Principles of descriptors

J. M. Bishop* and D. W. Barron†

Descriptors are a popular feature in the design of new computer architectures but little has been written about the principles and pitfalls involved. Experience with writing and investigating compilers for two structured architectures, the Burroughs B6700 and the ICL2900, has shown that the terms used in relation to descriptors often have contradictory meanings. Descriptors as implemented on these machines are often not the blessing they were made out to be. This paper seeks to enumerate and explain the operational function of descriptors; to define their area of application; to give evidence of the unwieldy aspects in current implementations; and to suggest how these may be improved.

(Received March 1980)

1. The operation of a descriptor

1.1 *Definitions*

There is no widely accepted definition of a descriptor but that of Creech (1970) comes closest to expressing the common view: 'A descriptor is a control word used to locate and describe areas of data or program storage.' This definition also serves to establish what a descriptor is not. It is not just any data word: it is a *control* word and is usually embodied in the architecture of the computer. It is not very easy or profitable to simulate descriptors on a machine which does not have them. The emphasis on control *word* rules out dope vectors as descriptors, although they are sometimes referred to as such (IBM). A descriptor is also more than an indirect address, since its function is to both *locate* and *describe*. A key word is *areas*: a descriptor should not normally be employed in conjunction with simple, primitive items. Its function is clearly associated with aggregates. Finally, the areas may contain *program* or *data*: this implies that descriptors are found not only in user-level compiled code, but also at operating system level, this being where areas of program are manipulated.

Associated with a descriptor is a *descriptor program* which interprets and acts upon its contents. This 'program' is an integral part of the machine's instruction execution hardware (or microprogram) but it is possible to isolate and examine it. The descriptor itself is divided up into *fields*, each of which contributes towards describing or locating the specified area. Terminology here is very confusing and therefore instead of using any particular manufacturer's terms, an 'everyman's' set has been devised for use in this paper.

ORIGIN	The address of the first item in an area
BOUND	The ordinal number of the last item in the area, numbering from OFFSET
DISP	The displacement from the first item in an area to a required item
LENGTH	The ordinal number of the last item of an area, numbering from 0
OFFSET	The ordinal number of the first item in the area
SIZE	An indication of the number of bits in an item (usually a code for one of a small number of options e.g. bit, byte or word)
TYPE	A broad classification of the kind of items in the area, unrelated to size (usually a code for one of a small number of options e.g. code, data or string)

In addition to these fields, the terms *area*, *item* and *subscript* will be used consistently, to avoid confusing the issue with terms such as block, element, index etc. Without preempting what follows, **Fig. 1** provides an illustration of a typical descriptor.

Two further terms need definition: *user's units* are those in which subscripts are expressed (e.g. the items are numbered 1 to 10 in the above example); *machine units* are those in which machine addresses are expressed (e.g. words are numbered 0 to 19 in the above example). The descriptor field ORIGIN will always be in machine units, as will DISP (except in a few variations). OFFSET and BOUND should be in user's units.

1.2 *Layout of a descriptor*

If all of the five fields ORIGIN through to SIZE were potentially able to contain integer values, a descriptor would be well on the way to being six words long. As Creech's definition suggests, this is not permissible and so some means must be found of fitting all the necessary information into one or maybe two words, i.e. between 24 and 64 bits, depending on the machine. As each function of a descriptor is discussed, the optimum field widths and types will be developed. These will be expressed precisely as fields of a Pascal packed record definition. One predefined subrange will be used i.e. the addressing range of the machine or

type *addressrange* = 0. .*maxaddress*

The memory is then regarded as

var *memory*:**array** [*addressrange*] **of** *word*

where word is some packed array of bits.

In the same way, the instructions that activate the descriptor program will be described by Pascal procedures. These procedures can be regarded as the microcode versions of machine instructions. For simplicity's sake, a stack machine is assumed for the illustrative examples. This machine uses the following instructions.

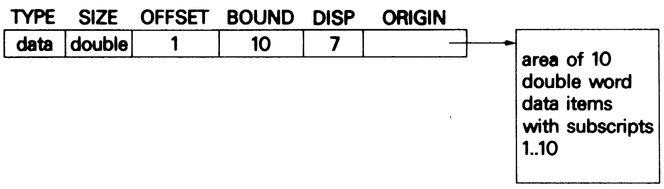


Fig. 1 A typical descriptor

*Computer Science Division, University of the Witwatersrand, 1 Jan Smuts Avenue, Johannesburg 2001, South Africa.
†Mathematics Department, The University, Southampton SO9 5NH, UK.

CCC-0010-4620/81/0024-0210 $06.00

LOAD *x*	Loads the value of *x* on to the stack
NAME *x*	Loads the address (or 'name') of *x* on to the stack
COMP *n*	Compares the value in *tos* to *n* and replaces *tos* by a condition code
JUMP*r* label	If the condition code in *tos* corresponds to the relation *r*, control is transferred to label. *r* may be EQ, GT, NE, LE etc
UP *n*	Adds the constant *n* to *tos*
DOWN *n*	Subtracts the constant *n* from *tos*
SCALE *n*	Multiplies *tos* by the constant *n*
ADD	Adds the top two elements of the stack, replacing them by the sum

The stack is regarded as being prefaced by the two registers *tos* and *tos*2 in which most operations are performed. Notice that the above instructions maintain the distinction between the binary stack operations such as ADD and those involving *tos* and a constant specified as an immediate operand such as UP.

1.3 *The descriptor program*

The operation of a descriptor program can be summed up by the acronym COSA (check, offset, scale and add). This is embodied in the instruction INDEX which takes a descriptor to an area (in *tos*) and a subscript (in *tos*2) and produces an indexed descriptor in their place. INDEX first *checks the subscript* against the offset and bounds and rejects it if it is outside them. It *offsets the subscript* to obtain a subscript relative to zero. This is then *scaled* by the size of the items. The adjusted subscript, now called DISP, combines with the ORIGIN to provide access to the required item. This sequence is exactly that which is produced as inline code for array accesses by many compilers. Consider the difference in length of code and complexity for accessing *a*[*i*] by descriptors:

```
a[i] ⇒ LOAD I
       NAME A
       INDEX
```

and by inline code:

```
a[i] ⇒ LOAD I
       COMP 1
       JUMPLT error     (check i ⩾ 1)
       COMP 10
       JUMPGT error     (check i ⩽ 10)
       DOWN 1           (offset by 1)
       SCALE 2          (scale for double words)
       NAME A           (add on the origin)
       ADD
```

1.4 *Check and offset*

This is the most valuable operation that a descriptor performs, partly because it *does* perform it, whereas checking is often omitted in inline code because of the expense involved. The check takes two forms. First, the type of access is checked against the type of the descriptor and second, the subscript is checked against the stored bounds. The type check can require the following descriptor fields

kind: (*data*, *code*, *string*);
readonly, *presentincore*: *boolean*;

Data descriptors permit access by most data movement operations whereas string descriptors only allow their structures (which contain text) to be accessed by special string operations. A code descriptor gives right of access only to a code execution module. *Readonly* and *presentincore* are examples of flags that can be included to control the mode of access within the various kinds.

Having passed the access method as valid, the *kind* field is then used to define the layout and interpretation of the rest of the descriptor, both of which may vary for the three kinds.

The simplest bound check is one against an upper limit, offset from zero, with the BOUND field being of type *addressrange*. However, many arrays do not naturally start at zero which leads to a call for two bounds, both potentially of type *integer*. This would make a descriptor untenably large. A more practical solution is to use an upper bound only, but augment it with a single one-bit field which specifies whether the bound is assumed to be offset from zero or from one. This is a neat compromise, not yet implemented in any system, which takes care of the vast majority of subscript ranges at small cost. Let these fields be added i.e.

bound: *addressrange*;
offset: 0..1;

If the area has a lower bound other than 0 or 1, e.g.

var *history*: **array**[−54..1914] **of** *dates*;

then the offsetting will have to be done explicitly before INDEX is entered. Thus we have:

```
history [year] ⇒ LOAD year
                 UP   54
                 NAME A
                 INDEX
```

The descriptor for *history* would have OFFSET = 0 and BOUND = 1968. (Note: UP is used instead of ADD which is a pure stack instruction with no arguments.)

A further field is sometimes found in descriptors which enables the bound check to be omitted. The reason why the ICL2900, for one, includes this switch is discussed later, but it does have a valuable application in accessing items via constant subscripts e.g. *A*[7]. The problem is how to switch the field on and off. Several solutions present themselves:

(1) Dynamically control the switch with ordinary masking instructions. This is far more trouble than the small saving in speed warrants, and could even be as longwinded as

```
LOAD  ACC descriptor
AND   ACC descriptor copy with bit set (or unset)
STORE ACC descriptor
```

(2) Keep two copies of the descriptor at the outset and let the code generators select whichever is appropriate. This is fairly acceptable for declared structures where only one descriptor exists, but is certainly not for those that are components of another structure. Two arrays of descriptors for the same array row is not a fair price to pay for inhibiting a bound check.

(3) Have two versions of the INDEX instruction. One would perform the check and the other would omit it. The compiler knows which is needed at each access. This is a solution which has no space or time overheads and even renders the switch in the descriptor redundant. Strangely, no existing machine has taken this approach.

Given a SAFEINDEX (offset, scale and add only) instruction, access to *A*[7] becomes

```
A[7] ⇒ CONST 7
       NAME A
       SAFEINDEX
```

If the instruction is going to be used often with constant indices, it might be better to split it into two and have a version, CONSTINDEX, which takes a constant argument thus giving

```
A[7] ⇒ NAME       A
       CONSTINDEX 7
```

SAFEINDEX would then be used when a variable subscript has been verified by other means. For example, in

```
var i:1..10; a:array [1..10] of real;
 for i := 1 to 10 do a[i] := 0;
```

the accesses to *a*[*i*] cannot possibly go wrong. Knowing this, the compiler would generate SAFEINDEX rather than INDEX.

1.5 *Scaling*

At this point, the checked and offset subscript is in user's units. The next step is to convert it to machine units, which may be bigger or smaller than the user's. For example, a subscript for an array of two-word reals must be doubled before becoming the effective index. A general *size* field might be
size:(*bit*, *halfbyte*, *char*, *byte*, *halfword*, *word*, *double*, *quad*)
though few machines would need such a range.

The first three functions of the descriptor program can now be incorporated in a definition of INDEX.

```
procedure index; {descr. in tos, subscript in tos2}
 begin
  check (subscript, {between} tos.offset, {and} tos.bound);
  subscript := subscript-tos. offset;
  subscript := scaled (subscript, {by} tos.size);
 end;
```

The way in which the function *scaled* is defined depends on which of the size options is the basic machine unit. For example, if the machine addresses by words then scaling for doubles will multiply by 2, scaling for bytes will divide by 4 (say).

There is one serious problem, however, that does not seem to be appreciated in machine design except at the latter stage when a 'fix' is employed to get around it. If the machine unit is not a multiple or factor of all possible sizes, the scaled index cannot be sensibly added to the start address.

For example, if the machine unit is a word, a subscript that is presented in bytes will produce a byte index which is incompatible with the address. The obvious solution is to store all addresses at the bit level but this would not meet with much acceptance in the real world. Other more practical solutions are discussed in Section 3.4.

Another disadvantage of this size definition is that it assumes that all the elements are the same. In modern languages and systems, the elements of an array are not restricted to FORTRAN-like integers and reals and might well be records of considerable size. Now, it is tempting to include a generalised scale factor of type *addressrange*. The difficulty comes in inserting the field selection between the scaling and adding. Assume

```
var coords:array [0..99] of record
  x:integer;
  y:real
 end;
```

Such an array of records would be represented as in **Fig. 2**(a). (The following discussion assumes that the size of the operand to be loaded by a descriptor is defined by the loading instruction —not the descriptor—see Section 3.2.)
When accessing *coords*[*i*].*y*, INDEX needs both a subscript and a field selector. There are three ways of achieving this.

(1) Use the idea of CONSTINDEX to perform the selection after the subscripting. The trouble is that scaling would then be done twice. Another instruction might be designed which will omit scaling and checking and simply add the given constant to the ORIGIN. If the offset is non-zero, it must only be subtracted once, so SELECT ignores the offset as well. Access now becomes

```
coords[i].y ⇒ LOAD    I
              NAME    A
              INDEX       (check, offset, scale × 3 and
                           add)
              SELECT 1    (add field offset for y).
```

There is a danger with any of the non-checking instructions, and compilers must be very reliable before using them.

(2) Without these variations of INDEX, the offset and scale operations have to be removed and done explicitly beforehand. The descriptor is also affected and all fields must be in machine units because that is how the subscript will come in. For example

```
var coords1:array [1..100] of record etc
```

would be represented as in Fig. 2(b) and becomes

```
coords1[i].y ⇒ LOAD     I
               DOWN     1      (offset)
               SCALE    3      (scale)
               UP       1      (field selection)
               NAME coords1
               INDEX           (check and add)
```

A compiler could usually reduce this sequence by remembering the offset and combining it with the field selection i.e.

```
coords1[i].y ⇒ LOAD     I
               SCALE    3
               DOWN     2      (i.e. −3 + 1)
               NAME coords1
               INDEX
```

This is only one instruction more than the first approach and has the advantage that every access via the descriptor is thoroughly checked. Its disadvantage is that the descriptor must be in machine units.

1.6 *Adding the origin*

It is not generally realised that there are two distinct ways of combining the calculated index with the origin of the structure as given in the descriptor. The principle common to both is that the descriptor should be left in a state that will either forbid further accesses or will allow them on the same controlled basis as before. The two approaches are:

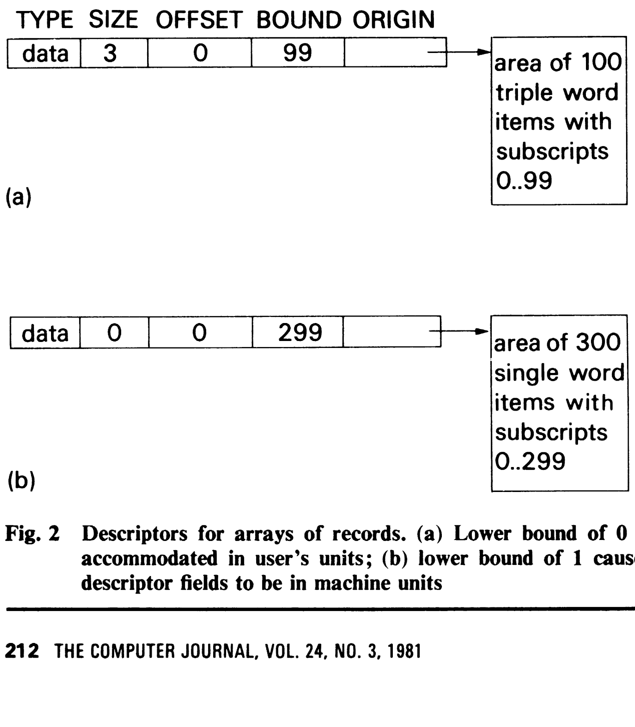

TYPE SIZE OFFSET BOUND ORIGIN
data
3
0
99
area of 100 triple word items with subscripts 0..99
(a)
data
0
0
299
area of 300 single word items with subscripts 0..299
(b)

Fig. 2 Descriptors for arrays of records. (a) Lower bound of 0 is accommodated in user's units; (b) lower bound of 1 causes descriptor fields to be in machine units

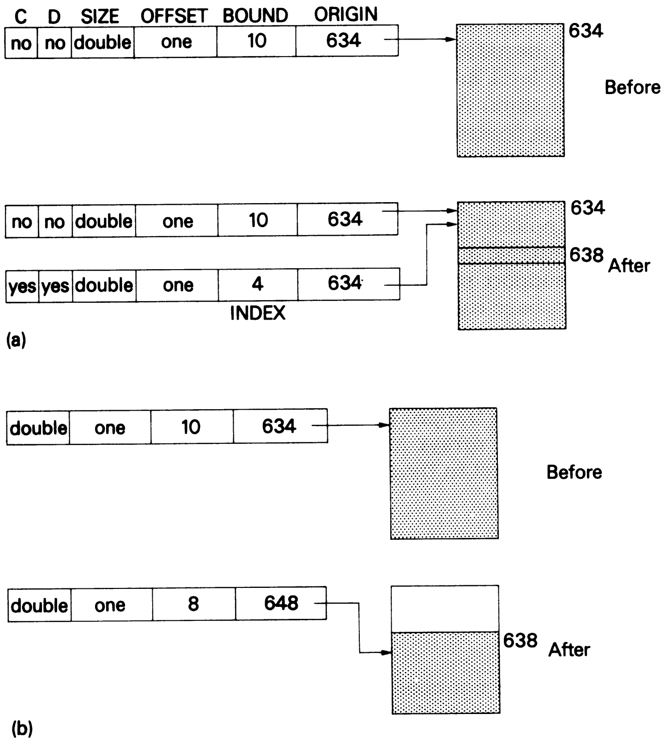


Fig. 3 Two methods of indexing *a*[*i*] where *i* = 3. (a) Indexing by replacement; (b) indexing by balancing

(1) *Indexing by replacement.* The descriptor is copied, the bound field is replaced by the index and the change is indicated by setting an 'indexed' flag. Access is made via the copied descriptor which can be stored and reused for further accesses to the identical element.

(2) *Indexing by balancing.* The index is added into the origin field and simultaneously subtracted from the bound. Subsequently, access can only be made to elements further on in the structure.

These methods are illustrated in **Fig. 3.** Both yield a descriptor that is susceptible to automatic index checking and that cannot be indexed to point outside its original bounds. Their advantages and disadvantages are:

(1) Replacement prevents reindexing from the current position, thus forcing an Iliffe vector scheme for multidimensional arrays. This indexing method is the one that is used in a 'segments on demand' system (see Section 2.2). Balancing alters the address field which renders the descriptor unusable for segmenting purposes.

(2) Balancing alters the original descriptor, which means that, either copies must be kept, or the indexing must be done 'on the fly' as part of an operand rather than an operation in itself. Balancing allows reindexing and hence favours a linear representation for multidimensional arrays.

1.7 *Ideal descriptor*

Given all these factors, it is obvious that deciding on a descriptor and its program is a matter of compromise. The authors' suggestion is:

```
type descriptor = packed record                    Bits
  kind:(data, code, string, descr);                  2
  size:(bit, byte, word, double);                    2
  offset:0..1;                                       1
  bound:segmentrange;                               16
  address:addressrange;                             20
  indexed, copy:boolean                              2
  other:bits                                         5
end;                                                 —
                                           Total    48
```

with

```
procedure index;
  begin
    check (tos2, tos.offset, tos.bound);
    tos2 := tos2 - tos.offset;
    tos2 := scale (tos2, tos.size);
    tos.bound := tos2;
    tos.indexed := true;
    tos2 := tos;
    pop(tos)
  end;
```

In conjunction the three variations of INDEX are essential. In summary, these are:

SAFEINDEX	As for INDEX but without the bound check
CONSTINDEX	As for SAFEINDEX but with the subscript as an immediate operand
SELECT	As for CONSTINDEX but without any scaling or offsetting.

2. The application of descriptors

2.1 *Name and addresses*

A stack consisting of data frames and accessed by means of a display or group of name bases is now the accepted means of representing a procedural high level language. In this representation, the *name* of an item is considered to correspond to its ordinal position within the declarations for a particular level. There are distinct advantages to keeping this correspondence one to one:

(1) The maximum offset in a stack frame will be comparable to the maximum number of names that are declared at any one level rather than the length of storage their items occupy. This reduces the demands on operand size for stack instructions, 10 bits being quite reasonable.

(2) If the length of structured item is not known at compile time, this does not affect the binding of names for subsequent declarations.

In other words, one machine unit (or word) will be allocated to each variable. If the variable requires more, this word will contain a descriptor pointing to the complete area allocated to the structure.

Names and addresses are therefore assigned as follows:

Names

(1) If an item is declared, its name is defined by its level of declaration and its ordinal position within that level.

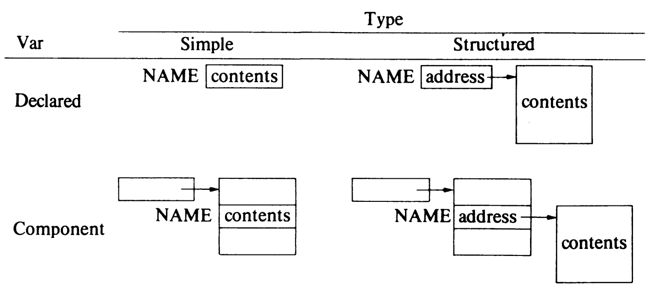
(2) If the item is a component of a structure, the name is deduced either at compile time (field selection) or run-time.

Addresses

(3) If the item is simple, its address is transparent and its name indicates the contents directly.

(4) If the item is structured, the name indicates the address of the start of the contents, this address being part of the descriptor associated with the structure.

Table 1 The relationship between names and addresses



These rules are summarised in **Table 1.**

The advantage of this approach is that, once a name is found, access depends only on its type, that is, simple or structured. However, there is a conflict between the source and machine levels as to how to classify items into these basic types. Several criteria might apply, such as what the syntax says, whether an indirect access method is required and even whether an item can fit in one word. For example, in Pascal, a record is structured according to the syntax but does not need indirect access. Another anomaly is a pointer which has its own class in the syntax (i.e. reference), fits in a single word when implemented, but definitely implies indirect access.

It is obvious that these ambiguities must be settled, not by the language designer, who has already stated his position, but by the compiler writer and machine designer. The following interpretations will maintain the integrity of a uniform name and address system.

(1) *Records.* The offset of a field in a record can be computed at compile time. Since this is the expressed intention of the language designer (Jensen and Wirth, 1972, p. 42), representing a record with a descriptor is inappropriate. The solution is to 'flatten' a record and treat each field as a variable in its own right, as in

NAME	contents
NAME	contents
NAME	contents
NAME	contents

This flattening holds for both declared and component records. Moving a sequence of fields could be done by naming each field, or the MOVE instruction can be defined to accept the first name and number of names.

(2) *Reals.* Following on from records, the solution for reals is simple. A real is regarded as (some version of)

type *real* = **packed record**
mantissa:*somerange*;
exponent:*someotherrange*
end;

Then

var *x*:*real*

can occupy one, two or as many words as dictated by the hardware and the implementor. The operative name as far as the programmer is concerned is that of the record, but it might be a useful trade-off to allow access to its fields. The benefit of this scheme is that reals do not become a special case if they are represented in more than one word. The operations peculiar to reals, as opposed to any other records, are introduced by the appropriate syntax, which is not valid for any other type.

(3) *Pointers.* If a pointer is considered structured, then the referenced variable might be allocated individually off-stack. Since such variables are typically small records, this might introduce considerable inefficiency at operating system level in recording their whereabouts in actual or virtual store. A neat solution which puts the heap under program control defines

type *heaprange* = 0. .*heapmax*;
var *heap*:**array** [*heaprange*] **of** *word*;

and a pointer as a simple integer variable. A call to create a new variable on the heap is handled by the program's run-time support package. Likewise, disposing such variables and garbage collection will be left to the compiler writer to implement or not. The operation $P\uparrow$ is therefore semantically equivalent to *heap*[*P*]. The nil pointer can be set to a large negative number so that accesses to *heap*[*nil* + *any field*] will be trapped by the INDEX operation on the heap. In addition, arrays inside a heap variable will have their indices subject to the bound checking of their own descriptors.

In summary therefore, it is only for arrays that descriptors should be used; these require the full facilities of bound checking, indirect access, and subscript scaling.

2.2 *Segments*

Descriptors are not confined to use by user programs alone: they can form part of the operating system's housekeeping. Here the idea of a segment is introduced. A program logically consists of several segments: some for code, some for data, one for the heap and a special one for the stack. The name and address of a segment depend on the way in which the segment is handled at operating system level. There are two approaches:

(1) *Segments on demand.* A segment is an exact equivalent of a variable sized program structure. The name of the segment is that of the code procedure or data structure which needs it. When the program first accesses the code or data, it asks the operating system for a specified number of words (BOUND − OFFSET + 1) and receives in reply a linear address. This is then placed in the descriptor. (e.g. Burroughs B6700).

(2) *Segments in virtual memory.* The program specifies at the outset how many code and data segments it requires and what their sizes are. This information is recorded in a segment table and constitutes the virtual memory for the program. The segment table contains an appropriate mapping of each segment on to the real linear memory, thus obviating the program from retaining the master descriptor for each segment. This, coupled with the implied need to keep the table to manageable proportions, encourages a compiler writer to put several program structures in one segment (for example, all the arrays in a FORTRAN COMMON block). A descriptor can therefore describe any portion of a segment (e.g. ICL2900).

From the operating system point of view, there is not much to choose between these two approaches. Virtual memory assists in protecting programs from each other at the expense of a table look-up for actual addresses (although associative memories and slave stores reduce this). The demand system is fast, but relies on the system software to handle descriptors responsibly; protection could be enhanced by capabilities in the segments themselves. For both approaches, information pertaining to multiprogramming such as dirty bits, presence bits, copy bits etc. can be kept in the descriptor or the segment table, but these are transparent to a running program and are maintained entirely by the operating software and hardware. Bauerle (1974) gives a most readable account of the overlaying of segments on demand and Tanenbaum (1975) discusses fully the virtual memory approach to segmentation.

The difference between the two methods comes in the way they adapt to indexing by replacement or balancing. As mentioned in Section 1.6, balancing cannot be used with segments on demand. Replacement favours Iliffe vectors and therefore might not dovetail well with the segment table approach.

3. A critique

3.1 *Two descriptor implementations*

'A descriptor is a control word used to locate and describe areas of data or program storage.' (Creech, 1970).

'A descriptor is a 64-bit entity which formally describes an item of information in the store.' (ICL, 1976).

Though superficially the same, there is a subtle difference between the above two definitions. The B6700 has one word describing an area; the 2900 thinks in terms of two words describing an item. The preceding sentence of the 2900 definition, 'The instruction code makes extensive use of descriptors for indirect addressing', strengthens the impression that the item will more often than not be simply one word.

For a compiler writer, the impact of descriptors is second only to that of a stack in governing the design of the object code and the choice of strategies to generate it. It is accepted that descriptors mean more data space but it is expected that there will be a corresponding reduction in code size. Creech (1970) adds to the above description: 'One does not have to stretch the imagination too far to view the descriptor as a 51-bit code sequence which is executed when encountered during [normal] accessing.' However, if the 'program' in a descriptor digresses even slightly from the needs of a particular high level language, then corrective action might have to be taken in the object code —the descriptor may even become a liability. In this situation, the language implementor will investigate whether the descriptor mechanism can be bypassed. This will only be possible if general purpose index registers (i.e. modifiers) are provided for addressing arrays. If there are no such index registers, then descriptors must be regarded as central to the architecture and the principal, if not only, means of accessing and moving structures.

On the other hand, descriptors should not be mandatory for indirect access to simple items. It is not sufficient for the name to be available as an operand; it must also be possible to store it and to access the item indirectly, for example as a **var** parameter. The B6700 carefully distinguishes between direct and indirect addressing for both simple items and structures, but the 2900 has made the mistake of providing only rudimentary direct addressing and having no way of storing an address for indirect access except in a descriptor. This aside, the definition and operation of both descriptor designs leave something to be desired. Three of the problem areas are now discussed with reference to the descriptor formats given in the Appendix. [The MU5 descriptors are those of the original machine design. Recently, the descriptors have been altered to resemble those of the ICL2900 (Morris and Ibbett, 1979). The older version is retained in this paper because it provides a contrast.]

Table 2 Access to mixed type records via a descriptor

	(*a*) 2900 with inline SIZE alteration		(*b*) 2900 with copy descriptors		(*c*) 2900 with string descriptors		(*d*) B6700 with string descriptors	
C	TYPE	*vector*	TYPE	*vector*	TYPE	*string*	TYPE	*data*
	SIZE	*word*	SIZE	*word*	SIZE	*byte*	SIZE	*word*
	BOUND	303	BOUND	303	LENGTH	4	BOUND	303
	SCALED		SCALED					
	CHECKED		CHECKED					
Q	*vector*, *double*, 0		*vector*, *double*, 303		NOTE: No bound			
	UNSCALED, UNCHECKED		UNSCALED, CHECKED		check done			
x := 0	LOAD.MOD	*I*	LOAD.MOD	*I*	LOAD.MOD	*I*	VALC	*I*
	MULT.MOD	3	MULT.MOD	3	MULT.MOD	12	NAMC	*C*
	STORE.ACC	(*C*) + MOD	STORE.ACC	(*C*) + MOD	STORE.ACC	(*C*) + MOD	INDX	
		3:8		3:8		3:8	STOD	3:5
y := 0	LOAD.MOD	*I*	LOAD.MOD	*I*	LOAD.MOD	*I*	VALC	*I*
	MULT.MOD	3	MULT.MOD	3	MULT.MOD	12	NAMC	*C*
	ADD.MOD	1	ADD.MOD	1	ADD.MOD	14	INDX	
	LOAD.DR	*C*	STORE.ACC	(*Q*) + MOD	LOAD.DR	*C*	BSET	42
	INDEX.DR	MOD			LOADBD.DR	8	LIT8	2
	LTYBD.DR	*Q*			STORE.ACC	DR + MOD	. . .	
	STORE.ACC	(DR)					TWSD	
		7:16		4:10		6:12		6:10

(1) The record is

```
var coords:array [0..100] of
 record
  x:integer;
  y:real
 end;
```

(2) Each sequence stores the number currently in ACC (2900) or on the top of the stack (B6700) into the given field.

(3) Measurements are in instructions and bytes (*i* : *b*).

3.2 *Conflicts between lengths and sizes*

Consider the general instruction:

STORE REG *via* DR *into a destination*

The number of bits that are stored could be governed by the instruction, by the size of the register, by the SIZE or LENGTH in the descriptor register, DR, or by the tag of the destination. There is no consensus among machines as to which part of the instruction has the final say; indeed there is considerable confusion in the machine specifications themselves. On the whole, there is a tendency for the descriptor to dominate both tags and register size, with the instruction remaining neutral. This is an unfortunate choice. Consider the definition of *coords* given earlier and how it would be accessed on an ICL2900.

Because scaling by three is not part of the descriptor 'program', and SELECT is unavailable, the SIZE of the descriptor is set to a single word and scaling is done by inline code. This is quite acceptable. However, because the SIZE governs the length of the operand, every time y is accessed, SIZE will have to be changed to double and scaling inhibited. This is inconvenient in principle and in practice. If the designers of the 2900 had genuinely taken high level languages into account, they could have anticipated this case and provided a streamlined instruction for resetting the control information in the upper byte of a descriptor. As it is, the descriptor must first be indexed, then the new type and a dummy bound loaded from the constant area [**Table 2**(a)]. It is therefore more efficient to set up duplicate copies of the descriptor initially, as in Table 2(b). As mentioned earlier, this would then discourage rows of descriptors.

A third possibility might be to set the size as a factor of both fields and make use of the LENGTH facility. Although 'word' is a factor for the representation of both x and y, LENGTH, on the 2900, works only in bytes. The advantage is that LENGTH can be altered in one short instruction. Although there is a major complication in that scaling must be done in machine units [Table 2(c)], this is an attractive compromise. Unfortunately, it is not acceptable because using the LENGTH field eliminates bound checking.

In comparison, the descriptors in the original MU5 retain the BOUND in conjunction with the LENGTH and, furthermore, permit SIZE to vary from bits to words. Thus scaling can be done in user units (in this case, 32-bit half words). While one may argue that addressing at the bit level, which is the basis for this flexibility, is downright inefficient, this is one of those cases where the architecture is helpful and should be used. The inefficiencies are confined to the hardware and it can be expected that they will be reduced as new technology is developed.

On the B6700, 48-bit reals are adequate, which is just as well because an array of mixed records is virtually impossible to implement. Making use of copies of the descriptors, as in Table 2(a) and (b), is not useful because scaling cannot be switched off. Keeping the SIZE as one word, two words can be moved by setting the descriptor type to *string* and using a 'transfer words' instruction as in Table 2(d). The tag, set to double by the 'extend' instruction, is copied as well.

On an ideal architecture, the problem of mixed types could be confined by setting the basic address unit large enough for all types and providing partword accessing by descriptors and instructions (Mullins, 1978). Two word reals, rarely needed, will be treated as multiwords, as previously stated.

3.3 *Moving structures*

Since descriptors describe areas, it is reasonable to expect that moving structures will be simple. The simplicity depends on the definition of the MOVE instruction. Both the 2900 and the B6700 require two string descriptors for this process, leading to the same switching problems of the previous example. Whereas the B6700 has instructions to transfer words, bytes, characters or digits, the 2900 MOVE instruction refers to bytes only. More serious than the units is the fact that descriptors are mandatory for a MOVE. Even on the B6700, a simple name is not an acceptable operand. On the 2900 this means that a descriptor must be created from a stored upper half and an offset; on the B6700 it has the drastic effect of forcing all structures off-stack.

On an ideal architecture, the MOVE count would refer to names and each element be handled by STORE. Thus, for example, an array of bytes could be correctly transferred to an array of words and be unpacked in the process. Of more value is the freedom to specify the source or destination as a stack address or descriptor, indexed or not.

3.4 *Partword addresses*

An interesting characteristic of the B6700 string descriptor and the 2900 and MU5 vector descriptors is that some or all of the user's units available through SIZE are at a finer resolution than the machine units possible through ORIGIN. **Table 3** highlights these relationships.

There are two ways of handling the case where $U < M$.

(1) The 2900 and MU5 rely on the finer index being provided by a subscript in an explicit modifier (register or store location). For example, if the descriptor register contains a bit vector descriptor, access via (DR + MOD) would scale the value in MOD down by three bits and then use these for the bit index. If no modifier is specified, then the zeroth (or leftmost) partword is accessed. However, if scaling is inhibited, the bit index is undefined.

(2) The B6700 interprets the INDEX field according to the size at the time of access. In effect, this is an in situ version of the first, with the advantage that there is only one way of accessing the zeroth partword and less chance of an error.

There seem to be two reasons for having special string descriptors: to obtain a finer address resolution and to define an operand length that is independent of the bound. By providing a different descriptor 'program' where finer addresses are most needed (i.e. in text processing), the B6700 and MU5 do not hamper normal access to vectors of words; on the 2900 a 'scale by 4' must be tolerated for every integer item accessed. A LENGTH can be used by sophisticated store to store instructions on MU5 and the 2900 but it has to be put into the descriptor and this is not always possible at compile time.

Table 3 Relationship between units in descriptors

Machine	*Vector sizes*	*String sizes*
B6700	$Mword \leqslant Uword..double$	$Udigit..word \leqslant Mword$
2900	$Ubit < Mbyte \leqslant Ubyte..quad$	$Mbyte = Ubyte$
MU5	$Ubyte..word \leqslant Mword$	$Mbit \leqslant Ubit..word$

Mx	x is the machine unit used in ORIGIN
$Ux..y$	$x..y$ is the range of user's units in SIZE.

Table 4 Comparison of array accesses using descriptors. Measurements are in instructions and bytes (*i* : *b*)

	B6700			2900		
A[*I*] := *X*	VALC	*I*		LOAD.ACC	*X*	
	NAMC	*A*		LOAD.DR	*A*	
	INDX			STORE.ACC	DR + (*I*)	
	VALC	*X*				
	STO		5:8			3:8
A[*I*] := *A*[*J*]	VALC	*I*		LOAD.MOD	*I*	
	NAMC	*A*		LOAD.ACC	(*A*) + MOD	
	INDX			LOAD.MOD	*J*	
	VALC	*J*		STORE.ACC	(*A*) + MOD	
	NAMC	*A*				
	NXLV					
	STO		7:11			4:12

The B6700 approach is to specify the length as an operand to the transfer or compare instruction, thus avoiding the descriptor set-up and any possible conflicts. Notice that the 2900 string descriptor is deficient in that the BOUND is missing and the LENGTH can only be specified in bytes The need for lengths defined in bits has only recently been acknowledged by a second major change to the 2900 hardware.

3.5 *Improving descriptor programs*

Before discussing alternative means of implementing arrays, possible ways of improving the descriptor access are examined. On the B6700, descriptors are so integrated into the design that alternative methods are impossible. There is no indexing possible on the stack and all off-stack segments are accessed via descriptors. Because of the problem of doing a field selection after an INDEX (the B6700 has no SELECT or SAFEINDEX instructions), Sale (1976) flattened all structures so that an array of arrays or an array of records is held in one segment with one descriptor. The 2900 descriptors are not as central to program structure as the B6700's but one cannot get away from them because they provide the only means of doing indirect addressing. There are two indirect operand forms which enable most instructions to reference descriptors. The first is 'descriptor in store' with an optional subscript in the modifier register (MOD) and the second is 'descriptor in DR' with the optional subscript in store. The use of the B6700 and ICL2900 instructions is shown in **Table 4.**

Notice that the 2900's 'descriptor in DR' form could be used to optimise *A*[*I*] := *A*[*J*] because any indirect access places the resultant descriptor in DR. The code size is then (3:10).

To obtain the advantages of automatic bound checking and scaling, descriptors must first be set up. It is strange that in both machines, the sequences to do this could be considerably improved by the definition of special purpose instructions. A descriptor for a simple vector in the local data frame is constructed on the B6700 in two instructions.

CONST48	*type, size, bound*	
SETTAG	*datadescriptor*	2:9

The origin does not appear at this stage since space is only allocated for the array when it is first used. Recognising this and the fact that the tag is a fixed code leads to the following instruction:

MAKEDD *type, size, bound* ⇒ creates a word on the top of the stack containing the information supplied as the four bytes, zero in the last two and the tag set for a data descriptor. 1:5

This is achieved in 5 bytes, as opposed to the 9 on existing B6700s.

The 2900 sequence for constructing a single descriptor is

STORE.LNB	MOD	[Get local name base	
LOAD.DR	MOD	[into DR, via MOD	
LTYBD.DR	*W*	[Set up type and dummy bound]	
LBOUND.DR	*bound*	[Insert actual bound	
INCADDR.DR	*address*	[Add the origin	
STORE.DR	*TOS*		6:14

where *W* is a constant, prestored by the compiler, containing an appropriate type and size and a dummy bound. The type, size and bound cannot appear as an immediate operand in the LTYBD instruction because it is more than 18 bits long. This sequence could be considerably improved by recognising the need to load type and size information separately from the bound. Let DESCR be defined as

DESCR.*reg* *type*, *size*, *scaling*, *checking* ⇒ creates a descriptor in DR with the information supplied in the 7 control bits, a bound of zero and the contents of the base register (one of LNB, XNB or CTB) in the origin.

The choice of registers allows for descriptors based in any of the three name spaces to be created with this instruction, followed by two more to add in the relevant bound and address. Since only seven bits are relevant in the control information byte, DESCR will always be a short instruction. The set up sequence becomes

DESCR.LNB	*type, size, scaling, checking*	
LBOUND.DR	*bound*	
INCADDR.DR	*address*	
STORE.DR	*TOS*	4:8

Two words of set up code, with no subsidiary constants, is a reasonable price to pay for descriptor access.

An alternative way of reducing the overhead is to adopt the policy of base descriptors for local arrays (Rees *et al.*, 1981). Sharing descriptors in this way means that the BOUND field must be empty and bound checking inhibited, which (*a*) defeats most of the object of having descriptors and (*b*) means that the checking has to be reinstated as inline code. Bearing in mind that there is a precedent for introducing new instructions on the 2900 there is more to be gained in the long run by using the structured features than by designing code and compilers as if the 2900 were a linear machine.

3.6 *Alternative methods for accessing arrays*

Up till now, it has been assumed that the natural way to store matrices and higher order arrays is with Iliffe vectors. Like the substantive descriptor, these cannot be set up as constants since a new address will be assigned for each activation of the matrix. There is a persistent feeling among compiler writers that Iliffe vectors are not worth the additional data space and the overhead involved in setting them up. On a linear machine this is true, since the addresses would be merely an expansion of the mapping function. With hardware descriptors, a different picture emerges.

Because it allocates segments only on demand, the B6700 provides operating system support for setting up Iliffe vectors. The number of dimensions and upper bound (offset to zero) for each are sent to an operating system procedure, arraydec, which returns the substantive descriptor. This points to a table in the operating system containing the details of the array. As soon as the descriptor is used, the hardware and operating system combine to set up the portion of the structure required (Bauerle, 1974). The setting up of the call to arraydec for a matrix takes (11 instructions, 18 bytes).

On a 2900 suitably endowed with the suggested DESCR instruction row descriptors are set up with an explicit sequence taking (8 instructions, 20 bytes) i.e.

```
      DESCR.LNB      control information
      LBOUND.DR      U2 − L2 + 1
      INCADDR.DR     address plus (U1 − L1 + 1)
                       minus (U2 − L2 + 1)
      LOAD.MOD       −(U1 − L1)
LAB:  INCADDR.DR     (U2 − L2 + 1)
      STORE.DR       TOS
      COMPINC.MOD    0
      JUMPNE         LAB                    8:20
```

(Without DESCR, the sequence takes 10 instructions, 26 bytes.)

The inner loop contains 4 instructions in 10 bytes. Having executed this sequence, access to any element in a matrix is given by

```
LOAD.MOD     subscript1
LOAD.DR      (address) + MOD
LOAD.ACC     DR + subscript2
```

Higher order arrays would repeat the second instruction. If the lower bound is not zero, an explicit adjustment must be made at run-time, as shown in the examples of **Table** 5(a) and (b). Since neither a lower bound or a lower bound bit is provided in the 2900, there should be a more efficient way of achieving this adjustment, such as instructions to subtract one from a store location.

Without Iliffe vectors, a matrix is represented linearly and the index computed by multiplication and addition of the subscripts. This can be combined with the necessary bound checking and lower bound subtraction by operating in a uniform way on a dope vector. For the example in Table 5, a suitable dope vector and access sequence would be:

```
B[I][J] ==> LOAD.DR BDOPE         BDOPE |  4 | upper
            LOAD.MOD 0                  |  0 | lower
            COSA      I                 | 13 | size
            COSA      J                 |  6 |
            LOAD.ACC (B) + MOD          | −6 |
                                                 5:2
```

COSA stands for check, offset, scale and add—in that order. Mindful of the hardware realisation of the instruction, it is defined so that the lower bound is checked immediately before being subtracted as an offset and index is accumulated in MOD. [This instruction is very similar to the SUB2 instruction described by Ibbett and Capon (1978) for the MU5.]

Foreseeing the need for such an instruction, ICL provided VMY—vector multiply. VMY is an awkward version of the hypothetical COSA. In the first place, offset and scaling are

Table 5 Various methods of accessing arrays

(*a*) B6700 Iliffe vector	(*b*) 2900 Iliffe vector	(*c*) 2900 Dope vector	(*d*) 2900 inline checks	(*e*) 2900 Total check only
B has TYPE *data* SIZE *word* BOUND 5	*B* has TYPE *descriptor* SIZE *double* BOUND 5 SCALED CHECKED	*B* has TYPE *vector* SIZE *word* BOUND — SCALED UNCHECKED	*B* has TYPE *vector* SIZE *word* BOUND — SCALED UNCHECKED ORIGIN of *B* + 6	*B* has TYPE *vector* SIZE *word* BOUND 65 SCALED CHECKED ORIGIN of *B* + 6
rows have TYPE *data* SIZE *word* BOUND 13 ORIGINs set by the OS on demand	rows have TYPE *vector* SIZE *word* BOUND 13 SCALED CHECKED ORIGIN of *B*[*I*]	BDOPE has TYPE *vector* 0 SIZE *word* 13 BOUND 6 52 SCALED −6 CHECKED 1 ORIGIN 12		NOTE: Total subscript check only
VALC *I* NAMC *B* NXLN VALC *J* LIT8 6 ADD NXLV	LOAD.MOD *I* LOAD.DR (*B*) + MOD LOAD.MOD *J* ADD.MOD 6 LOAD.ACC DR + MOD	LOAD.DR BDOPE VMY *I* STORE.MOD *TOS* VMY *J* ADD.MOD *TOS* LOAD.ACC (*B*) + MOD	LOAD.ACC *I* JLTZERO error COMP.ACC 4 JGT error STKLOAD.ACC *J* COMP.ACC −6 JLT error COMP.ACC 6 JGT error STORE.ACC MOD STKLOAD.MOD *TOS* MULT.MOD 13 ADD.MOD *TOS* LOAD.ACC (*B*) + MOD	LOAD.MOD *J* STKLOAD.MOD *I* MULT.MOD 13 ADD.MOD *TOS* LOAD.ACC (*B*) + MOD
7:11	5:12	6:16	14:38	5:12

(1) The array is
 var *B*:**array** [0..4] **of array** [−6..6] **of** *integer*;
(2) Access is to *B*[*I*] [*J*];
(3) Measurements are in instructions and bytes (*i*:*b*)

done before the upper bound check. Thus the lower bound is in user units but the upper bound is in machine units! Second, MOD is used by the instructions as a working register so that subscripts cannot be accumulated there in a natural way. Third, the definition is incompatible with the descriptor bound checking mechanism in that the scaled subscript must be <BOUND (descriptors) and ⩽BOUND (VMY). The actual 2900 code for using VMY is given in Table 5(c).

From the byte counts in Table 5, it can be seen that after four references to the matrix in a program, the Iliffe vector method will have recouped the size of the code for setting up the row descriptors, compared to VMY. As regards speed of access, even without going as far as to time the sequences, it is obvious that VMY is a lengthy operation—three store accesses, two comparisons, a multiplication and a subtraction.

A third option is to linearise the array and do the checking, scaling and row mapping inline as shown in Table 5(d). It is obvious from the size of the code that the 2900 does not provide any instructions to assist this time-honoured method and that Iliffe or dope vectors are far more efficiently implemented. Timing figures indicate that VMY is at present slower than the equivalent inline sequence. While VMY could be improved in the future, the explicit JUMPs in the inline code will always disturb the pipeline.

These results indicate that in terms of code size and speed, the investment in Iliffe vectors is worthwhile. The justification for the additional data space can only be subjective. First, the occurrence of two dimensional arrays in student and scientific programs is between 9·5% (Knuth, 1971) and 14·1% (Robinson and Torsun, 1976) of all variables declared. They do not, therefore, represent a significant portion of the entire data space and the increase caused by row descriptors can be absorbed. Second, for a very large (say 100 × 100) matrix, the increase would be 100 descriptors in 10000 elements—once again, this is tolerable when viewed against the speed of access achieved by virture of these 100 descriptors. Third, an explosion of row vectors in three and higher dimensional arrays would be very rare—Robinson and Torsun report only one three-dimensional array and Knuth measured 1·2% of accesses to arrays or functions as having three arguments. Finally, higher dimensions in FORTRAN programs are used to hold what are very likely record structures and would be represented as such in a modern language.

Table 5 gives rise to two further observations.

(1) A second non-zero lower bound will increase the Iliffe vector code by two bytes for making the adjustment. The dope vector sequence is unaffected. The inline code remains the same because the address in the descriptor can be suitably offset at compile time. It is important to realise that this can only be done if the descriptor bound checking is inhibited.

(2) With delayed indexing employed on linearised arrays, the subscripts will be stacked in order of appearance and need to be reordered for a row storage mapping function. In Table 5(c), *I* and *J* (which could have been expressions) are loaded, checked and stacked, whereupon they must be swopped because *I* is needed for the initial multiplication. The instruction STACKLOAD.MOD *TOS* does this neatly, but the sequence becomes more complicated with more than two subscripts. On a linear machine, the subscripts would be stored in named temporary locations and the question of order does not arise. This illustrates that, with a push-down store, delayed evaluation cannot be carried to any length without causing difficulties which were not inherent in the original operation.

3.7 *Descriptors and optimisation*

Descriptors are themselves an optimisation: they represent the factoring out into hardware of the common operations of scaling and checking subscripts. Of the immediate optimisations commonly used in compilers, one has relevance here, namely constant folding. Descriptor access leaves two operations to inline code—lower bound subtraction and scaling for multi-words. If the subscript is a constant then these operations can be done at compile time. The saving is two or three bytes in each case on the B6700 and 2900 and is simple to include in the compiler.

Of course, the bound checking done automatically by the descriptor is then superfluous and instructions such as CONSTINDEX should always be provided. If descriptors are not used for access to arrays, the equivalent inline bound check takes 12 bytes (or 10 if a bound is zero) as shown in Table 5(d). In this situation, the recognition of constant subscripts will result in considerable savings. Following on from this idea, Welsh (1981) developed a technique for checking well defined subscripts at compile time. This is an admirable aid to program development and, on a linear machine such as the 1900, can certainly reduce the overhead of bound checking. It could therefore be argued that because checking is not required in a large number of cases, more efficient code can be obtained without descriptors. As shown in **Table 6** this is only true for a simple vector without checking. Since this is a common case, the descriptor method can match that of the inline code by providing a copy descriptor with bound checking off and the address suitably offset. The compiler can then choose the appropriate 'program' at each access. However, for matrix access there is nothing to be gained by avoiding descriptors for Iliffe vectors.

Given this evidence, it is surprising that the 2900 Pascal compiler does not use descriptors until forced to do so for the final access to a subscripted item. The reason is a combination of misplaced faith in compile time range checking as an aid to

Table 6 The effect of compile time subscript checking on the 2900

Access to	*Using descriptors*		*Inline code*	
	Instr	*Bytes*	*Instr*	*Bytes*
$A[I]$	3	8	7	20
$A[I]$ no check	3	8	2	6
$B[I][J]$	5	12	14	38
$B[I][J]$ one check	5	12	8	22
$B[I][J]$ no checks	5	12	5	12

(1) The arrays are
var *A*:**array** [1..10] **of** *integer*;
B:**array** [0..4, −6..6] **of** *integer*;
(2) Measurements are in instructions and bytes (*i*:*b*)

reducing code (as distinct from its role in compile time error detection) and a largely unwarranted fear of set-up overheads. For uniform treatment inside the compiler, entire and component arrays are bereft of descriptors, leading to the base descriptor mechanisms described earlier. While it is true that setting up descriptors in a virtual machine is more visibly time consuming than in a demand segment system, it is only by exercising the structured features of a machine like the 2900 that designers and engineers will be encouraged to improve them. The basic philosophy of the 2900 is sound: it is unfortunate that tradition and certain unwieldy aspects of the machine have so far resulted in it being treated as an old-fashioned linear model.

3.8 *Future possibilities*

Although descriptors, as discussed above, are only suitable for aggregate structures, the idea of storing information about data with the data could be extended to scalars. In particular, the bounds of a variable declared on a specific subrange could be stored with it and used by the hardware to validate any attempt to store a value there. For example,

var *daysofweek*: 1 . . 7;
colours: (*red*, *blue*, *indigo*, *yellow*, *green*, *orange*, *violet*);

could be represented as *bounded scalars* as follows

1	7	value

0	6	value

This is the ideal, but the same considerations concerning bounds that were discussed in Section 1.4 are relevant here, i.e. how to reconcile the field width of a bound with the possible values one would want to put in there. These ideas have been explored in Bishop (1980).

4. Conclusions

Descriptors are a powerful tool and provide for easier code generation for high level languages as well as increased security in data handling. Current hardware implementations of descriptors are not easily used by modern high level languages, but could be enhanced by the introduction of additional instructions, particularly to facilitate setting up descriptors dynamically. For future machine designs, the principles developed and discussed in this paper, if adhered to, will make descriptors a truly usable facility. In summary, these are

(1) A descriptor should occupy no more space than the normal basic machine unit e.g. one word (Section 1.1).

(2) To save space, a lower bound can be replaced by a one bit offset, for use by aggregates starting at 0 or 1. Any other starting point will be represented as offset from zero (Section 1.4).

(3) Index instructions which omit bound checking and/or scaling are essential for implementing field selection, as well as for access to arrays with constant subscripts. These facilities should not be controlled by the descriptor (Sections 1.4, 1.5).

(4) If the machine is a stack machine, instructions to add, subtract and multiply by a constant should be available. That is SCALE 3, rather than CONST 3; MULT. These instructions could also set overflow when *addressrange* is reached, rather than the machine's *integer* limit (Section 1.4).

(5) When choosing to implement indexing by replacement or balancing, the effect of these on segmentation and on the methods of storing arrays must be borne in mind. That is, replacement favours segments on demand and Iliffe vectors; balancing favours segment tables and a linear representation for arrays (Section 1.6).

(6) A descriptor should not control the sizes of items fetched or stored in its area because these could well vary. Store and move instructions should be provided with adequate count facilities for this purpose (Sections 3.2, 3.3).

(7) The units in which users may specify items must always be either a multiple of the machine units or an integral factor of them, otherwise additional indexing facilities will crop up on the side (Section 3.4).

(8) Instructions to set up descriptors should be definite and concise. If certain fields of descriptors can be changed at run-time then instructions should be provided to do this (Section 3.5).

Acknowledgements

The assistance and advice of Arthur Sale, University of Tasmania, and David Messham, ICL (Kidsgrove) is gratefully acknowledged.

Appendix

Examples of descriptor formats from the Burroughs B6700, the ICL2900 and the original version of MU5 the Manchester University MU5 are given. The current MU5 descriptors resemble those of the ICL2900.

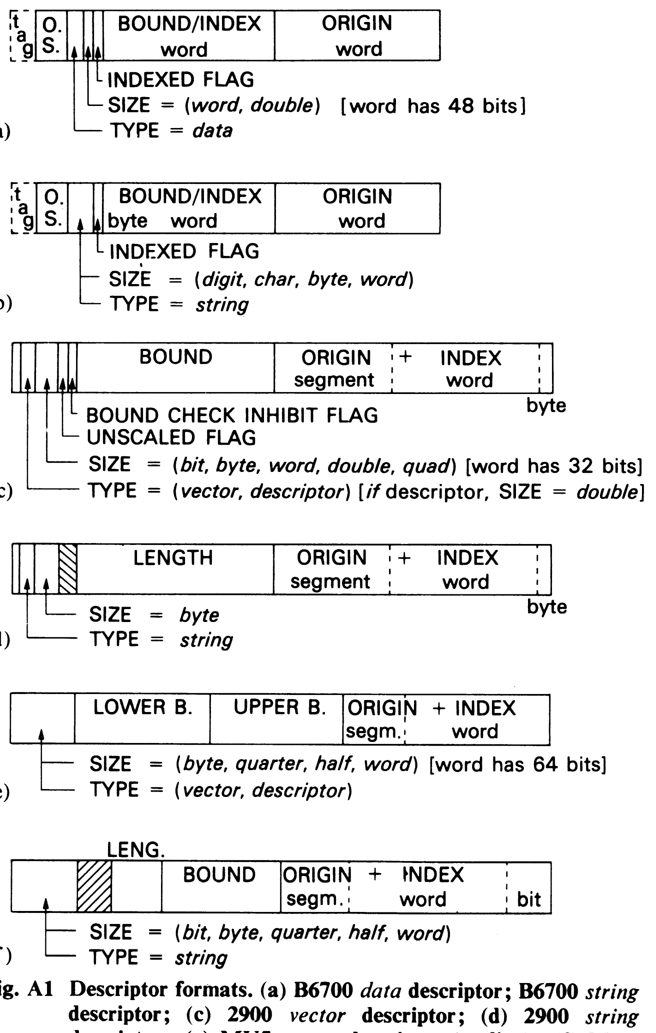


Fig. A1 Descriptor formats. (a) B6700 *data* descriptor; B6700 *string* descriptor; (c) 2900 *vector* descriptor; (d) 2900 *string* descriptor; (e) MU5 *vector* descriptor (outline); (f) MU5 *string* descriptor (outline)

References

Bauerle, D. A. (1974). The Organisation of Fast Store in the B6700, in *Computer Design*, Series 3 (17), pp. 387-398, Infotech.

Bishop, J. M. (1980). Effective Machine Descriptors for Ada, *SIGPLAN Notices*, Vol. 15 No. 11, pp. 235-242.

Creech, B. A. (1970). Architecture of the B6500, in *Software Engineering*, Vol. 1, pp. 29-43, ed. J. T. Tou, Academic Press, New York.

ICL (1976). 2900 Primitive Level Interface, Product Specification Document 2.5.1, Issue 5/1, May.

Ibbett, R. N. and Capon, P. C. (1978). The Development of the MU5 Computer System, *CACM*, Vol. 21 No. 1, pp. 13-24.

Jensen, K. and Wirth, N. (1972). Pascal User Manual and Report, Springer, Berlin.

Knuth, D. E. (1971). An Empirical Study of FORTRAN Programs, *Software—Practice and Experience*, Vol. 1 No. 2, pp. 105-133.

Morris, D. and Ibbett, R. N. (1979). *The MU5 Computer System*, Macmillan, London.

Mullins, J. M. (1978). Code Generation and Structured Architectures, Ph.D. Thesis, University of Southampton.

Rees, M. J. *et al.* (1981). Pascal on an Advanced Architecture, in *Pascal—the Language and its Implementation*, ed. D. W. Barron, pp. 261-275. Wiley, Chichester.

Robinson, S. K. and Torsun, I. S. (1976). An Empirical Analysis of FORTRAN programs, *Computer Journal*, Vol. 7 No. 7, pp. 56-62.

Sale, A. H. J. (1976). Private correspondence, 15 September.

Tanenbaum, A. S. (1975). *Structured Computer Organisation*, Prentice Hall, New Jersey.

Welsh, J. (1981). Two 1900 Pascal Compilers, in *Pascal—the Language and its Implementation*, ed. D. W. Barron, pp. 171-179. Wiley, Chichester.

Book reviews

Cobol Programming, by Peter Abel, 1980; 408 pages. (*Prentice Hall*, £8·40)

Although this book is subtitled *A Structured Approach*, the structure of the book leaves something to be desired. The first chapter, as an introduction to computers, is redundant. If the reader needs this information he will almost certainly not be able to profit from the rest of the book.

The book is based on IBM implementations of COBOL, which makes some of it of limited value. This is, sadly, especially true of the chapters on file organisation methods: sequential files are adequately covered; indexed files are described including much of the gory detail of overflow records; relative files receive only a sketchy treatment; one chapter describes IBM's VSAM files but with no discussion of when such organisation is appropriate.

A major criticism, which is often applied to such books, is the use of the flowchart as a design tool. Structured analysis and design are not mentioned at all, and the flowcharts incorrectly map the PERFORM . . . UNTIL . . . format. Since the programs are correct, they do not implement the given flowcharts; this is confusing to the novice and irritating to the informed.

Several criticisms of detail will suffice to give a flavour of the style. A section is devoted to describing buffers on p. 105, and the only other reference to this topic is a single paragraph on p. 302. REDEFINES is used in example programs before its effect is explained. MOVE CORRESPONDING, whose use can be a major aid in a well organised COBOL program, is described with three short paragraphs and a 16 line listing; the data movements which occur as a result of the example are not disclosed!

Use of the standard COBOL syntax notation is somewhat sporadic, and a summary in an appendix would have been a useful adjunct. The typographical layout of the book could be better: all COBOL is reproduced from chain-printer output, which is occasionally illegible. The author's narrative style is on the whole clear, but his objectives would have been better met by choosing a more limited language subset and giving it fuller treatment. In short, there are better books on COBOL. They are, however, often more expensive.

S. C. Holden (Manchester)

More Chess and Computers, by D. Levy and M. Newborn, 1980; 177 pages. (*Computer Science Press*, $12·95)

This book analyses a number of chess games involving computers played in the years 1975–1978. It starts by outlining the bet made by David Levy in 1968 that no computer would beat him in a match within 10 years. Several challenges were made and most of the resulting games are analysed in the book. The second chapter "The State of the Art" describes several of the games involving the strongest computer chess program CHESS 4, plus one game involving a purely positional program from Germany. The third chapter, on Blitz play, analyses the defeat by CHESS 4.6 of various chess Masters and one Grand Master (Michael Stean) at Blitz chess. Chapter 4 discusses two tournaments which took place in 1977, *The Second World Computer Championship* in Toronto and *The Eighth ACM Computer Championship* in Seattle.

Perhaps the most interesting part of the book is Chapter 5 which describes the relatively new field of microcomputers and chess. This is the commercial end of computer chess and inevitably financial considerations place very tight restraints on the program designer. Chapter 6, "Computer Chess Miscellany" is followed by Appendices on an unsolved problem and a listing of over 50 games played in computer tournaments in 1977. Finally there is a bibliography on computer chess.

The book is unfortunately marred by numerous errors. One might expect a few errors of chess notation involving ambiguous moves or erroneous use of the + symbol for check, but not the large percentage of errors in the illustrated board positions. As none of the illustrations have any indication of their location in a game, the reader must assume that they are positioned correctly in the text. In fact six of the illustrated positions were in the wrong place in the text and in addition seven illustrations had pieces missing, wrongly positioned on the board or transformed to a different piece type.

In summary, the book gives a good description of the state of the art up to 1978, but the reader should ignore the illustrations and play the games through with a chess set.

P. Kent (Didcot)