An Informal Introduction to HAL70

General Remarks

HAL70 (High level Assembler Language) was designed to offer a convenient means of coding programs for Interdata (and several other similar) machines. This document is an informal guide, a reference manual exists which constitutes the definitive account of the language.

HAL70 makes it possible for the programmer to have explicit and detailed control over the location of instructions and data within the memory while at the same time offering a number of the advantageous features to be found in high level languages, such as expression evaluation, assignment, conditional & repetitive statements.

Tags

A tag (or identifier) is a sequence of letters and digits starting with a letter. The HAL assembler ignores any more than the first 6 characters in a tag, thus: total, accum, p0,p1, totalfrequency are all legal tags, the last being treated as totalf.

There are many tags which are known to the assembler initially, such as opcode mnemonics and register tags.

Register Tags

The 16 general purpose registers are denoted by the predefined tags:-

r0, r1,r2, . . . r13, r14, r15

The programmer may define other tags to denote the registers as described below.

Use of Tags

Tags are used for many purposes in the language, but the 2 most common uses are as the names of values and the names of variables. A variable may be either a register or a memory location and if the latter then it may be either a halfword (16 bit) location or a byte (8 bit) location.

Explicit Tag Definition

Tags to be used as the names of values or registers must be defined (declared?) explicitly before they are used. The form of statement for doing this is illustrated in the following examples:-

```
$define eot=4,  maska=x'F8F8'
$define maskb=maska+3,  dcbl=108,  dcbmask=dcbl&maskb>>2
```

Following such definitions the tags defined may be used in place of the associated values e.g.

```
r0 = r1 ! maskb
```


To give names, other than the predefined ones, to registers the define statement contains defintions of the following form:-

```
$define index=r4,  acc=r5,  char=r6
```

In any define statement the keyword $define may be abbreviated to $def:-

```
$def work=char,  pointer=r14
```

Following these statements the tags may be used as the names of registers (variables) e.g.

```
index=index-2
acc=char<<8+acc-'9'
```

Labels, Forward References and Implicit Definition

As in conventional assemblers, HAL70 statements may be labelled. This is done by preceding the statement with a tag which is immediately followed by a colon. For example:-

```
        y=y-1
loop:   x=x+1
        - - -
        - - -
count:  0
```

The first label (loop) identifies an instruction, the second (count) identifies a data location.

A tag occurring as a label is (effectively) defined to be a halfword reference to the corresponding memory location.

The following statement subtracts the contents of index (= r4) from the memory location and replaces the result back into that location:-

```
count=count-index
```

Tags defined as labels in this way have the unique property that they may be used before they are defined. For example a program fragment might be:-

```
        count=index-1
        - - -
        - - -
        index = index +2 if count>= '0'
        - - -
        - - -
        - - -
count: 15
```

As a consequence of this freedom, any tag which is used without previous definition is assumed to be a forward reference to some label and is therefore (and thereafter) treated as a halfword memory reference. The assembler will fault any tag which is used in this manner but does not occur somewhere as a label.

Expressions

Much of the expressive power of HAL70 comes from the fact that the assembler can (automatically) generate the code for evaluating expressions. The following are examples of expressions:-

```
char & maska - char
index>>3 + 1<<3
```

The evaluation of such expressions may require many instructions. All arithmetic is performed on 16 bit values.

The operators available are:-

binary		unary	
+ -	add, subtract	-	minus
& !	and, or	\	not (logical complement)
\	exclusive or		
>>, <<	right and left shift		

There is no priority among the operators, they are evaluated strictly from left to right. However brackets may be inserted to override this order of evaluation e.g:-

```
pointer+1<<4    is equivalent to    (pointer+1)<<4

pointer + (1<<4)    is equivalent to    pointer + 16
```

The operands of an expression may be either constants or variables. If the operand is a constant it will be in the form of either a literal (e.g. 308) or a defined tag (e.g. maskb). If the operand is a variable it will be either a register or a reference to a memory location. (There is also a means of specifying an operand to be the address of a memory location rather than a reference to its contents.)

The most general form of memory reference has the syntax:-

```
<tag> ( <expression> )
```

where the tag itself is a memory reference (usually a label). This structure, known as a <term>, forms a reference to a memory location whose address is the sum of the address of the memory location denoted by the tag and the result of evaluating the expression.

For instance, if tab is some label, then terms involving it might be:-

tab(4) the memory location whose address is 4 greater than that of tab.

tab(r6) the memory location whose address is given by the sum of the address of tab and the contents of register 6.

tab(count-2&15) the memory location whose address is the sum of the address of tab and the result of evaluating the expression in brackets.

Temporary Registers

In order to be able to generate the code for evaluating expressions, the assembler frequently needs to be able to store temporary results. For this purpose it uses registers. The registers which it may use are determined by the programmer through a statement of the form:-

```
$temp r2,r5,r6
$temp r1
$temp
```

Following one of these statements the assembler will use any of the registers listed during the evaluation of expressions. The basic assumption is that the programmer will not explicitly use these registers at all. The assembler will do so in any manner that it chooses.

It is frequently sufficient to provide only a single temporary register for use by the assembler. The third example above, with no register list, states that no registers are available for use as temporaries by the assembler.

Conditions

Conditions are formed simply as a relation between (the results of evaluating) 2 expressions e.g:-

```
x=1  if  tab(index) < 15
```

and-conditions and or-conditions can be formed in the familiar way:-

```
x=lett if 'A'<=char and char <='Z'
pointer =pointer+1 if char=' ' or char=nl
```

Explicit tests on the condition code (CC of PSW) may be made by a form of condition which is simply a value. For example:-

```
jump away if 3
x=x-1 if \2
```

If the value is positive it indicates a branch on true condition i.e. perform the imperative if any of the given bits are set within the condition code. If the value is negative it is first complemented (at assembly time) and indicates a branch on false i.e. perform the imperative iff none of the bits in the resulting value are set within the condition code.

Assignment Statements

The fundamental imperative, whose syntax is:-

```
<term> = <expression> <if <condition>>0
```

Note that the lefthand side is formally described by:-

```
<term> -> <tag> ( <expression> )
       -> <tag>
```

Many examples have already been included, but here are some more:-

```
r0 = r0-1
tab(pointer-16) = char&127 + x'8000' if index=0
```

Conditional Statements

These are of one of three related forms:-

```
if <condition>              if <condition>      if <condition>
    [statements]                [statements]        [statements]
else if <condition>         else                finish
    [statements]                [statements]
else if <condition>         finish
    - -
    - -
else
    [statements]
finish
```

In the most general form, the statements following the first true condition are executed, or those following the final else if none of the conditions is true. The final else and/or any/all of the else-if's may be omitted. Examples are:-

```
if char=','                 if char='t'
  readsym                     bal r1,transfer
else if char='/'            else if char='e'
  readsym                     bal r1,edit
  index=4
else if char=nl             else if char='h'
  index=8                     readsym
else                          if char='a'
  bal ret,error                 readsym; char=r1
finish                          bal r1,hal if char='l'
                              finish
                            finish
```

Repetitive Statements

The 2 familiar forms are:-

```
while <condition>           cycle
  [repeated group]            [repeated group]
repeat                      repeat if <condition>
```

The position of the condition indicates that in the while form it is evaluated before entry to the repeated group for the first time, whereas in the second form the repeated group is executed at least once and then repeatedly until the condition becomes false. Examples might be:-

```
while tab(char)=' '
  char=char+1
repeat

cycle
  ss dev,stat
repeat if busy
```


Jump Statements

Labels may be used to reference imperative statements, and control transferred to a labelled imperative by means of a jump statement:-

```
loop: x=x+1
      - -
      - -
      - -
      jump loop if x#0
```

Note that a jump may be made subject to a condition.
An alternative form of statement using the tag "jumps" may be used to force the assembler to use the SI format of jump instruction - it will do so automatically for backward jump instructions.

Data Statements

Locations to be used as variables may be set up by simply placing a value where a statement might go e.g:-

```
      maska, 3, maskb, bcbmask>>1
table: 0, 0, -1, 0, -1, 0, 0, 0
```

The first plants 4 16 bit values in successive locations, the second shows a common structure in which the first location of a table (array?) is labelled.

Data may be placed in successive byte locations by prefixing the data statement with b:-

```
      b  127, 128, 0, 15, x'3F', 2
```

This sets up 6 byte values in 3 adjacent locations. The forms of literal value are hexadecimal (x'FFFF' etc.) and also character (iso) values such as 'a' and '='. A form such as:-
'AaBbCc' is eqivalent to b 'A','a','B','b','C','c'.
Data statements may extend over several lines, if each is terminated by a comma:-

```
b 0, 1, 3, 2, 16, 127, 0,
  1, 1, 5, 127, 127, maska&15+x'100', 14
```

which places 14 values in adjacent byte locations.

If a value is to be replicated within a data statement then the following form may be used:-

```
lab:  8 $ -1                    / which is the same as
lab:  -1,-1,-1,-1,-1,-1,-1,-1   / 8 half word values.
```

The value preceding the dollar ($) is a replication factor.

```
getp: 0
putp: 0
      b 72$0      / reserves 72 byte locations, initialising them to 0.
```

Machine Code Instructions

These may be planted explicitly by statements such as those given below:-

```
mh r2, tab(index)
exbr char,char
clb r0, putp(index+2)
svc 14, x'1ff' if x#0
```

The general form is an opcode mnemonic followed by appropriate operands, seperated by commas. There must be the correct number of operands of an acceptable type, otherwise the statement will be faulted.

Note that a machine code instruction can be made the subject of a condition.

The Assembler Location Counter

The assembler assumes that the code and data, as it is produced, will eventually be loaded into successive memory locations, and creates the object file accordingly. It therefore has its own notional location counter denoting the location into which the next 16 bits of code or data will, ultimately, be placed. This location counter is accessible to the programmer through the predefined "tag" denoted by an asterisk (*). (This is not syntactically a tag but behaves like one to all intents and purposes.) Examples of the use of this pseudo-tag are given in the next section.

The Tags W and B

W and B are pre-defined tags used in creating memory references from arbitrary values.

w(<expression>) is a halfword memory reference to the location whose address is given by evaluating the expression.

b (<expression>) is a byte reference to the location whose address is given by evaluating the expression.

Byte references to memory must all be defined, ultimately, in terms of some b( ). For instance a table of byte locations is defined by statements such as :-

```
$def btab=b(*)
b 0, 1, 2, 3, 'ABcdef'
```

Btab(6) is then a byte reference to a location which contains 'c' initially.

Addresses

The address of a memory reference is obtained by preceding the <term> with a hash (#) sign, e.g:-

```
r12 = #btab(p-16)
```

places the address of this location into register twelve.

Assembler Directives

These are statements which do not generate any code or data, but alter the way in which the assembler continues to operate. One such statement has already been introduced, namely the $define statement. The remainder are briefly described below.

$loc <expression>
: this sets the value of the assembler location counter. E.g.
  $loc *+100
  leaves 100 bytes untouched and continues loading from the new location.

$list <expression>
: Controls the amount of listing information produced. The least significant 4 bits switch on/off various features, namely:-

  1 - if 1 then print all the code or data produced by each statement. If 0 then print only the first line of the same.
  2 - }
  4 - } see reference manual
  8 - flag any jump instruction which could have been made into a short jump (jumps).

  If the <expression> is negative then all listing is suppressed. The current value of the list control can be accessed via the pseudo-tag *list.

$end
: the end of the source text (not strictly necessary).