Lady Ada Mediates Peace Treaty in Endianness War

Thomas Quinot and Eric Botcazou

AdaCore
46, rue d'Amsterdam
75009 Paris, France
{botcazou,quinot}@adacore.com
http://www.adacore.com

Abstract. There is no universal agreement on the order in which the successive bytes constituting a scalar value are stored. Some machines (so-called *big-endian* architectures) store the most significant byte first, while others (*little-endian* architectures) adopt the opposite convention. When porting an application across platforms that use different conventions, programmers need to convert data to the appropriate convention, and this may cause difficulties when exact memory layouts need to be preserved (e.g. for communication with legacy systems).
This paper describes the features of the Ada language that help supporting programmers in these situations, identifies some of their shortcomings, and introduces two novel solutions: a code generation approach based on data representation modeling, and a new representation attribute *Scalar_Storage_Order*, allowing the byte order convention to be specified for a given composite data structure.

Keywords: endianness, retargeting, code generation

1 Introduction

As Gulliver landed on Lilliput, he discovered the fierce war raging between *little endians* — whose soft boiled eggs they would always eat starting with the little end — and the *big endians* who furiously defended the exact opposite standpoint, and ended up in exile on the nearby island of Blefuscu [10].

Nowadays' software engineers, like the explorer of yore, are still finding themselves in the middle of the same battleground, where hardware interfaces, communication protocols, or other external constraints require multi-byte values to be stored and exchanged in either big-endian format (most significant digits first), or little-endian (least significant digits first). The war has been raging for decades [3], with rules imposed by standard interfaces, or stemming from the requirement of interoperability with third party or legacy applications.

All is well as long as all components of a system happen to use the same convention. Splitting any data structure into its elementary components is then just a matter of masking and shifting bits. However, as soon as different conventions must come into play, trouble arises: the order of the bytes constituting a

data structure (the *endianness*) then needs to be changed at strategic conversion points, which may or may not be well identified in source code. It is up to the application developer to identify appropriate swapping points and keep track of whether or not a given value has been swapped at any given point in time. This proves a significant hassle, with a costly distributed impact.

This situation specifically arises when applications that used to rely on legacy big-endian computers, for example based on PowerPC or SPARC CPUs, are retargeted to now ubiquitous Intel-based platforms, which are little-endian. Integration with legacy components, and processing of stored data from existing systems requires that exact data representations be preserved, and software must compensate for the fact that the new platform assumes a different storage order.

Introducing explicit reordering (*byte swapping*) of scalar values throughout software may prove a costly endeavour. The mere extraction of scalars crossing byte boudaries in data structures requires extra shift and mask operations. In addition, the need for explicit code handling endianness conversions hinders maintainability as data structures themselves evolve. This paper discusses how tools can provide valuable assistance to application developers in addressing endianness conversion issues, alleviating the need for such "manual" byte swapping.

In section 2, we first give a summary of the data representation constructs provided by Ada. These allow the explicit specification of a data structure's layout according to an external constraint. They can be used to provide endianness independence to a limited extent. However, users are often disconcerted at first by the exact semantics of these features, which indeed do not provide a fully transparent and general solution to endianness conversion issues. In section 3, we focus on explicit byte swapping approaches, and we present the *Tranxgen* code generator, which affords automated support to produce endianness independent accessors. In section 4, we then describe another solution, introducing a new representation attribute providing transparent in-place access to data of arbitrary endianness.

2 Composite layout specifications in Ada

In this section we discuss standard features of Ada that allow the layout of a data structure to be specified according to an external constraint. We show how these features support endianness independence to a limited extent.

2.1 Record representation clauses

Ada record representation clauses allow developers to specify, for each component of a record:

- its starting position, i.e. the byte offset of the first underlying storage element (the one with the lowest memory address)
- a bit range indicating its specific extent over the underlying storage elements.

```
-- A two byte data structure
type R is record
   X : Character;
   Y : Boolean;
   Z : I7;          -- type I7 is range 0 .. 127
end record;
for R use record
   X at 0 range 0 .. 7;
   Y at 1 range 0 .. 0;
   Z at 1 range 1 .. 7;
end record;
```

Listing 1: Elementary record representation clause

An elementary example is given in listing 1.

These record declaration and record representation clause declare a data structure occupying two storage elements (which are assumed to be 8-bit bytes throughout this discussion) which is thus laid out:

- the first component, X, is an 8-bit character that fits exactly in the storage element at offset 0
- the second and third components, a 1-bit Boolean value Y and a 7-bit integer value Z, share space in the second storage element at offset 1. Y uses one bit numbered "0", and Z uses the remaining seven bits, numbered "1" thru "7".

It is important to note that the semantics of this very simple fragment of code in terms of data representation is different, depending on the bit numbering convention used by a particular compiler:

	Low order first	**High order first**
X	first byte of representation	
Y	Least sig. bit of 2nd byte	Most sig. bit of 2nd byte
Z	*Shift_Right* (2nd byte, 1)	2nd byte *and* 2#0111_1111#

2.2 Endianness neutral representation clauses

Tricks have been proposed to express such clauses in a neutral way, so that little or no code modifications are required to obtain the same representation when porting between a big-endian and a little-endian platform [4, 8]. These essentially consist in having an integer constant whose value reflects the endianness of the platform, and expressing all component positions and bit ranges as arithmetic expressions depending on this constant.

Ada 95 introduced an alternative to these tricks, in the form of the representation attribute *Bit_Order* (Ada 95 Reference Manual section 13.5.3 [5]). When this attribute is defined for some record type, a record representation clause for the type is interpreted using the specified bit numbering convention. In the above example, one can thus specify:

```
for R'Bit_Order use System.Low_Order_First;
```

The effect of this attribute definition clause is that the bit numbers in the record representation clause will always be interpreted as on a little-endian machine. The memory representation of objects of type R therefore becomes independent of the machine endianness.

2.3 Crossing byte boundaries

The semantics of bit positions greater than the number of bits in a storage element is pretty clear when using the default (system) bit order. The situation becomes more confused in the opposite case, and this caused a binding interpretation of the Ada 95 standard to be issued [1] to clarify the meaning of a record representation clause in that case.

The rule as clarified in Ada 2005 (and retrospectively in Ada 95 by virtue of this binding interpretation) is that operations on record components can only extract information from contiguous bit ranges taken from some machine integer (what Ada 2005 calls "machine scalars"). This makes sense because this reflects the requirement that extracting a record component is performed using load, store, shift, and mask operations of the underlying machine architecture. This constraint limits the spectrum of data layouts that can be specified in an endianness independent way.

This limitation becomes clear when one considers the following data type, together with its representation clause:

```
subtype Yr_Type is Natural range 0 .. 127;
subtype Mo_Type is Natural range 1 .. 12;
subtype Da_Type is Natural range 1 .. 31;
subtype Ho_Type is Natural range 0 .. 23;
subtype Mi_Type is Natural range 0 .. 59;

subtype S2_Type is Natural range 0 .. 29;
-- Two seconds unit

type Date_And_Time is record
   Years_Since_1980 : Yr_Type;
   --  Bits Y0 (most significannt)
   --    to Y6 (least significant)

   Month            : Mo_Type;
   --  Bits M0 (most significannt)
   --    to M3 (least significant)

   Day_Of_Month     : Da_Type;
   --  Bits D0 (most significannt)
   --    to D4 (least significant)
```

```
   Hour              : Ho_Type;
   Minute            : Mi_Type;
   Two_Second        : S2_Type;
end record;

for Date_And_Time use record
   Years_Since_1980 at 0 range 0  .. 6;
   Month            at 0 range 7  .. 10;
   Day_Of_Month     at 0 range 11 .. 15;
   Hour             at 2 range 0  .. 4;
   Minute           at 2 range 5  .. 10;
   Two_Second       at 2 range 11 .. 15;
end record;
```

Listing 2: Record with components crossing byte boudaries

The data for the first three components, as described by the above representation clause, is stored as two storage elements, as shown on figure 1.

	Most sig. bit ⋯ Least sig. bit							
Byte 0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	M_0
	0	1	2	3	4	5	6	7
Byte 1	M_1	M_2	M_3	D_0	D_1	D_2	D_3	D_4
	8	9	10	11	12	13	14	15

(a) Big-endian

	Most sig. bit ⋯ Least sig. bit							
Byte 0	M_3	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
	7	6	5	4	3	2	1	0
Byte 1	D_0	D_1	D_2	D_3	D_4	M_0	M_1	M_2
	15	14	13	12	11	10	9	8

(b) Little-endian

Fig. 1: Date and time structure (first two bytes)

To extract components using shift and mask operations, this data must be considered as a single (16-bit) integer value, whose bits are numbered from 0 (MSB) to 15 (LSB) or 0 (LSB) to 15 (MSB), depending on whether Bit_Order is High_Order_First or Low_Order_First (figure 2).

Byte 0								Byte 1							
Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	M_0	M_1	M_2	M_3	D_0	D_1	D_2	D_3	D_4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Byte 1								Byte 0							
D_0	D_1	D_2	D_3	D_4	M_0	M_1	M_2	M_3	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fig. 2: Date and time structure as a 16-bit scalar (BE top, LE bottom)

As an example, if the date is November 18, 2012, the values for the first three components is `(Years_Since_1980 => 32, Month => 11, Day => 18)`, and on a High_Order_First machine the bit pattern is as shown on figure 3. The corresponding sequence of storage elements is `(65, 114)`, and the corresponding scalar value is $65 \times 256 + 114 = 16\,754$.

MSB(byte 0) .. LSB(byte 0)								MSB(byte 1) .. LSB(byte 1)							
Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	M_0	M_1	M_2	M_3	D_0	D_1	D_2	D_3	D_4
0	1	0	0	0	0	0	1	0	1	1	1	0	0	1	0
65								114							
16 754															

Fig. 3: Bit pattern for Nov. 18, 2012 on a big-endian machine

	MSB(byte 1) .. LSB(byte 1)								MSB(byte 0) .. LSB(byte 0)							
Original data	M_1	M_2	M_3	D_0	D_1	D_2	D_3	D_4	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	M_0
Example bits	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	1
Byte values	114								65							
Scalar value	29 249															

Fig. 4: Bits of big-endian structure from fig. 3, as seen on a little-endian machine

Now if the same memory region is accessed on a *little-endian* machine as a 16-bit integer, the binary value now is as shown on figure 4. For our example case the original bit pattern now translates to scalar value $114 \times 256 + 65 = 29\,249$. It should be noted that the Month field is not contiguous anymore in this representation: bit 0 ends up at position 0 (least significant bit), where as bits 1 to 3 end up at positions 13 to 15. More generally, on big endian machines the least significant bit of one storage element is adjacent to the most significant bit of the next one when considering a machine scalar, whereas on a little endian machine it is the most significant bit of the first byte that is adjacent with the least significant bit of the following one.

As a result, *no record component representation clause in standard Ada can describe the LE layout on a BE platform.*

The Bit_Order attribute changes the way indices are assigned to bits (i.e. in the above integer, in High_Order_First ordering the bits are denoted with indices 0 .. 15, whereas in Low_Order_First they are numbered 15 .. 0). In other words, the only effect of setting `Date_And_Time' Bit_Order` to `Low_Order_First` is to change the bit numbering from $0 \cdots 15$ to $15 \cdots 0$. This does not change the order in which a CPU load operation takes bytes from memory to build an integer value in a register, on which shift and mask operations are applied to extract an individual component.

Note that we arbitrarily chose to consider just the first three components, and the corresponding underlying *16 bit* scalar, but we could just as well have considered the complete structure and the associated 32 bit scalar: the bits of the `Month` component would have been separated in the same way.

This situation is encountered anytime a record component crosses a storage element boundary. In this case no standard representation clause can be written that will yield identical representations on big-endian and little-endian machines: additional work is then required to access such data structures. Several approaches are discussed in the remainder of this paper.

3 Explicit byte reordering

3.1 Individual component swapping

If each component in a record type occupies an integral number of storage elements, then the extraction of the component's bits from the enclosing data structure does not require any shifting and masking operation; the component's underlying storage itself is a machine scalar, and the only remaining issue when accessing the components is the ordering of bytes within the component itself. In other words, in this case storage elements can be reordered after extracting the component from the struture according to a record representation clause, and the reordering operation can be considered at the level of the value of a single component. (In contrast with cases such as the example discussed above, where components did not occupy integral storage elements, and reordering operations were necessary even to just gather the bits constituting a single component).

This simple situation is encountered for example when writing code that binds directly to the standard BSD sockets API, where all data structures are traditionally big-endian; byte-swapping functions `htons/ntohs` and `htonl/ntohl` are provided by the standard API to perform byte swapping (respectively for short and long integer values) when operating on little-endian platforms (these operations are nops on big-endian platforms).

The GNAT run-time library includes a set of generic procedures to perform byte swapping in package *GNAT.Byte_Swapping*. This package provides a set of generic byte swapping subprograms for 16, 32, and 64-bit objects. These take advantage of GCC builtins to perform byte swapping operations and which use dedicated, efficient CPU instructions where available.

When using explicit byte swapping at the component level, care must be taken by the programmer to identify whether a given value has been byte swapped or not at any given point in time. This means a strict isolation is desirable between the data types used for input/output (or interaction with standard library calls), which require data in the externally mandated byte order, and data structures used for internal processing (where components need to be in their correct native order). When retargeting legacy code that was not written with portability in mind in the first place, such isolation may be found wanting.

3.2 Arithmetic component extraction

Another alternative is for the user to extract component values from storage elements using explicit arithmetic operations on raw storage arrays. For example, suppose that `SE (0)` and `SE (1)` are the first and second storage elements of the date/time structure from listing 2 as stored on a big-endian machine. The following expressions can be used to extract the `Year` and `Month` components in a platform independent way:

```
Year  := SE (0) / 2;
Month := (SE (0) and 1) * 8 + SE (1) / 32
--         ^^^   M0    ^^^        ^^M1 .. M3^^
```

This way of expressing data layout is independent of endianness, and as such ensures maximum portability. However it is a cumbersome notation, reducing the legibility, and hence the maintainability, of application code. It also hinders optimization by the code generator, by pushing detailed representation information up to the highest levels of the intermediate representations handled by the code generator. Moreover, in this case again there must be a strict separation between the raw arrays of storage elements used for external operations, and the native byte order data structures used internally by the software.

3.3 Wholesale byte reversal

An interesting solution has been proposed by R. Andress in [2], where he suggests to revert the order of storage elements constituting a given data structure *as a whole*, and to then construct a new record representation clause mapping components on the reversed structure. He observes that when changing platform endianness endianness, the complete reversal of byte order makes all components that crossed byte boundaries contiguous again. One can then construct a new representation clause that locates each component within the reversed structure.

This approach is elegant and expressive; it has the merit of minimal intrusiveness on existing code. On the other hand it still requires an explicit byte reordering operation, and the storage of data structures in two copies (one in original order, and the other in reversed order). The representation clause for the reversed structure also needs to be carefully written and maintained up-to-date with respect to the original one.

3.4 Data modeling approaches

An important drawback of the manual byte reordering approaches discussed above is the verbosity of notations for arithmetic component extraction or record representation clauses. One way of alleviating such a concern is to replace manually implemented code with code generated from a model.

In the context of endianness conversions, the model is a formal description of the bit layout of some data structure, and the operations provided by the generated code are accessors to the components of that structure.

The idea is akin to that of the ASN.1 standard [6]. However in ASN.1 one describes a data structure in an abstract semantics perspective. This description can then be mapped to one (or more) concrete representations through some standard *encoding rules* [7].

In the case of externally mandated data representations, on the contrary, the model starts by describing the exact structure in terms of bits and bytes, and from there describes how these elementary pieces of data must be interpreted to form higher level values.

This is the approach we followed in *Tranxgen*, a code generation tool we introduced while developing a portable, certifiable TCP/IP stack. A similar path has been followed by existing tools for other languages [9].

Tranxgen accepts a data structure description description in the form of an XML document, and produces a set of Ada (more specifically, SPARK 95) accessors for the data structure.

```
<package name="Date_And_Time_Pkg">
 <message name="Date_And_Time">
  <field name="Years_Since_1980" length="7" />
  <field name="Month"            length="4" />
  <field name="Day_Of_Month"     length="5" />
  <field name="Hour"             length="5" />
  <field name="Minute"           length="6" />
  <field name="Two_Seconds"      length="5" />
 </message>
</package>
```

Listing 3: Tranxgen specification for date/time record

From this specification, *Tranxgen* produces a record type declaration with representation clause, none of whose components crosses a byte boundary. Accessors decompose and reconstruct component values using arithmetic expressions, following the method outlined in section 3.2, as seen in the following generated code excerpt:

```
package Date_And_Time_Pkg is
   type Date_And_Time is record
      Years_Since_1980 : U7_T;   -- 7 bits
      Month_0          : Bits_1; -- 1 bit integer
      Month_1          : Bits_3; -- 3 bits
      Day_Of_Month     : U5_T;   -- 5 bits
      Hour             : U5_T;   --   ""
      Minute_0         : Bits_3; -- 3 bits
      Minute_1         : Bits_3; --   ""
      Two_Seconds      : U5_T;   -- 5 bits
   end record;

   for Date_And_Time'Alignment use 1;
```

```
   for Date_And_Time'Bit_Order
     use System.High_Order_First;
   for Date_And_Time use record
      Years_Since_1980 at 0 range 0 .. 6;
      Month_0          at 0 range 7 .. 7;
      Month_1          at 1 range 0 .. 2;
      Day_Of_Month     at 1 range 3 .. 7;
      Hour             at 2 range 0 .. 4;
      Minute_0         at 2 range 5 .. 7;
      Minute_1         at 3 range 0 .. 2;
      Two_Seconds      at 3 range 3 .. 7;
   end record;

end Date_And_Time_Pkg;

package body Date_And_Time_Pkg is

   --  [...]

   function Month (P : System.Address) return U4_T is
      M : Date_And_Time;
      for M'Address use P;
      pragma Import (Ada, M);
   begin
      return U4_T (M.Month_0) * 2**3 + U4_T (M.Month_1);
   end Month;

   procedure Set_Month (P : System.Address; V : U4_T) is
      M : Date_And_Time;
      for M'Address use P;
      pragma Import (Ada, M);
   begin
      M.Month_0 := Bits_1 (V / 2**3);
      M.Month_1 := Bits_3 (V mod 2**3);
   end Set_Month;

   --  [...]

end Date_And_Time_Pkg;
```

4 The *Scalar_Storage_Order* attribute

In this section we introduce the *Scalar_Storage_Order* attribute, which allows the specification of the storage endianness for the components of a composite (record

or array) type. Byte reordering is performed transparently by compiler generated code upon access to elementary (scalar) components of the data structure.

4.1 Formal definition

Scalar_Storage_Order is an implementation-defined attribute specified as follows by the GNAT Reference Manual.

> For every array or record type S, the representation attribute `Scalar_Storage_Order` denotes the order in which storage elements that make up scalar components are ordered within S. Other properties are as for standard representation attribute `Bit_Order`, as defined by Ada RM 13.5.3(4). The default is `System.Default_Bit_Order`.
> For a record type S, if *S*`'Scalar_Storage_Order` is specified explicitly, it shall be equal to *S*`'Bit_Order`.

This means that if a `Scalar_Storage_Order` attribute definition clause is not confirming (that is, it specifies the opposite value to the default one, `System.Default_Bit_Order`), then the type's `Bit_Order` shall be specified explicitly and set to the same value. Also note that a scalar storage order clause can apply not only to a record type (like the standard `Bit_Order` attribute), but also to an array type (of scalar elements, or of other composite elements).

> If a component of S has itself a record or array type, then it shall also have a `Scalar_Storage_Order` attribute definition clause. In addition, if the component does not start on a byte boundary, then the scalar storage order specified for S and for the nested component type shall be identical.
> No component of a type that has a `Scalar_Storage_Order` attribute definition may be aliased.

These clauses ensure that endianness does not change except on a storage element boundary, and that components of a composite with a *Scalar_Storage_Order* attribute are never accessed indirectly through an access dereference (instead all accesses are always through an indexed or selected component).

> A confirming `Scalar_Storage_Order` attribute definition clause (i.e. with a value equal to `System.Default_Bit_Order`) has no effect.
> If the opposite storage order is specified, then whenever the value of a scalar component of an object of type S is read, the storage elements of the enclosing machine scalar are first reversed (before retrieving the component value, possibly applying some shift and mask operatings on the enclosing machine scalar), and the opposite operation is done for writes.

This is where the new attribute introduces the byte reordering.

The following clause generalizes the notion of *machine scalar* to cover some useful cases not taken into account by the original wording of the standard in the definition of the underlying machine scalar of a given (scalar) component.

In that case, the restrictions set forth in 13.5.1(10.3/2) for scalar components are relaxed. Instead, the following rules apply:

- the underlying storage elements are those at `(position + first_bit / SE_size) .. (position + (last_bit + SE_size - 1) / SE_size)`
- the sequence of underlying storage elements shall have a size no greater than the largest machine scalar
- the enclosing machine scalar is defined as the smallest machine scalar starting at a position no greater than `position + first_bit / SE_size` and covering storage elements at least up to `position + (last_bit + SE_size - 1) / SE_size`
- the position of the component is interpreted relative to that machine scalar.

4.2 Example usage and effect

Let us assume that the type declaration for the `Date_And_Time` structure in listing 2 is from a legacy big-endian application, a component of which is now being retargeted to a new little-endian board. Of course the underlying memory representation must not be changed, as this board exchanges messages with a legacy black-box module whose source code is unavailable. We will therefore apply attribute definition clauses as follows as shown in listing 4.

```
for Date_And_Time'Scalar_Storage_Order
  use System.High_Order_First;

for Date_And_Time'Bit_Order use System.High_Order_First;
-- Bit order and scalar storage order must be consistent.
```

Listing 4: Attribute definition clauses for scalar storage order and bit order

The memory representation of an object of type `Date_And_Time` as created on a big-endian machine is shown on figure 1. If we read the `Month` component of an object of that type, we first load the underlying machine scalar. As noted above, the storage elements have values 65 followed by 114, and on a little-endian machine this represents the 16-bit scalar value 29 249.

But now by virtue of the *Scalar_Storage_Order* that has been defined for the type as `High_Order_First`, since we are on a little-endian machine we reverse the order of bytes within this machine scalar, which gives us back the original value 16 754. The bit pattern of this scalar is now identical to the original one from the big-endian specification, and thus shifting and masking operations will yield the original component value.

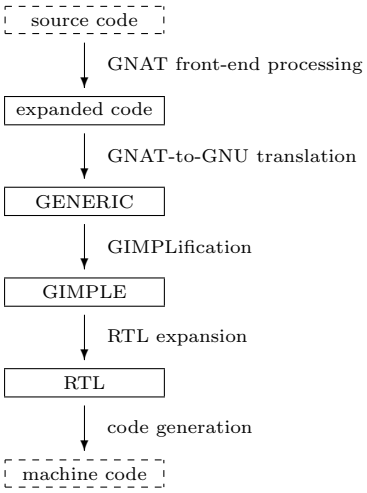
The reverse byte swapping is performed upon write operations, after the component value has been installed into a machine scalar, and prior to this machine scalar being stored back to memory. This transformation is applied only for scalar components, so that nested records are handled correctly (i.e. no extra swapping is introduced when accessing a subcomponent of a nested record).

4.3 Implementation

The implementation of the *Scalar_Storage_Order* attribute has been done in the version of the GNAT compiler using the GCC back-end as the code generator.

Even if a growing number of processors have the capability to run either in big-endian or in little-endian mode, the mode is generally selected once for all at startup and cannot be changed afterwards. The compiler therefore needs to generate explicit byte swapping operations.

The primary design decision to be considered is the level at which these byte swapping operations are made explicit in the hierarchy of intermediate representations of the compiler. The GCC-based GNAT compiler has a 4-tiered hierarchy of representations (the framed boxes):



The higher in the hierachy the byte swapping operations are made explicit, the simpler the implementation is, but the less efficient the machine code will be when run on the target. This is because the bulk of the compiler is parameterized for the endianness of the target, and so explicit byte swapping operations act as optimization barriers in the various intermediate representations.

The choice has been made to generate the byte swapping operations during RTL expansion: the first intermediate representation in which they are explicitly present is RTL (Register Transfer Language), which is a very low-level representation. All high-level GIMPLE optimizations, which are the most powerful ones, work without change on code requiring endianness conversions; only the low-level RTL optimizations are affected.

The other major design decision pertains to the representation and the manipulation of the storage order in the GIMPLE representation (the expanded code and the GENERIC representation being essentially extremely verbose versions of the source code, they do not need substantial adjustments; at these levels, the scalar storage order is just another property of composite types, like

```
movzwl  (%ecx), %eax
rolw    $8, %ax
shrw    $5, %ax
andl    $15, %eax
```

Listing 5: x86 assembly code generated for load of Month

packedness or atomicity). Storage order could conceivably be tracked on a scalar-by-scalar basis, i.e. with the finest possible granularity. With this approach, every scalar gets a new property, the endianness, in addition to the usual properties, for example the size and the bounds. However this would have required major surgery in the high-level part of the code generator, and would have introduced an undesirable additional layer of complexity.

We therefore chose instead to consider storage order only as a property of memory stored scalars (and specifically, only those scalars stored as part of an enclosing composite object); all other scalars always have the default storage order. Moreover, scalar *values* (considered outside of any *object*) are always in the default endianness. This makes the implementation far simpler, because the various transformations and optimizations applied to the intermediate representation need not take the endianness into account. They only have to preserve the invariant that some particular scalars in memory must be accessed in a special way.

It is also worth noting that, in a few specific cases, the GNAT front-end needs to apply low-level transformations to the source code before passing it to the code generator, which may depend on the storage order. In these cases, the front-end may need to generate explicit byte swapping operations (for example to initialize bit-packed arrays). The code generator therefore exposes its internal byte swapping primitives as builtins that can be directly invoked by the front-end. These are ultimately translated into explicit byte swapping operations during RTL expansion.

The implementation is fully generic: it imposes no additional requirement on the target architecture, such as availability of specific byte swapping or byte manipulation instructions. However, the code generator will take advantage of them if present, for example on the Intel x86 and IBM PowerPC architectures, with a measurable performance gain in these cases.

Going back to the example of the `Month` component of an object of type `Date_And_Time`, the assembly code generated on x86 to read the component is shown on listing 5.

The first instruction `movzwl` loads the underlying machine scalar, i.e. the 16-bit integer value at offset 0 in the record. The second instruction `rolw $8` swaps the bytes in this scalar. The remaining instructions extract the component.

4.4 Performance discussion

The introduction of the *Scalar_Storage_Order* attribute represents a significant gain in terms of application development cost, in that in relieves developers from the need to implement data conversions from one endianness to the other.

However the execution time cost of such conversions does not disappear: they are still present, and if opposite-endianness data structures are frequently used, they are liable to cause an unavoidable degradation in application performance (compared to the same application using data structures in native endianness). This is less likely when explicit data conversions are used, because in the latter case conversion points are well identified, and internal processing in the application is done efficiently on data structures that have native endianness.

It may therefore be advisable, even when relying on *Scalar_Storage_Order* to perform data conversions, to apply this attribute to a derived type used for external interfaces. Ada type conversions from the derived type to the ancestor type (which has no representation attributes, and hence has the standard native representation) can then be used to convert data from the external representation (possibly using a different endianness than the native one) to the internal (native endianness) representation, as show on listing 6.

```
type Date_And_Time is record
   ...
end record;
--  Native, efficient representation

type External_Date_And_Time is new Date_And_Time;
for External_Date_And_Time use record
   ...
end record;
for External_Date_And_Time'Scalar_Storage_Order
  use External_Bit_Order;
for External_Date_And_Time'Bit_Storage_Order
  use External_Bit_Order;

function To_Internal
  (DT : External_Date_And_Time) return Date_And_Time
is
begin
   return Date_And_Time (DT);
   --  Type conversion with change of representation
end To_Internal;
```

Listing 6: Setting scalar storage order on a derived type

In this manner, developers retain the advantage of automated, transparent generation of the code effecting the required representation change, while at the same time avoiding the distributed overhead of pervasive byte swapping throughout the application.

5 Conclusion and future directions

We have presented the issues posed by data representations with different endianness in Ada applications. We have discussed the current Ada features supporting record layout specification, and identified some of their limitations in conjunction with support for endianness conversions. We have introduced two separate approaches to overcoming these limitations: a code generation tool *Tranxgen* producing accessors from a data representation model, and a new representation *Scalar_Storage_Order* allowing transparent access to data structures of arbitrary endianness. These tools allow application code to be written in a portable way, guaranteeing consistent data representations between little-endian and big-endian platforms without the need for explicit conversion operations.

Possible improvements to *Tranxgen* include support for a wider variety of data structures, and using the *Scalar_Storage_Order* attribute in generated code. The specification for the attribute will be proposed to the Ada Rapporteur Group for inclusion in the next revision of the Ada language.

References

1. Ada Rapporteur Group: Controlling bit ordering. Ada Issue AI95-00133, ISO/IEC JTC1/SC22/WG9 (2004), http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ais/ai-00133.txt?rev=1.17, adopted amendment to the Ada 95 standard [5]
2. Andress, R.P.: Wholesale byte reversal of the outermost Ada record object to achieve endian independence for communicated data types. Ada Letters XXV(3), 19–27 (Sep 2005)
3. Cohen, D.: On Holy Wars and a Plea for Peace. IEEE Computer 14(10), 48–54 (Oct 1981)
4. Cohen, N.H.: Endian-independent record representation clauses. Ada Letters XIV(1), 27–29 (Jan 1994)
5. ISO: Information Technology – Programming Languages – Ada. ISO (Feb 1995), ISO/IEC/ANSI 8652:1995
6. ITU-T: Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation. Recommendation X.680 (Nov 2008)
7. ITU-T: Information technology — ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Recommendation X.690 (Nov 2008)
8. Mardis, M.: Endian-safe record representation clauses for Ada programs. Ada Letters XIX(4), 13–18 (Dec 1999)
9. Protomatics: Transfer Syntax Notation One (TSN.1). Tech. rep., http://www.protomatics.com/tsn1.html
10. Swift, J.: Travels into Several Remote Nations of the World. By Lemuel Gulliver. (1726)