# I. The A-Natural Language

**NAME**

As.80 - Introduction to A-Natural

**FUNCTION**

A-Natural, otherwise known as As.80, is a narrative assembly language for use with the Intel 8080, Intel 8085, or Zilog Z/80 computers. It features a uniform naming convention for registers, a concise operator notation for most machine instructions, and formatting rules that make logical groupings very visible. Nevertheless, the language makes available to the programmer the full power of the underlying instruction set, and is careful not to generate unexpected side effects, such as altering unnamed registers.

The contrast between A-Natural and assembly language can be made by the code sequence for subtracting two 16-bit integers, at memory location X and at 4 + de:

```
a = *(bc = &X) - *(hl = 4 + de) -> *bc = *(bc + 1) -^ *(hl + 1) -> *bc
```

or, in assembler:

```
LXI B,X        INX B
LDAX B         LDAX B
LXI H,4        INX H
DAD D          SBB M
SUB M          STAX B
STAX B
```

Note that register a is explicitly named, that all references to hl use the same name (not H, M or implicit), that all data moves are called out with the assigning operators "=" and "->", and that a few operators are used to express addressing modes ("*", "&") and arithmetic ("-", "-^").

To read a line of A-Natural requires the repeated application of a few rules:

1) Reading left to right, locate the first binary operator, in this case "=".

2) If the operand to the left is not a simple term, evaluate the subexpression. In this case, the operand is just a.

3) If the operand to the right is not a simple term, evaluate the subexpression. Here *(bc = &X) requires code production for bc = &X, which is "LXI B,X", and then reduces to *bc.

4) Produce code for the binary operator using the two reduced operands, here a = *bc.

5) Keep the left operand, discard the rest, and repeat. The line then becomes

```
a - *(hl = 4 + de) -> *bc = *(bc + 1) -^ *(hl + 1) -> *bc
```

Under this scheme, all binary operators have the same precedence, so expressions read strictly left to right, except where order is explicitly modified by parentheses.

The use of A-Natural is described by the following subsections:

**As.80** - you're looking at it.

**Syntax** - how to form identifiers, constants, expressions, etc.

**Defining** - giving values to symbols and expressions, generating constants.

**Moves** - the 8080 registers and basic data moves.

**Arithmetic** - manipulation of bytes and addresses, and fancier moves.

**Control** - flow of control and process control.

**Techniques** - how to deal with integers and addresses, how to mix with C code.

**Summary** - the language in a nutshell.

**EXAMPLE**

Here is the complete text of a library module in A-Natural:

```
/   imul
/   copyright (c) 1979 by Whitesmiths, Ltd.
/   mul int into int
/   stack: P, bc, hl, pc, R, L
/
/   P = 0
/   while (R)
/       if (R & 1)
/           P =+ L
/       L =<< 1
/       R =>> 1
/   L = P

    public  c.imul
R := 8 L := 10 c.imul:
    sp <= hl <= bc <= (hl = 0)
.1:
    a = *(hl = R[1] + sp) | *(hl - 1) jz .3
    a = *hl <^> -1 jnc .2
    bc =^ (hl = L + sp)
    hl <= sp + bc => sp
.2:
    a = *(hl = L + sp) + a -> *hl = *(hl + 1) +^ a -> *hl
    a = *(hl = R[1] + sp) <-> -1 -> *hl
    a = *(hl - 1) <^> -1 -> *hl
    jmp .1
.3:
```

```
bc <= sp ->^ (hl = L[-2] + sp)
jmp c.lret
```

**NAME**

Syntax - rules for writing A-Natural

**FUNCTION**

At the lowest level, an A-Natural program is represented as a text file, consisting of lines each terminated by a semicolon ';', a newline '\n', or a comment. A comment consists of all characters between a '/' and the next newline, inclusive.

Text lines are broken into tokens, strings of characters possibly separated by whitespace. Whitespace consists of one or more non-printing characters, such as blank or tab; its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms:

**An identifier** - consists of a letter, dot '.', or underscore '_', followed by zero or more letters, dots, underscores, or digits. Uppercase letters are distinct from lowercase letters; no more than nine characters are significant in comparing identifiers. More severe restrictions may be placed on identifiers by the world outside the translator. There are also a number of identifiers reserved as key-words:

| | | | | | |
|---|---|---|---|---|---|
| . | cm | de | jmp | public | rst1 |
| .data | cmc | di | jnc | rc | rst2 |
| .text | cnc | e | jnz | ret | rst3 |
| a | cnz | ei | jp | rm | rst4 |
| af | cp | h | jpe | rnc | rst5 |
| b | cpe | hl | jpo | rnz | rst6 |
| bc | cpo | hlt | jz | rp | rst7 |
| c | cz | in | l | rpe | rz |
| call | d | jc | nop | rpo | sp |
| cc | daa | jm | out | rst0 | stc |

**A numeric constant** - consists of a decimal digit, followed by zero or more letters and digits. If the leading characters are "0x" or "0X", the constant is a hexadecimal literal and may contain the letters 'a' through 'f', in either case, to represent the digit values 10 through 15, respectively. Otherwise a leading '0' implies an octal literal, which nevertheless may contain the digits '8' and '9'. A nonzero leading digit implies a decimal literal. Numbers are represented to 16 bits of precision; overflow is not diagnosed.

**A character constant** - consists of a single quote, followed by zero or more character literals, followed by a second single quote. A character literal consists of a) any character except '\', newline, or single quote, the value being the ASCII representation of that character; b) a '\' followed by any character except newline, the value being the ASCII representation of that character, except that characters in the sequence <'b', 't', 'v', 'f', 'n', 'r', '(', '!', ')', '^'> in either case have the ASCII values for the corresponding members of the sequence <backspace, horizontal tab, vertical tab, form feed, newline, carriage return, '{', '|', '}', '~'>; or c) a '\' followed by one to three decimal digits, the value being the octal number represented by those digits. A newline is never permitted inside quotes. The value of the constant is an integer base 256, whose digits are the character literal values. Only the last two such digits are retained. Overflow is not diagnosed.

**A string constant** - is just like a character constant, except that double quotes '"' are used to delimit the string. A string is shorthand for a series of byte constants, the values being the character literals in the string.

**Punctuation** - consists of predefined strings of one to three characters. The complete set of punctuation for A-Natural is:

```
!        +        -^       <+>      =        [
&        +^       :        <->      =!       ]
(        -        ::       <=       =>       ^
)        ->       :=       <>       =a^      |
#        ->^      <#>      <^>      =^       ,
```

The longest possible punctuation string is matched, so that "=!", for example, is recognized as "=!" always, and never as '=' followed by '!'. The operators "!" and "|" are equivalent.

**Other characters** - such as '@' or '\' alone, are illegal outside of character or string constants, and are diagnosed.

Nothing else is a token in A-Natural.

For those who love Backus-Naur Form, A-Natural can be succinctly expressed by the productions:

```
program :=  <NULL>
         |  program line
line    :=  <NULL> EOL
         |  term EOL
         |  STRING EOL
         |  expr EOL
         |  IDENT : line
expr    :=  term BINOP term
         |  expr BINOP term
term    :=  NUMBER
         |  IDENT
         |  UNOP term
         |  IDENT : term
         |  IDENT := expr
         |  ( expr )
         |  term [ expr ]
```

where the following definitions obtain:

| | |
|---|---|
| <NULL> | is nothing |
| EOL | is a line terminator, e.g. ';' or '\n' |
| STRING | is a string constant |
| BINOP | is one of several binary operators to be described |
| NUMBER | is a numeric constant or a character constant |

| | |
|---|---|
| IDENT | is an identifier |
| UNOP | is one of several unary operators to be described |

In plain English, this says that a program is a concatenation of zero or more lines, each of which may be empty, or contain a term, or contain a string constant, or contain an expression. A line may be preceded by zero or more labels, where a label is an identifier followed by a colon.

An expression is a sequence of two or more terms, separated by binary operators. Expressions read left to right. A term, at its simplest, consists of just a numeric constant, a character constant, or an identifier. It may also consist, however, of a definition, which is an identifier followed by ":=" and an expression, or it may consist of an expression inside parentheses. It may be preceded by zero or more labels and/or unary operators; it may be followed by zero or more subscripts, each enclosed in brackets.

This seeming generality is circumscribed by a few important semantic restrictions. These restrictions are discussed as each of the interesting productions are visited in more detail.

**BUGS**

m should be a reserved identifier.

**NAME**

Defining - defining symbols, data, and code

**FUNCTION**

The goal of A-Natural is to produce a module, of assembler statements or relocatable binary, that defines certain globally known symbols, that specifies the initial content of data areas, and that expresses the sequences of instructions that form each function body. An important subgoal is to define the value and attributes of terms and internally used symbols.

Symbols and terms stand for data items to be manipulated; each can be viewed as a recipe for locating the data in question. There are a number of predefined identifiers for talking about registers, while numerical values are represented by numeric and character constants, as well as strings. All memory addresses and references derive from a few sources.

There is a predefined symbol .text which stands for the next location to generate code into in the program text (read only) section of memory. The symbol .data stands for the next location to generate code into in the data (writeable) section of memory. A third symbol, written simply ., is the next location that A-Natural will generate code into, either .text or .data.

An as yet undefined identifier may appear in a defining statement of the form:

```
    x := 3
```

Henceforth, any reference to x will be taken as a reference to the literal number 3, a one-byte constant. The right side of the defining operator ":=" must be an expression that evaluates to a number, a memory address, or a memory reference.

The usual source of memory references is ., as in:

```
    M := .
```

This defines M as the contents of the next byte (or two bytes) of memory. So often does this occur that a special label notation is provided:

```
    M:
```

has the same effect, of defining M as ., but is shorter to write and may be followed by additional labels or by a term.

If it is necessary to talk about a memory address proper, instead of the contents of memory, the "address of" operator &M converts a memory reference M to a two-byte constant whose value is the address of M. Similarly, &3 converts the one-byte constant to two-bytes, while &&3 leaves &3 unchanged.

To convert a two-byte constant to a memory reference, the unary "indirect on" operator "*" is used. Thus, *&M is the same as M and *&3 is the memory location at absolute address 3. The "*" operator is also used in conjunction with certain two-byte registers to specify register indirect reference.

Other unary operators are "+", "-", and "!", which modify the value of one or two-byte absolute constants. The value is left alone by "+", negated by "-", and ones complemented by "!". The operator public makes its operand, which must be a symbol, known across all files at load time:

```
    public x
```

This term does nothing but declare x to be globally known.

The value of a term may be modified by a constant offset by using the subscript notation M[5]. This term refers to the location five bytes up from M, while 3[5] is the one-byte constant 8, and &3[5] is the two-byte absolute address 8.

Code generation uses this notation plus a few additional concepts. A one-byte constant standing alone generates one byte of code having that value:

```
    N: 0
```

Byte location N is initialized to zero.

To generate a two-byte constant:

```
    list: &0
```

or:

```
    list: &head
```

where head is presumably another label. By special dispensation, the & can be omitted in this context.

Strings may be used to specify a sequence of bytes. For instance:

```
    "help\n\0"
```

is the same as

```
    'h'; 'e'; 'l'; 'p'; '\n'; 0
```

only much easier to type.

Concatenation is the only operation permitted on strings. The comma operator permits one-byte constants, two-byte absolute constants, and strings to be concatenated to form longer strings, as in:

```
    "help", "\n", 0
    'h', 'e', "lp\n\0"
```

both of which are equivalent to the string above.

One last unary operator is a peculiar one called "literal". The term =&M causes a two-byte literal whose value is the address of M to be memorized in a table. At the end of source input, code generation reverts to .text and the table is emitted; identical literals are merged as much as possible. The value of the term =&M is a memory reference to that literal to be emitted. This permits relocatable addresses to be manipulated a byte at a time by references to (=&M)[0] and (=&M)[1].

String literals may also be generated, by writing lines like

```
    hl = &="help\n\0" => sp
```

which is a convenient way of defining a literal string and pushing its address on the stack all at once. As another example, a long constant can be written either as a string of four bytes or two two-byte constants, as:

```
    hl = &=(&-1, &-4) => sp
```

to get a pointer to a long -4 on the stack.

To switch from generating code into .text (which A-Natural always starts out doing) to generating data into .data:

```
    . := .data
```

and to switch back:

```
    . := .text
```

and to reserve space, as for an array:

```
    . := .[50] / reserve 50 bytes
```

Note that . is the only identifier that may be redefined, that its redefinition must be relative to .text or .data, and that . cannot be backed up over code already produced.

All other code generation is specified by binary operators between terms, as described in the As.80 subsection of this manual. Thus,

```
    X op Y op Z
```

is shorthand for

```
    X op Y; X op Z      / X a simple term
```

and leads to code production for 1) the term X, 2) the term Y, 3) X op Y, 4) the term Z, and 5) X op Z.

Later sections of this manual describe the defined binary operators, the combinations of terms they accept, and the actions performed.

**NAME**

Moves - how to move data about

**FUNCTION**

The Intel 8080 is a computer that deals in 8-bit bytes and 16-bit addresses (words) using twos-complement arithmetic. All instruction and data bytes are fetched from the same 65,536 byte address space, which may be a mixture of ROM (for code and tables) and RAM (for stack and data). Words are stored less significant byte first. Input/output is a byte at a time from a separate 256-port address space.

Seven byte registers can participate in byte operations. These are called a (which frequently serves as a byte accumulator), b, c, d, e, h and l. Another byte register, called f, holds flags that are set or tested as a side effect of some instructions. These flags are, in order of descending significance:

**f7 (S)** - is made to match the sign of the result of most byte arithmetic instructions.

**f6 (Z)** - is set if the result is zero for most byte arithmetic instructions.

**f5 (0)** - is always 0.

**f4 (AC)** - is set on a carry from bit 3 to bit 4 on most byte arithmetic instructions.

**f3 (0)** - is always 0.

**f2 (P)** - is set if the result has an even number of one bits on most byte arithemtic instructions. [N.B. This flag doubles as an overflow indicator on the Z/80, the only known flaw in its upward compatibility with the 8080.]

**f1 (1)** - is always 1.

**f0 (C)** - is set on a carry from bit 7 of many byte arithemtic instructions, is set on a carry from bit 15 on word adds, participates in byte rotations, and can be explicitly set or complemented. It is usually called the carry flag.

Flag settings will be discussed with each instruction that uses them.

These byte registers are combined in pairs to form four word registers, called af, bc, de, and hl. In each case, the first register named holds the more significant byte. hl is most heavily used as a pointer accumulator; af can only be pushed or popped.

Another word register is sp, which is dedicated as a stack pointer. Only word quantities may be moved on or off the stack, although there is no requirement that sp contain an even address. A quantity is pushed by decrementing sp by two, then using sp as the storage address of the less significant byte; popping first uses sp as the source address, then adds two to sp. Thus sp always points at the last word pushed on the stack, i.e., the top of stack.

The final word register is pc, which is dedicated as the program counter. At the start of each instruction pc points to the opcode byte, which may be followed by a one-byte immediate operand or a two-byte address. A special class of instructions is used to modify pc, and hence alter flow of control; these are called jumps, calls, returns, and resets. pc is not a reserved identifier in A-Natural.

Any byte register may be copied to another, as in:

```
c = b
```

or equivalently,

```
b -> c
```

which copies b into c. Moreover, any operation that is permissible for b, c, d, e, h, and l is also permissible for the byte pointed at by hl, i.e., *hl. Thus,

```
a = *hl -> c
```

calls for a to be loaded with a copy of the byte pointed at by hl, then copied into c. No flags are affected by these byte moves.

Note that the eight special cases

```
b = b; c = c; ...; *hl = *hl
```

should be avoided, because the instruction either does nothing or halts the computer (*hl = *hl). A one-byte no operation is best coded as

```
nop
```

while a halt is best written

```
hlt
```

The identifiers nop and hlt are simply defined as small numbers having the proper secret values; since stating a number alone causes a byte with that value to be generated inline, the proper instructions are produced.

It is possible to assign a byte constant to any byte register, or *hl, by writing

```
d = 13
*hl = 0200
```

The first line copies a decimal 13 into d, the second an octal 200 into the byte pointed at by hl.

Register a can participate in several other moves:

```
    a = *bc; a -> *bc
    a = *de; a -> *de
    a = M; a -> M
```

Here, M stands for an arbitrary memory address, of the byte to be loaded or stored.

Word values may be assigned to most of the word registers:

```
    bc = &1024
    de = &M
    hl = 3
    sp = &stack
```

The third line is simply shorthand for

```
    hl = &3
```

while the fourth is very rarely used, since sp is usually setup only once per program. A much more frequently used assignment is:

```
    sp = hl
```

after computing a modified version of sp in hl. Note that af may not be assigned to; assigning to pc occurs in the guise of jumps, calls, etc.

It is also permissible to load and store hl directly from memory:

```
    hl = X -> Y
```

This copies the word in memory at X to memory at Y, using hl, and should not be confused with the loading of an immediate value as shown above.

Finally, it is possible to copy words arbitrarily among bc, de, and hl, by writing lines like:

```
    bc = de
    hl = bc
```

These operations are actually performed a byte at a time, and so may be viewed either as two byte instructions or as shorthand for lines like:

```
    c = e; b = d
    l = c; h = b
```

The difference is rarely significant.

**NAME**

Arithmetic - manipulating data on the 8080

**FUNCTION**

The 8080 manipulates data as eight-bit bytes and, to a much lesser extent, as 16-bit words. Arithmetic in the usual sense of the term is restricted to addition and subtraction; there are no multiply or divide instructions. But there are also a number of logical operations, as well as rotates of various flavors, that make possible multiple precision computations and more elaborate operations. Hence all this stuff is lumped together.

It is easy to add or subtract one from any byte register or *hl:

```
c + 1
*hl - 1
```

These operations set all but the carry flag. On the other hand, the word increment and decrement operations:

```
bc + 1, de + 1, hl + 1, sp + 1
bc - 1, de - 1, hl - 1, sp - 1
```

affect no flags at all. Incrementing or decrementing sp is rarely useful, by the way.

Any of the byte registers, *hl, or a byte constant may be added to a:

```
a + c + 060
```

Each addition affects all flags. Word addition may only occur from bc, de, or sp to hl:

```
hl = 4 + sp -> bc
```

This is the best way to get sp + 4 into bc, using hl as a word accumulator. Only the carry flag is affected.

All other arithmetic in the 8080 involves a byte operand that is in a register, at *hl, or constant and a second byte operand in a, with the result left in a and all flags affected. The complete set of operations, including the addition already discussed, is:

```
a + a       / add
a +^ *hl    / add with carry
a - 3       / subtract
a -^ b      / subtract with borrow
a & h       / bitwise and
a ^ l       / bitwise exclusive or
a | a       / bitwise inclusive or
a :: e      / compare
```

Adding a to a effectively doubles it, or shifts it left one place, at the same time moving the previous sign bit a7 into the carry flag.

Add with carry is the same as add, except that, if the carry flag is set at the start of the instruction, a one is added to a along with the right operand.

Subtract adds the twos complement of the right operand to a, setting the carry flag if a "borrow" must be propagated to the next more significant byte. Subtract with borrow continues this chain, much like add with carry does for addition.

The bitwise operators follow the standard truth tables:

| X Y | X & Y | X ^ Y | X \| Y |
|---|---|---|---|
| 0 0 | 0 | 0 | 0 |
| 0 1 | 0 | 1 | 1 |
| 1 1 | 1 | 0 | 1 |
| 1 0 | 0 | 1 | 1 |

These operations unconditionally clear the carry and AC flags.

Note that a | a or a & a leaves a unchanged, but sets the S and Z flags to reflect its value. Compare also leaves a unchanged, setting the flags to reflect the effect of subtracting the right operand from a.

The complement operator

```
    a =! a
```

replaces a with its ones complement, affecting no flags, while

```
    daa
```

jiggers up the a register after an add to give the correct result for adding decimal numbers packed as two four-bit digits per word. This is the only instruction that makes sensible use of the AC flag; it sets all flags so that the decimal sum may be continued to more significant bytes.

The a register may be rotated one place left or right with or without the carry flag being interposed between a7 and a0:

```
    a <^> 1     / rotate left with carry
    a <^> -1    / rotate right with carry
    a <*> 1     / left without carry
    a <*> -1    / right without carry
```

To manipulate the carry flag directly:

```
    stc      / sets carry
    cmc      / complements carry
    a | a    / clears carry (but also affects other flags)
```

Two other operators perform rotates after forcing carry to a known state:

```
    a <+> 1     / set carry, rotate left
    a <+> -1    / set carry, rotate right
    a <-> 1     / clear carry via a|a, rotate left
    a <-> -1    / clear carry via a|a, rotate right
```

The last is most frequently used to effect a right shift with no sign propagation. All rotates affect the carry flag; if it does not participate in the shift, it is still set as if it did. Note that the "<->" operator curdles the flags, as a result of the a|a.

The 8080 has some peculiar instructions for more specialized moves, such as stack pushing and popping, and exchanges. Combined with some simple arithmetic sequences, these give an additional set of fancier data moves.

To push a word register onto the stack:

```
    sp <= bc    / or de, or hl, or af
    bc => sp    / or de, or hl, or af
```

Either form has the same effect. Popping is the reverse notation:

```
    sp => bc    / or de, or hl, or af
    bc <= sp
```

It is also possible to exchange hl with the top stack word:

```
    hl <> *sp
```

or:

```
    *sp <> hl
```

Note that it is *sp that is swapped with hl, not sp. The only other exchange is

```
    hl <> de
```

which may also be written either way around.

The only way to load a word register from a computed address is to put the address in hl and pick up a byte at a time, as in:

```
    c = *hl
    hl + 1
    b = *hl
```

This occurs so often that a special notation is used:

```
    bc =^ hl
```

where the ^ is a reminder that hl is upped by one as a side effect. Legal variations are:

```
bc =^ hl, de =^ hl
bc ->^ hl, de ->^ hl
```

The latter two cause word registers to be stored using hl as a memory pointer.

Finally, the oft needed sequence:

```
a = *hl
hl + 1
h = *hl
l = a
```

which replaces hl with the word it points at, using a, can be written:

```
hl =a^ hl
```

Again, the a and ^ warn that a is clobbered, and hl incremented.

**BUGS**

The only way to add one and set the carry flag is by the ruse:

```
a + 0401
```

**NAME**

Control - jumps and process control

**FUNCTION**

Control takes two forms in a computer, directing the sequence of operations within a given process, and directing the interaction among processes and their environment. The former is effected by conditional jumps, subroutine calls, and returns; the latter is usually implemented as a hodgepodge of peculiar instructions. In both cases, the 8080 is most typical.

An unconditional transfer of control is called a jump, which can be written as a unary operator:

```
jmp X
```

which moves the address X into pc (jumps to X), or as a binary operator:

```
a = 0 jmp M
```

The latter sequence clears a, then jumps to M.

There are also eight conditional jumps, that jump only if the appropriate flag state obtains. There are:

| OPERATOR | JUMPS IF |
|---|---|
| jnz X | Z == 0 |
| jz X | Z == 1 |
| jnc X | C == 0 |
| jc X | C == 1 |
| jpo X | P == 0 |
| jpe X | P == 1 |
| jp X | S == 0 |
| jm X | S == 1 |

To transfer to a subroutine, the call instruction

```
call X
```

pushes pc on the stack, then jumps to X. This permits a return with

```
ret
```

to get back to the calling sequence by popping pc from the stack. Calls, like jumps, are treated as operators, while returns are numbers.

The 8080 also has a full complement of conditional calls and returns, although they are rarely used:

| | |
|---|---|
| cnz X | rnz |
| cz X | rz |
| cnc X | rnc |
| cc X | rc |
| cpo X | rpo |

```
cpe X   rpe
cp X    rp
cm X    rm
```

Finally, it is possible to jump to a computed address with

```
jmp *hl
```

which copies hl into pc.

A more specialized way to call a function is with a restart instruction:

```
rst0; rst1; ...; rst7
```

The general form rstn pushes pc on the stack, then jumps to absolute location 8*n, i.e., to location 0, 8, 16, 24, etc. Restarts are usually reserved for communicating with system functions. In fact, when an interrupt occurs, the 8080 usually behaves as if an rstn instruction were injected into the code sequence, with n supplied by the interrupting hardware.

Interrupts can be enabled (allowed to happen) by executing

```
ei
```

and disabled with

```
di
```

If interrupts were disabled and ei is encountered, the instruction following ei is executed before interrupts are actually enabled.

Up to 256 logical ports may be addressed by the 8080 for performing byte at a time input/output. There is no requirement that a port be actually present, or that an output port be associated in any way with an input port having the same number.

To read data:

```
in; n
```

copies whatever input port n is sending into the a register, while

```
out; n
```

presents the contents of a and the port address n to the outside world.

**NAME**

Techniques - 8080 for grownups

**FUNCTION**

Just knowing the instruction set of the Intel 8080 is not enough to program it properly because a) the set is primitive, b) it is incomplete, and hence c) the most commonplace of operations are often nontrivial and unobvious. This section describes a number of techniques for coding maturely in such an immature environment.

With rare exception, byte (or any other) operations require the use of a as an accumulator; hence af should be viewed as extremely volatile. Returning byte values in a, for instance, is often suboptimal because a is frequently needed just to get back from a function! More important is to know:

1) What can be loaded into a - {b, c, d, e, h, l, *hl, #, *bc, *de, M}

2) What can be added to a - {b, c, d, e, h, l, *hl, #}

where # is any one-byte constant and M is a memory reference. Fortunately, whatever can be loaded into a can also be stored from a, except of course a constant; and what can be added to a can also be subtracted, anded, etc.

Thus, the most general technique for performing an operation <op> between bytes is:

1) Set up destination address in bc

2) Set up source address in hl

3) Write "a = *bc <op> *hl -> *bc".

Naturally, steps (1) or (2) can be omitted if the operand happens to be already within reach.

To get operands in reach requires address arithmetic, which usually involves hl as a pointer accumulator. If an operand is on the stack,

```
    hl = D + sp
```

allows the data at displacement D off of sp (also written D(sp)) to be addressed as *hl. The C language uses de as a "frame pointer", so as not to have to keep close tabs on the state of sp. Thus, arguments and automatic variables are addressed as D(de), where D runs from 4 up for arguments and from -7 down for autos. The sequence

```
    bc = (hl = 4 + de)
```

clearly loads bc with the address of the first argument.

Often it is necessary to trace down a chain of pointers to get to the operand. This is known variously as indirect addressing or dereferencing.

For this, the two complex moves "=^" and "=a^" frequently pay off. If, for instance, the first argument is a pointer to the operand of interest:

```
    bc =^ (hl = 4 + de)     / operand now at *bc
```

or:

```
    hl =a^ (hl = 4 + de)    / operand now at *hl
```

Further modification, such as adding a constant offset into an array or structure, usually requires both bc and hl:

```
    hl = 6 + bc -> bc   / for bc + 6
```

or:

```
    hl + (bc = 4)       / for hl + 4
```

If the offset to be added is smaller than those shown, however, it is shorter and faster to write:

```
    bc + 1 + 1 + 1      / for bc + 3
```

or:

```
    hl + 1 + 1 + 1      / for hl + 3
```

and no other registers are needed.

Testing invariably requires that the appropriate flag in f be set. For signed arithmetic, there are six possibilities:

| TO JUMP IF | CODE |
|---|---|
| a < b | a :: b jm X |
| a >= b | a :: b jp X |
| a == b | a :: b jz X |
| a != b | a :: b jnz X |
| a > b | b :: a jm X |
| a <= b | b :: a jp X |

Note that all but the two equality comparisons give incorrect results if a and b differ by 128 or more. This is a hard error to avoid, unless the test can be converted to an unsigned comparison, shown below.

To compare the byte at b against zero:

| TO JUMP IF | CODE |
|---|---|
| b < 0 | a = b \| a jm X |
| b >= 0 | a = b \| a jp X |
| b == 0 | a = b \| a jz X |
| b != 0 | a = b \| a jnz X |
| b > 0 | a - a :: b jm X |
| b <= 0 | a - a :: b jp X |

Unsigned tests are the same as signed, except that the carry flag is inspected instead:

| TO JUMP IF | CODE |
|---|---|
| a < b | a :: b jc X |
| a >= b | a :: b jnc X |
| a == b | a :: b jz X |
| a != b | a :: b jnz X |
| a > b | b :: a jc X |
| a <= b | b :: a jnc X |

These tests are correct for all values of a and b. The meaningful zero comparisons for unsigned collapse to equality and inequality, which are the same as above:

| TO JUMP IF | CODE |
|---|---|
| b == 0 | a = b \| a jz X |
| b != 0 | a = b \| a jnz X |

Multiple precision arithmetic, which frequently includes even two-byte calculations, requires judicious stepping of pointer registers and close attention to the carry flag. Fortunately, the designers of the 8080 were careful that these operations should seldom clash. The prototype multiple precision sequence is integer subtract, presented here for *bc - *hl:

```
    a = *bc - *hl -> *bc = *(bc + 1) -^ *(hl + 1) -> *bc
```

This chain can be extended to arbitrary length, or placed in a loop; to do eight bytes, for example:

```
    de => sp
    e = 7
    a = *bc + *hl -> *bc
L:
    a = *(bc + 1) -^ *(hl + 1) -> *bc
    e - 1 jnz L
    de <= sp
```

Clearly, addition can be performed the same way by using the "+^" operator.

Comparing multiple precision operands is analogous to one byte tests:

| TO JUMP IF | CODE |
|---|---|
| *bc < *hl | a = *bc - *hl = *(bc + 1) -^ *(hl + 1) jm X |
| *bc >= *hl | a = *bc - *hl = *(bc + 1) -^ *(hl + 1) jp X |
| *bc == *hl | a = *bc :: *hl jnz T = *(bc + 1) :: *(hl + 1) jz X; T: |
| *bc != *hl | a = *bc :: *hl jnz X = *(bc + 1) :: *(hl + 1) jnz X |
| *bc > *hl | sp <= bc <= hl => bc => hl; / see (*bc < *hl) |
| *bc <= *hl | sp <= bc <= hl => bc => hl; / see (*bc >= *hl) |

Once again, these are incorrect if the arguments differ widely enough to cause overflow (jp/jm), but are always correct for unsigned compares (jnc/jc).

To test the integer at *hl against zero:

```
TO JUMP IF          CODE
 *hl < 0        a = *(hl + 1) | a jm X
 *hl >= 0       a = *(hl + 1) | a jp X
 *hl == 0       a = *hl | *(hl + 1) jz X
 *hl != 0       a = *hl | *(hl + 1) jnz X
 *hl > 0        a - a - *(hl + 1) jm X
 *hl <= 0       a - a - *(hl + 1) jp X
```

All of these operations extend easily to more bytes of precision. Note, however, that C represents longs and doubles in such a way that often only *(hl + 1), or *hl and *(hl + 1) at most, need be inspected to determine the value of the whole.

As a final note, space on the stack must frequently be bought and sold as control blocks are entered and left. In the general case

```
hl = N + sp -> sp
```

is the best way to add a constant N to sp. If N is even and less than 10, however, it is quicker to push or pop a register several times:

```
sp <= af <= af      / reserve 4 bytes
sp => af => af      / pop 4 bytes
```

af is a good candidate for popping junk into, since it is highly volatile.

**NAME**

Summary - A-Natural in brief

**FUNCTION**

Symbols may be defined as one of:

a) one-byte constants (kept to two bytes of precision)

b) two-byte constants, which may be relocatable addresses

c) memory references

Definition occurs by either:

a) X := Y defines X as Y

b) X: defines X as .

Code generation is directed:

a) into the text section initially, for literals at the end, and after ". := .text".

b) into the data section, after ". := .data"

Code generation occurs for:

a) a one-byte term alone, giving a byte of that value

b) a two-byte term alone, giving two bytes of that value

c) a string, giving a byte for each character in the string

d) a space reservation, giving n zero bytes for ". := .[n]"

e) a binary operator X op Y, giving the appropriate instruction bytes

Terms are modified by:

a) unary operator of the form <op> X

b) subscripts of the form X[n], where n is an absolute constant.

A string is either a literal string or the result of the comma concatenation operator. Valid operands for comma are one-byte constants, two-byte absolute constants, and strings.

Unary operators are:

a) "&" to make a two-byte constant

b) "*" to make a memory reference, or to indirect on a register

c) "+" to do nothing to an absolute number

d) "-" to negate an absolute number

e) "!" to ones complement an absolute number

f) "=" to generate a two-byte literal

g) public to make a symbol known across files

Binary operators, and their valid operands, use the following abbreviations:

a) Cr is short for {a, b, c, d, e, h, l, *hl}.

b) Ir is short for {bc, de, hl, sp}

c) N is a one-byte constant.

d) NN is a two-byte constant.

e) M is a memory reference.

f) X = Y is equivalent to Y -> X.

g) X <= Y is equivalent to Y => X.

h) X <> Y is equivalent to Y <> X.

Operations are evaluated left to right, except as modified by parentheses. The left operand is evaluated first and is retained for all subsequent operations. The valid forms are:

| BYTE OPERATIONS | WORD OPERATIONS |
|---|---|
| a = *bc | Ir = N |
| a -> *bc | Ir = NN |
| a = *de | hl = M |
| a -> *de | hl -> M |
| a = M | sp = hl |
| a -> M | bc = de |
| Cr = N | bc = hl |
| Cr = Cr | de = bc |
| Cr + 1 | de = hl |
| a + Cr | hl = bc |
| a + N | hl = de |
| Cr - 1 | bc => sp |
| a - Cr | de => sp |
| a - N | hl => sp |
| a +^ Cr | af => sp |
| a +^ N | bc <= sp |
| a -^ Cr | de <= sp |
| a -^ N | hl <= sp |
| a & Cr | af <= sp |
| a & N | hl <> *sp |

```
        a ^ Cr                  hl <> de
        a ^ N                   hl + Ir
        a | Cr                  Ir + 1
        a | N                   Ir - 1
        a :: Cr                 bc =^ hl
        a :: N                  de =^ hl
        a <^> 1                 bc ->^ hl
        a <^> -1                de ->^ hl
        a <#> 1                 hl =a^ hl
        a <#> -1
        a <+> 1
        a <+> -1
        a <-> 1
        a <-> +1
        a =! a

CONDITIONALS
    jmp X       call X      ret
    jnz X       cnz X       rnz
    jz X        cz X        rz
    jnc X       cnc X       rnc
    jc X        cc X        rc
    jpo X       cpo X       rpo
    jpc X       cpc X       rpc
    jp X        cp X        rp
    jm X        cm X        rm

OTHER PREDEFINED CONSTANTS
    cmc     rst0
    daa     rst1
    di      rst2
    ei      rst3
    hlt     rst4
    in      rst5
    nop     rst6
    out     rst7
    stc
```