particularly valuable because they can convert the result of a comparison to a boolean value (false/true or 0/1). This is especially important when translating statements from a high level language like Pascal or C/C++ into assembly language. The following example shows how to use these instructions in this manner:

```
// Bool := A <= B

        mov( A, eax );
        cmp( eax, B );
        setle( bool );    // bool is a boolean or byte variable.
```

Since the SET*cc* instructions always produce zero or one, you can use the results with the AND and OR instructions to compute complex boolean values:

```
// Bool := ((A <= B) and (D = E))

        mov( A, eax );
        cmp( eax, B );
        setle( bl );
        mov( D, eax );
        cmp( eax, E );
        sete( bh );
        and( bl, bh );
        mov( bh, Bool );
```

For more examples, see “Logical (Boolean) Expressions” on page 604.

10.2.5 The TEST Instruction

The 80x86 TEST instruction is to the AND instruction what the CMP instruction is to SUB. That is, the TEST instruction computes the logical AND of its two operands and sets the condition code flags based on the result; it does not, however, store the result of the logical AND back into the destination operand. The syntax for the TEST instruction is similar to AND, it is

```
test( operand1, operand2 );
```

The TEST instruction sets the zero flag if the result of the logical AND operation is zero. It sets the sign flag if the H.O. bit of the result contains a one. TEST always clears the carry and overflow flags.

The primary use of the TEST instruction is to check to see if an individual bit contains a zero or a one. Consider the instruction “test(1, AL);” This instruction logically ANDs AL with the value one; if bit one of AL contains zero, the result will be zero (setting the zero flag) since all the other bits in the constant one are

zero. Conversely, if bit one of AL contains one, then the result is not zero so TEST clears the zero flag. Therefore, you can test the zero flag after this TEST instruction to see if bit zero contains a zero or a one.

The TEST instruction can also check to see if all the bits in a specified set of bits contain zero. The instruction "test($F, AL);" sets the zero flag if and only if the L.O. four bits of AL all contain zero.

One very important use of the TEST instruction is to check to see if a register contains zero. The instruction "TEST(reg, reg);" where both operands are the same register will logically AND that register with itself. If the register contains zero, then the result is zero and the CPU will set the zero flag. However, if the register contains a non-zero value, logically ANDing that value with itself produces that same non-zero value, so the CPU clears the zero flag. Therefore, you can test the zero flag immediately after the execution of this instruction (e.g., using the SETZ or SETNZ instructions) to see if the register contains zero. E.g.,

```
test( eax, eax );
setz( bl );          // BL is set to one if EAX contains zero.
```

10.3 Arithmetic Expressions

Probably the biggest shock to beginners facing assembly language for the very first time is the lack of familiar arithmetic expressions. Arithmetic expressions, in most high level languages, look similar to their algebraic equivalents, e.g.,

```
X:=Y*Z;
```

In assembly language, you'll need several statements to accomplish this same task, e.g.,

```
mov( y, eax );
intmul( z, eax );
mov( eax, x );
```

Obviously the HLL version is much easier to type, read, and understand. This point, more than any other, is responsible for scaring people away from assembly language.

Although there is a lot of typing involved, converting an arithmetic expression into assembly language isn't difficult at all. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break down any arithmetic expression into an equivalent sequence of assembly language statements. By learning how to convert such expressions to assembly language in three steps, you'll discover there is little difficulty to this task.

10.3.1 Simple Assignments

The easiest expressions to convert to assembly language are the simple assignments. Simple assignments copy a single value into a variable and take one of two forms:

```
variable := constant
```

or

```
variable := variable
```

Converting the first form to assembly language is trivial, just use the assembly language statement:

```
mov( constant, variable );
```

This MOV instruction copies the constant into the variable.

The second assignment above is slightelly more complicated since the 80x86 doesn't provide a memory–to-memory MOV instruction. Therefore, to copy one memory variable into another, you must move the data through a register. By convention (and for slight efficiency reasons), most programmers tend to use AL/AX/EAX for this purpose. If the AL, AX, or EAX register is available, you should use it for this operation. For example,


```
var1 := var2;
```

becomes

```
mov( var2, eax );
mov( eax, var1 );
```

This is assuming, of course, that *var1* and *var2* are 32-bit variables. Use AL if they are eight bit variables, use AX if they are 16-bit variables.

Of course, if you're already using AL, AX, or EAX for something else, one of the other registers will suffice. Regardless, you must use a register to transfer one memory location to another.

Although the 80x86 does not support a memory-to-memory move, HLA does provide an extended syntax for the MOV instruction that allows two memory operands. However, both operands have to be 16-bit or 32-bit values; eight-bit values won't work. Assuming you want to copy the value of a word or dword object to another variable, you can use the following syntax:

```
mov( var2, var1 );
```

HLA translates this "instruction" into the following two instruction sequence:

```
push( var2 );
pop( var1 );
```

Although this is slightly slower than the two MOV instructions, it is convenient.

10.3.2 Simple Expressions

The next level of complexity up from a simple assignment is a simple expression. A simple expression takes the form:

```
var1 := term1 op term2;
```

Var1 is a variable, *term1* and *term2* are variables or constants, and *op* is some arithmetic operator (addition, subtraction, multiplication, etc.).

As simple as this expression appears, most expressions take this form. It should come as no surprise then, that the 80x86 architecture was optimized for just this type of expression.

A typical conversion for this type of expression takes the following form:

```
mov( term1, eax );
op( term2, eax );
mov( eax, var1 )
```

Op is the mnemonic that corresponds to the specified operation (e.g., "+" = add, "-" = sub, etc.).

There are a few inconsistencies you need to be aware of. Of course, when dealing with the multiply and divide instructions on the 80x86, you must use the AL/AX/EAX and DX/EDX registers. You cannot use arbitrary registers as you can with other operations. Also, don't forget the sign extension instructions if you're performing a division operation and you're dividing one 16/32 bit number by another. Finally, don't forget that some instructions may cause overflow. You may want to check for an overflow (or underflow) condition after an arithmetic operation.

Examples of common simple expressions:

```
x := y + z;
```

```
mov( y, eax );
add( z, eax );
mov( eax, x );
```

```
x := y - z;
```



```
        mov( y, eax );
        sub( z, eax );
        mov( eax, x );

x := y * z; {unsigned}

        mov( y, eax );
        mul( z, eax );     // Don't forget this wipes out EDX.
        mov( eax, x );

x := y div z; {unsigned div}

        mov( y, eax );
        mov( 0, edx );       // Zero extend EAX into EDX.
        div( z, edx:eax );
        mov( eax, x );

x := y idiv z; {signed div}

        mov( y, eax );
        cdq();               // Sign extend EAX into EDX.
        idiv( z, edx:eax );
        mov( eax, z );

x := y mod z; {unsigned remainder}

        mov( y, eax );
        mov( 0, edx );       // Zero extend EAX into EDX.
        mod( z, edx:eax );
        mov( edx, x );       // Note that remainder is in EDX.

x := y imod z; {signed remainder}

        mov( y, eax );
        cdq();               // Sign extend EAX into EDX.
        imod( z, edx:eax );
        mov( edx, x );       // Remainder is in EDX.
```

Certain unary operations also qualify as simple expressions. A good example of a unary operation is negation. In a high level language negation takes one of two possible forms:

```
var := -var  or   var1 := -var2
```

Note that var := -constant is really a simple assignment, not a simple expression. You can specify a negative constant as an operand to the MOV instruction:

```
mov( -14, var );
```

To handle "var = -var;" use the single assembly language statement:

```
// var = -var;

neg( var );
```

If two different variables are involved, then use the following:

```
// var1 = -var2;

mov( var2, eax );
neg( eax );
mov( eax, var1 );
```


10.3.3 Complex Expressions

A complex expression is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in a high level language. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, etc. While the conversion of some complex expressions to assembly language is fairly straight-forward, others require some effort. This section outlines the rules you use to convert such expressions.

A complex expression that is easy to convert to assembly language is one that involves three terms and two operators, for example:

```
w := w - y - z;
```

Clearly the straight-forward assembly language conversion of this statement will require two SUB instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```
mov( w, eax );
sub( y, eax );
sub( z, eax );
mov( eax, w );
```

and

```
mov( y, eax );
sub( z, eax );
sub( eax, w );
```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result (assuming Pascal-like semantics for the original statement). Associativity is the problem. The second sequence above computes W := W - (Y - Z) which is not the same as W := (W - Y) - Z. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```
mov( y, eax );
add( z, eax );
sub( eax, w );
```

This computes W:=W-(Y+Z). This is equivalent to W := (W - Y) - Z.

Precedence is another issue. Consider the Pascal expression:

```
X := W * Y + Z;
```

Once again there are two ways we can evaluate this expression:

```
X := (W * Y) + Z;
```

or

```
X := W * (Y + Z);
```

By now, you're probably thinking that this text is crazy. Everyone knows the correct way to evaluate these expressions is the second form provided in these two examples. However, you're wrong to think that way. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another.

Most high level languages use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Such programming languages usually compute multiplication and division before addition and subtraction. Those that support exponentiation (e.g., FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive since almost everyone learns them before high school. Consider the expression:

$$X\ op_1\ Y\ op_2\ Z$$

If op_1 takes precedence over op_2 then this evaluates to (X op_1 Y) op_2 Z otherwise if op_2 takes precedence over op_1 then this evaluates to X op_1 (Y op_2 Z). Depending upon the operators and operands involved, these two computations could produce different results.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
// w := x + y * z;

        mov( x, ebx );
        mov( y, eax );      // Must compute y*z first since "*"
        intmul( z, eax );   //  has higher precedence than "+".
        add( ebx, eax );
        mov( eax, w );
```

If two operators appearing within an expression have the same precedence, then you determine the order of evaluation using *associativity* rules. Most operators are *left associative* meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left associative. A *right associative* operator evaluates from right to left. The exponentiation operator in FORTRAN and BASIC is a good example of a right associative operator:

```
        2^2^3 is equal to 2^(2^3) not (2^2)^3
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w := x - y - z

        mov( x, eax );   // All the same operator, so we need
        sub( y, eax );   //  to evaluate from left to right
        sub( z, eax );   //  because they all have the same
        mov( eax, w );   //  precedence and are left associative.

// w := x + y * z

        mov( y, eax );      // Must compute Y * Z first since
        intmul( z, eax );   // multiplication has a higher
        add( x, eax );      // precedence than addition.
        mov( eax, w );


// w := x / y - z

        mov( x, eax );      // Here we need to compute division
        cdq();              //  first since it has the highest
        idiv( y, edx:eax ); //  precedence.
        sub( z, eax );
        mov( eax, w );

// w := x * y * z

        mov( y, eax );      // Addition and multiplication are
        intmul( z, eax );   // commutative, therefore the order
        intmul( x, eax );   // of evaluation does not matter
        mov( eax, w );
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is generally better to perform the multiplication first. For example, given an expression of the form:

```
        W := X/Y * Z        // Note: this is x/y * z  not  x/(y*z) !
```

It is usually better to compute X*Z and then divide the result by Y rather than divide X by Y and multiply the quotient by Z. There are two reasons this approach is better. First, remember that the IMUL instruction always produces a 64 bit result (assuming 32 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the EDX register so you do not have to sign extend EAX prior to the division. This saves the execution of the CDQ instruction. A second reason for doing the multiplication first is to

increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
w := x/y*z;
```

You can usually convert it to the assembly code:

```
mov( x, eax );
imul( z, eax );       // Note the use of IMUL, not INTMUL!
idiv( y, edx:eax );
mov( eax, w );
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following Pascal statement:

```
w := x - y * x;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute *y* * *z* and then subtract *x* from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load x into a register, multiply *y* and *z* leaving their product in a different register, and then subtract this product from *x*, e.g.,

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of *x* until it computes the product of *y* and *z*. As your expressions increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

```
w := (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

```
Temp1 := a + b;
Temp2 := y + z;
w := Temp1 * Temp2;
```

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov( a, eax );
add( b, eax );
mov( eax, Temp1 );
mov( y, eax );
add( z, eax );
mov( eax, Temp2 );
mov( Temp1, eax );
intmul( Temp2, eax );
mov( eax, w );
```


Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```

Yet another example:

```
x := (y+z) * (a-b) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 := (y+z)
Temp2 := (a-b)
Temp1 := Temp1 * Temp2
X := Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov( y, eax );      // Compute eax = y+z
add( z, eax );
mov( a, ebx );      // Compute ebx = a-b
sub( b, ebx );
imul( ebx, eax );   // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that temporary values, if possible, should be kept in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the results. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

10.3.4 Commutative Operators

If "@" represents some operator, that operator is *commutative* if the following relationship is always true:

```
(A @ B) = (B @ A)
```

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should always check to see if there is a better sequence you can use to improve the size or speed of your code. The following tables list the commutative and non-commutative operators you typically find in high level languages:

Table 5: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	|| or |	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Table 6: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

10.4 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

```
B := ((X=Y) and (A <= C)) or ((Z-A) <> 5);
```

B is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose (as such, HLA also allocates a whole byte for a BOOLEAN variable). With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like "C") choose zero to represent false and anything else to represent true. Some people prefer to represent true and false with one

and zero (respectively) and not allow any other values. Others select all one bits ($FFFF_FFFF, $FFFF, or $FF) for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their own advantages and drawbacks.

Using only zero and one to represent false and true offers two very big advantages: (1) The SETcc instructions produce these results so this scheme is compatible with those instructions; (2) the 80x86 logical instructions (AND, OR, XOR and, to a lesser extent, NOT) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

```
// c = a AND b;

    mov( a, al );
    and( b, al );
    mov( al, c );

// c = a OR b;

        mov( a, al );
        or( b, al );
        mov( al, c );

// c = a XOR b;

        mov( a, al );
        xor( b, al );
        mov( al, c );

// b = not a;

        mov( a, al );     // Note that the NOT instruction does not
        not( al );        // properly compute al = not al by itself.
        and( 1, al );     // I.e., (not 0) does not equal one.  The AND
        mov( al, b );     // instruction corrects this problem.

        mov( a, al );     // Another way to do b = not a;
        xor( 1, al );     // Inverts bit zero.
        mov( al, b );
```

Note, as pointed out above, that the NOT instruction will not properly compute logical negation. The bitwise not of zero is $FF and the bitwise not of one is $FE. Neither result is zero or one. However, by ANDing the result with one you get the proper result. Note that you can implement the NOT operation more efficiently using the "xor(1, ax);" instruction since it only affects the L.O. bit.

As it turns out, using zero for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: you cannot use the 80x86 AND, OR, XOR, and NOT instructions to implement the boolean operations of the same name. Consider the two values $55 and $AA. They're both non-zero so they both represent the value true. However, if you logically AND $55 and $AA together using the 80x86 AND instruction, the result is zero. True AND true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing boolean operations.

A system that uses non-zero values to represent true and zero to represent false is an *arithmetic logical system*. A system that uses the two distinct values like zero and one to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression:

```
B := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
```

The simple expressions resulting from this expression might be:

```
        mov( x, eax );
        cmp( y, eax );
```



```
sete( al );       // AL := x = y;

mov( a, ebx );
cmp( ebx, d );
setle( bl );     // BL := a <= d;
and( al, bl );   // BL := (x=y) and (a <= d);

mov( z, eax );
sub( a, eax );
cmp( eax, 5 );
setne( al );
or( bl, al );     // AL := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
mov( al, b );
```

When working with boolean expressions don't forget the that you might be able to optimize your code by simplifying those boolean expressions. You can use algebraic transformations (especially DeMorgan's theorems) to help reduce the complexity of an expression. In the chapter on low-level control structures you'll also see how to use control flow to calculate a boolean result. This is generally quite a bit more efficient than using *complete boolean evaluation* as the examples in this section teach.

10.5 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well-known and commonly found in assembly language programs. Some of them can be really tricky, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

10.5.1 Multiplying without MUL, IMUL, or INTMUL

If you take a quick look at the timing for the multiply instruction, you'll notice that the execution time for this instruction is often long[1]. When multiplying by a constant, you can sometimes avoid the performance penalty of the MUL, IMUL, and INTMUL instructions by using shifts, additions, and subtractions to perform the multiplication.

Remember, a SHL instruction computes the same result as multiplying the specified operand by two. Shifting to the left two bit positions multiplies the operand by four. Shifting to the left three bit positions multiplies the operand by eight. In general, shifting an operand to the left n bits multiplies it by 2^n. Any value can be multiplied by some constant using a series of shifts and adds or shifts and subtractions. For example, to multiply the AX register by ten, you need only multiply it by eight and then add in two times the original value. That is, 10*AX = 8*AX + 2*AX. The code to accomplish this is

```
shl( 1, ax );     // Multiply AX by two.
mov( ax, bx);     // Save 2*AX for later.
shl( 2, ax );     // Multiply ax by eight (*4 really, but it contains *2).
add( bx, ax );    // Add in AX*2 to AX*8 to get AX*10.
```

The AX register (or just about any register, for that matter) can often be multiplied by many constant values much faster using SHL than by using the MUL instruction. This may seem hard to believe since it only takes one instruction to compute this product:

1. Actually, this is specific to a given processor. Some processors execute the INTMUL instruction fairly fast.


```
intmul( 10, ax );
```

However, if you look at the timings, the shift and add example above requires fewer clock cycles on many processors in the 80x86 family than the MUL instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it. Of course, on the later 80x86 processors, the multiply instructions are quite a bit faster than the earlier processors, but the shift and add scheme is often faster on these processors as well.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by seven:

```
mov( eax, ebx );      // Save EAX * 1
shl( 3, eax );        // EAX = EAX * 8
sub( ebx, eax );      // EAX*8 - EAX*1 is EAX*7
```

This follows directly from the fact that EAX*7 = (EAX*8)-EAX.

A common error made by beginning assembly language students is subtracting or adding one or two rather than EAX*1 or EAX*2. The following does *not* compute eax*7:

```
shl( 3, eax );
sub( 1, eax );
```

It computes (8*EAX)-1, something entirely different (unless, of course, EAX = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

You can also use the LEA instruction to compute certain products. The trick is to use the scaled index addressing modes. The following examples demonstrate some simple cases:

```
lea( eax, [ecx][ecx] );       // EAX := ECX * 2
lea( eax, [eax]eax*2] );      // EAX := EAX * 3
lea( eax, [eax*4] );          // EAX := EAX * 4
lea( eax, [ebx][ebx*4] );     // EAX := EBX * 5
lea( eax, [eax*8] );          // EAX := EAX * 8
lea( eax, [edx][edx*8] );     // EAX := EDX * 9
```

10.5.2 Division Without DIV or IDIV

Much as the SHL instruction can be used for simulating a multiplication by some power of two, you may use the SHR and SAR instructions to simulate a division by a power of two. Unfortunately, you cannot use shifts, additions, and subtractions to perform a division by an arbitrary constant as easily as you can use these instructions to perform a multiplication operation.

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. Since the multiply instruction is faster than the divide instruction; multiplying by a reciprocal is usually faster than division.

Now you're probably wondering "how does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by one tenth, there is no way you can load the value $1/10^{th}$ into an 80x86 register prior to performing the multiplication. However, we could multiply $1/10^{th}$ by 10, perform the multiplication, and then divide the result by ten to get the final result. Of course, this wouldn't buy you anything at all, in fact it would make things worse since you're now doing a multiplication by ten as well as a division by ten. However, suppose you multiply 1/10th by 65,536 (6553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides AX by ten:

```
mov( 6554, dx );      // 6554 = round( 65,536/10 ).
mul( dx, ax );
```

This code leaves AX/10 in the DX register.

To understand how this works, consider what happens when you multiply AX by 65,536 ($10000). This simply moves AX into DX and sets AX to zero (a multiply by $10000 is equivalent to a shift left by sixteen bits). Multiplying by 6,554 (65,536 divided by ten) puts AX divided by ten into the DX register. Since MUL is faster than DIV , this technique runs a little faster than using a straight division.

Multiplying by the reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal only pays off if you perform the division many, many times (by the same value).

10.5.3 Implementing Modulo-N Counters with AND

If you want to implement a counter variable that counts up to 2^n-1 and then resets to zero, simply using the following code:

```
inc( CounterVar );
and( nBits, CounterVar );
```

where *nBits* is a binary value containing n one bits right justified in the number. For example, to create a counter that cycles between zero and fifteen, you could use the following:

```
inc( CounterVar );
and( %00001111, CounterVar );
```

10.5.4 Careless Use of Machine Idioms

One problem with using machine idioms is that the machines change over time. The DOS/16-bit version of this text recommends the use of several machine idioms in addition to those this chapter presents. Unfortunately, as time passed Intel improved the processor and tricks that used to provide a performance benefit are actually slower on the newer processors. Therefore, you should be careful about employing common “tricks” you pick up; they may not actually improve the code.

10.6 The HLA (Pseudo) Random Number Unit

The HLA rand.hhf module provides a set of pseudo-random generators that returns seemingly random values on each call. These pseudo-random number generator functions are great for writing games and other simulations that require a sequence of values that the user can not easily guess. These functions return a 32-bit value in the EAX register. You can treat the result as a signed or unsigned value as appropriate for your application.

The rand.hhf library module includes the following functions:

```
procedure rand.random; returns( “eax” );
procedure rand.range( startRange:dword; endRange:dword ); returns( “eax” );

procedure rand.uniform; returns( “eax” );
procedure rand.urange( startRange:dword; endRange:dword ); returns( “eax” );

procedure rand.randomize;
```

The *rand.random* and *rand.uniform* procedures are both functions that return a 32-bit pseudo-random number in the EAX register. They differ only in the algorithm they use to compute the random number sequence (*rand.random* uses a standard linear congruential generator, *rand.uniform* uses an additive generator. See Knuth’s “The Art of Computer Programming” Volume Two for details on these two algorithms).

The *rand.range* and *rand.urange* functions return a pseudo-random number that falls between two values passed as parameters (inclusive). These routines use better algorithms than the typical “mod the result

by the range of values and add the starting value" algorithm that naive users often employ to limit random numbers to a specific range (that naive algorithm generally produces a stream of numbers that is somewhat less than random).

By default, the random number generators in the HLA Standard Library generate the same sequence of numbers every time you run a program. While this may not seem random at all (and statistically, it certainly is not random), this is generally what you want in a random number generator. The numbers should appear to be random but you usually need to be able to generate the same sequence over and over again when testing your program. After all, a defect you encounter with one random sequence may not be apparent when using a different random number sequence. By emitting the same sequence over and over again, your programs become deterministic so you can properly test them across several runs of the program.

Once your program is tested and operational, you might want your random number generator to generate a different sequence every time you run the program. For example, if you write a game and that game uses a pseudo-random sequence to control the action, the end user may detect a pattern and play the game accordingly if the random number generator always returns the same sequence of numbers.

To alleviate this problem, the HLA Standard Library rand module provides the *rand.randomize* procedure. This procedure reads the current date and time (in milliseconds) and, on processors that support it, reads the CPU's timestamp counter to generate an almost random set of bits as the starting random number generator value. Calling the *rand.randomize* procedure at the beginning of your program essentially guarantees that different executions of the program will produce a different sequence of random numbers.

Note that you cannot make the sequence "more random" by calling *rand.randomize* multiple times. In fact, since *rand.randomize* generates a new seed based on the date and time, calling *rand.randomize* multiple times in your program will actually generate a less random sequence (since time is an ever increasing value, not a random value). So make at most one call to *rand.randomize* and leave it up to the random number generators to take it from there.

Note that *rand.randomize* will randomize both the *rand.random* and *rand.uniform* random number generators. You do not need separate calls for the two different generators nor can you randomize one without randomizing the other.

One attribute of a random number generator is "how uniform are the results the generator returns." A uniform random number generator[2] that produces a 32-bit result returns a sequence of values that are evenly distributed throughout the 32-bit range of values. That is, any return result is as equally likely as any other return result. Good random number generators don't tend to bunch numbers up in groups.

The following program code provides a simple test of the random number generators by plotting asterisks at random positions on the screen. This program works by choosing two random numbers, one between zero and 79, the other between zero and 23. Then the program uses the *console.puts* function to print a single asterisk at the (X,Y) coordinate on the screen specified by these two random numbers (therefore, this code runs only under Windows). After 10,000 iterations of this process the program stops and lets you observe the result. **Note**: since random number generators generate random numbers, you should not expect this program to fill the entire screen with asterisks in only 10,000 iterations.

```
program testRandom;
#include( "stdlib.hhf" );

begin testRandom;

    console.cls();
    mov( 10_000, ecx );
    repeat

        // Generate a random X-coordinate
```

2. Despite their names, both *rand.uniform* and *rand.random* generate a uniformly distributed set of pseudo-random numbers.


```
            // using rand.range.

            rand.range( 0, 79 );
            mov( eax, ebx );            // Save the X-coordinate for now.

            // Generate a random Y-coordinate
            // using rand.urange.

            rand.urange( 0, 23 );

            // Print an asterisk at
            // the specified coordinate on the screen.

            console.puts( ax, bx, "*" );

            // Repeat this 10,000 times to get
            // a good distribution of values.

            dec( ecx );

        until( @z );

        // Position the cursor at the bottom of the
        // screen so we can observe the results.

        console.gotoxy( 24, 0 );

    end testRandom;
```

Program 10.1 Screen Plot Test of the HLA Random Number Generators

The rand.hhf module also provides an *iterator* that generates a random sequence of value in the range 0..n-1. However, a discussion of this function must wait until we cover iterators in a later chapter.

10.7 Putting It All Together

This chapter finished the presentation of the integer arithmetic instructions on the 80x86. Then it demonstrated how to convert expressions from a high level language syntax into assembly language. This chapter concluded by teaching you a few assembly language tricks you will commonly find in programs. By the conclusion of this chapter you are (hopefully) in a position where you can easily evaluate arithmetic expressions in your assembly language programs.

