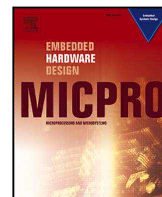
Microprocessors and Microsystems 78 (2020) 103268

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



Address-encoded byte order

David Guerrero *, German Cano-Quiveu, Jorge Juan-Chico, Alejandro Millan, Manuel J. Bellido, Julian Viejo, Paulino Ruiz-de-Clavijo, Enrique Ostua

Departamento de Tecnología Electrónica, Universidad de Sevilla, 41012 Sevilla, Spain

ARTICLE INFO

Keywords:
ISA
Data alignment
Byte order
Endianness
Shared memory
MPSoC

ABSTRACT

Unaligned accesses are forbidden in many high-performance architectures. In most of these architectures, the least significant address bits of a multibyte memory access must be zero. Otherwise, the program generating the access is considered erroneous and an exception is flagged. The objective of this paper is to propose an alternative behaviour using the least significant address bits to encode the byte order of the accessed data. Modifying a traditional architecture to support the proposed behaviour presents several advantages, including backward compatibility at binary-code level, and the possibility of carrying out an endianness conversion during multibyte memory accesses without increasing the execution time nor using additional opcodes. The technique is demonstrated by modifying an OpenRISC 1000 implementation without introducing any penalty in hardware resources or performance. Subroutines written and compiled for the traditional architecture and originally designed for only the native byte order can, in the modified architecture, read and write data in a non-native byte order without any need to recompile. The execution of a sample algorithm operating on non-native byte order shows a reduction of 60% in the user execution time in the modified implementation when compared to the original implementation.

1. Introduction

In the memory model of most modern processors, memory locations are 8 bits wide, and therefore the minimum unit the processor can write or read from memory is a byte. The greatest number that can be represented by a byte using positional notation is $2^8 - 1 = 255$. To represent a greater integer, a processor must use a concatenation of bytes, known as *a word*. Using a single instruction, many processors can read or write a word greater than a byte, although the size of that word in bytes must usually be power of 2. Typical sizes for access are 2, 4, and 8 bytes, but many vector processors support larger sizes [1]. The designer of an architecture must decide two main issues when the size of a memory access can be greater than the size of the memory locations. One is *data alignment*, which determines the addresses permitted for accesses that are wider than a memory location. The other is *endianness*, which maps the memory locations involved in each access to chunks of the same size of the accessed word. Since this size is usually a byte, endianness is also referred to as *byte order*. For example, suppose that the word includes at least two bytes, Bm and Bl, and that the bits of Bm are more significant than the bits of Bl: In the *little-endian* byte order, Bm will be stored in a higher memory location than Bl, while in the *big-endian* byte order, Bm will be stored in a lower memory location than Bl [2,3]. Most architectures use one of the previous orders. For example, a picoJava processor uses the little-endian byte order [4], while the SPARC V8 architecture uses the big-endian byte order [5]. Several architectures, called bi-endian [6], use both. However, a few use other byte orders called *mixed-endian* (or *middle-endian*). In a mixed-endian order, the most significant half of a word is stored immediately after or immediately before the least significant half, depending on the size of that word. For example, in the vintage PDP-11 processor, words of two bytes are stored in little-endian order, that is, the most significant byte is stored after the least significant byte. If this processor stores a word of four bytes, then the word is split into two subwords of two bytes and the most significant subword is stored before the least significant subword, whereby each two-byte subword is still stored in little-endian order.

The communication between systems that use different byte orders is problematic [7]. Nowadays, heterogeneous Multiprocessor System-on-Chip (MPSoC) can include several processors with different endianness and therefore these problems can arise even when the systems are on the same chip [8–10]. For example, if a processor has to submit an integer to another processor with a different byte order through shared memory, then a protocol must be established. The first processor

* Corresponding author.
E-mail addresses: guerre@dte.us.es (D. Guerrero), germancq@dte.us.es (G. Cano-Quiveu), jjchico@dte.us.es (J. Juan-Chico), amillan@us.es (A. Millan), bellido@dte.us.es (M.J. Bellido), julian@us.es (J. Viejo), pruiz@us.es (P. Ruiz-de-Clavijo), ostua@dte.us.es (E. Ostua).

https://doi.org/10.1016/j.micpro.2020.103268
Received 2 December 2019; Accepted 10 September 2020
Available online 12 September 2020
0141-9331/© 2020 Elsevier B.V. All rights reserved.

could permute the bytes used to code the integer before writing them in accordance with the order used by the second processor. Alternatively, the first processor could write the bytes using its native order and the second should carry out the permutation after reading said bytes. Similar problems arise when dealing with files. Not all the file formats use the same byte order. If a system must read or write files with a format that uses a different byte order, then part of the executed instructions must be employed to reorder bytes [11]. Byte reordering also implies a severe penalty in emulators if the endianness of the guest and the host systems differ [12,13]. This penalty can be considerably reduced in architectures that have instructions to reorder the bytes of a register, such as the Intel 486 [14], although the penalty is not completely removed. It may be deemed that this presents no problem in bi-endian architectures, but it should be taken into account that, in certain architectures, application processes cannot change the endianness. This means that application processes must call the operating system to use the alien byte order. Worse still, library functions will probably be designed to use the native byte order and hence application processes will have to carry out a system call to switch to the native endianness before calling each function and then carry out another system call after every function call to return to the alien endianness. Similarly, if a function is going to use a byte order that differs from that used in the main code, then a system call must be carried out at the beginning of the function and another call before returning. Of course, each system call implies a severe overhead. In other bi-endian architectures, application processes can change the bit of the configuration register employed to set the endianness and hence they can switch the byte order with very little penalty [15]. In certain processors, the operation code specifies the endianness to be used and hence the penalty can be completely removed. For example, the native byte order of an x86 processor is little-endian, but if its Instruction Set Architecture (ISA) includes the MOVBE instruction, then it can carry out big-endian memory accesses [13]. Obviously, this instruction has its own operation code. Analogously, including instructions to support other mixed-endian byte orders would require additional operation codes. The purpose of this paper is to introduce an efficient way to encode and implement multi-endian support in architectures with data alignment restrictions.

The rest of the paper is organized as follows. In the next section, byte order and data alignment are formally defined. In Section 3, the proposed functionality is described. In Section 4, implementation details are discussed. Section 5 details how to write software to take advantage of the introduced functionality. Experimental performance results are shown in Section 6. The last section presents a summary of the conclusions.

2. Background

Hereinafter, we will use the following notation to describe a memory access:

- W: Word to be read or written.
- N: Size of W in bytes. In this paper we will assume this to be a power of 2.
- t: Logarithm of N to base 2. Hence $N = 2^t$.
- x: An element of $\mathbb{Z}_N = \{0, 1, 2, \ldots, N-1\}$.
- x_i: ith bit of the binary (base 2) representation of x. Hence $x = \sum_{i=0}^{t-1} x_i 2^i$.
- W_i: ith bit of W, whereby the concatenation of the bits W_{8N-1}, $\ldots$, W_1 and W_0 is W.
- B_x: concatenation of the bits W_{8x+7}, $\ldots$, W_{8x+1} and W_{8x+0}, hence the concatenation of the bytes B_{N-1}, $\ldots$, B_1 and B_0 is W.
- A: Address of the lowest memory location employed to store W.
- A_i: ith bit of the binary (base 2) representation of A.

Additionally, we will use the prefix $ for hexadecimal numerals. In the following subsections, we will detail decisions that must be made when designing an architecture whenever the size of the memory locations and the size of the words read or written may differ.

Table 1
Permutation used by the PDP-11 in four byte accesses.

x	0	1	2	3
$P_{\text{PDP4B}}(x)$	2	3	0	1

Table 2
Memory dumps of systems using different processors.

Address	$A+0$	$A+1$	$A+2$	$A+3$
Data picoJava	$40	$30	$20	$10
Data SPARC V8	$10	$20	$30	$40
Data PDP-11	$20	$10	$40	$30

2.1. Byte order

In most architectures, to access a word W of length N, the address A of the lowest memory location to be read or written must be provided. For example, if it is a write access, the component bytes $B_0, B_1, \ldots, B_{N-1}$ of W will be stored in the memory locations $A + 0, A + 1, \ldots, A + N - 1$, but not necessarily in that order. The byte order convention of the architecture maps those bytes to the memory locations. More formally, each byte B_x will be read or written in the memory location $A + P_N(x)$, where P_N is a permutation of $\mathbb{Z}_N$ as determined by the architecture. The most widely used permutations include the following:

- Identity function on $\mathbb{Z}_N$: This permutation is defined by $id_{\mathbb{Z}_N}(x) = x$.
- $(N-1)$'s complement: This permutation is defined by $C_{N-1}(x) = N - 1 - x$.

Little-endian architectures use the first permutation, while big-endian architectures use the second. The mixed-endian permutation used by the PDP-11 in four-byte accesses, which we denote as P_{PDP4B}, is shown in Table 1. As an example, Table 2 shows the content of the accessed memory locations of computers using a PicoJava processor (little-endian), a SPARC V8 processor (big-endian), and a PDP-11 processor (mixed-endian) immediately after storing the word $10203040 at address A.

Although other mixed-endian orders are rarely used, we will define them formally in order to explain the contribution introduced in this paper. In order to specify a mixed-endian order, it is necessary to set, for every word size greater than a byte, whether the most significant half of the word must be stored before or after the least significant half. Thus, if the size of the word is $N = 2^t$ (with $t > 0$) it is necessary to ascertain:

1. whether the most significant half of each word of size 2^1 must be stored before or after the least significant half.
2. whether the most significant half of each word of size 2^2 must be stored before or after the least significant half.

   ⋮

t. whether the most significant half of each word of size 2^t must be stored before or after the least significant half.

If the same decision is made for every word size, then the resulting order will be little-endian or big-endian and therefore little-endian and big-endian are particular cases of mixed-endian orders. In general, since for every size there are two options, the permutation of a mixed-endian order can be defined by a string of t bits. Let $m_{t-1} \ldots m_1 m_0$ be that string, the meaning of each bit m_i can be chosen arbitrarily. For example, the following convention can be used:

- $m_i = 0$: the most significant half of each word of size 2^{i+1} is stored in higher memory locations than those of the least significant half (as in little-endian).

- $m_i = 1$: the most significant half of each word of size 2^{i+1} is stored in lower memory locations than those of than the least significant half (as in big-endian).

We denote the permutation specified by the string $m_{t-1} \dots m_1 m_0$ using this convention as $Pl(m_{t-1}..m_1 m_0)$, since if every $m_i = 0$, then the little-endian order is set. This permutation can be computed with the formula

$$Pl(m_{t-1} \dots m_1 m_0)(x) = \sum_{i=0}^{t-1} (m_i \oplus x_i) 2^i \tag{1}$$

where $\oplus$ is the XOR operator and x_i is the bit at position i of the binary representation of x. This is simply a formal way to state that the binary representation of $Pl(m_{t-1} \dots m_1 m_0)(x)$ is a bitwise XOR of $m_{t-1} \dots m_1 m_0$ and $x_{t-1} \dots x_1 x_0$. The reason is simple: if $m_i = 0$ then the storing order of the two halves of each word of size 2^{i+1} cannot be different from the little-endian order and therefore the ith bits of the binary representations of x and $Pl(m_{t-1} \dots m_1 m_0)(x)$ must be equal; otherwise they must be different. An alternative convention can assign the opposite meaning to each bit m_i of the string $m_{t-1} \dots m_1 m_0$, that is:

- $m_i = 1$: the most significant half of each word of size 2^{i+1} is stored in higher memory locations than those of the least significant half (as in little-endian).
- $m_i = 0$: the most significant half of each word of size 2^{i+1} is stored in lower memory locations than those of the least significant half (as in big-endian).

We denote the permutation specified by the string $m_{t-1} \dots m_1 m_0$ using the second convention as $Pb(m_{t-1} \dots m_1 m_0)$, since if every $m_i = 0$, then the big-endian order is set. This permutation can be computed with the formula

$$Pb(m_{t-1} \dots m_1 m_0)(x) = \sum_{i=0}^{t-1} (m_i \odot x_i) 2^i \tag{2}$$

where $\odot$ is the XNOR operator. Again, this is a formal way to state that the binary representation of $Pb(m_{t-1} \dots m_1 m_0)(x)$ is a bitwise XNOR of $m_{t-1} \dots m_1 m_0$ and $x_{t-1} \dots x_1 x_0$. For example, the permutation used by the PDP-11 in four-byte accesses can be described by the string $m_1 m_0 = 10$ using the first convention, or by the string $m_1 m_0 = 01$ using the second convention. Note that $P_{\mathrm{PDP4B}} = Pl(10) = Pb(01)$ as shown in Table 1.

2.2. Data alignment

By definition, a memory access to a word W of N bytes at address A is aligned if and only if A is multiple of N. The problems associated to unaligned accesses arise when designing the interconnection of the load/store units of a processor with the memory system. Such interconnection is exemplified in Fig. 1. The system in this example uses little-endian order, has addresses of P bits and general purpose registers of 2^r bytes with $r = 2$, although it can be easily extrapolated to other byte orders and values of r. When a word is read or written, the load/store unit provides to the memory system all the address bits except the r least significant, i.e. $A_{p-1} A_{p-2} \dots A_3 A_2$. The memory system can provide simultaneous access to the memory locations at the consecutive addresses $A_{p-1} \dots A_2 00$, $A_{p-1} \dots A_2 01$, $A_{p-1} \dots A_2 10$ and $A_{p-1} \dots A_2 11$. The load/store units generates the respective control signal EN_{00}, EN_{01}, EN_{10} and EN_1 to tell the memory system which of those four bytes memory locations will be accessed. Internally, the load/store units uses the r least significant address bits, i.e. A_1 and A_0, to map the accessed memory locations to the bytes of the word W to be read or written. For example, if a single byte at a logical address A such that $A_1 = 1$ and $A_0 = 0$ is accessed, then the data lines corresponding to the memory location $A_{p-1} \dots A_2 10$ will be connected to B_0, i.e. the bits $W_{7\dots0}$ of W, and only the control signal EN_{10} will be activated. The same happens if a word of two bytes at the same address is accessed, but additionally the data lines corresponding to the memory location

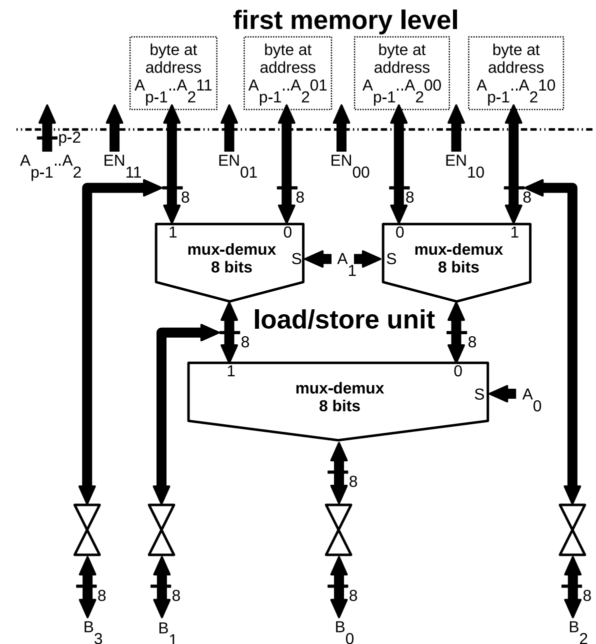


Fig. 1. Interconnection of a load/store unit with memory.

$A_{p-1} \dots A_2 11$ will be connected to B_1, and the control signal B_{11} will be also activated. In general, in an aligned access to a word of size $N = 2^t$, the logical address A of the lowest memory location of those employed to store the word is a multiple of 2^t, that is, the t least significant bits of this logical address are zero. Therefore, all the addresses in the range $[A, A + N - 1]$ match in the $P - t$ most significant bits. Since $t \le r$, this implies that the $P - r$ most significant bits of the logical addresses of the memory locations employed to store the word are the same and can be accessed simultaneously. From the load/store unit point of view, this involves a single access to a N-byte width memory.

In order to exemplify why unaligned accesses are more problematic, suppose that the load/store unit of Fig. 1 must read a word of two bytes at address 3. The bytes of this word are stored in memory locations 3 and 4. Since the bits A_2 of the addresses 3 and 4 fail to match, the load/store unit has to carry out two consecutive memory accesses to read the whole word. In general, any unaligned access has to be split into sub-accesses. Hence, implementing unaligned accesses involves several drawbacks:

- Unaligned accesses are slower.
- The second sub-access necessary to carry out an unaligned access can produce a page fault. Taking this into account complicates page fault management.
- The atomic read–modify–write operations are more complex since they may require two additional memory sub-accesses.
- The implementation of unaligned accesses requires hardware resources.

For these reasons, many high performance processors do not implement unaligned accesses. As previously mentioned, the t least significant address bits $A_{t-1} \dots A_1 A_0$ of any aligned access to a word of 2^t bytes are zero. Hereinafter, these bits are denoted *LSA* bits. If a processor does not support unaligned accesses, its behaviour when any of the LSA bits is not zero must be specified. Two alternative behaviours have been used in existing architectures:

1. One option, used in the Altivec ISA extension [16], is to simply ignore the t LSA bits. Therefore, when the processor is instructed to access a word at address A, the starting address of

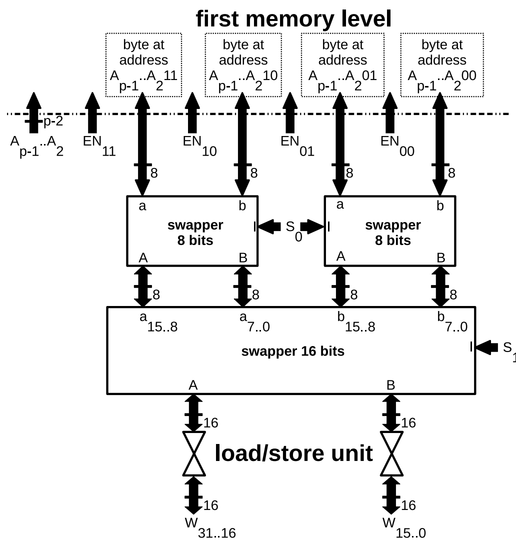


Fig. 2. Interconnection of a load/store unit implementing the proposed functionality with memory.

the word that will actually be accessed is $A-(A \bmod 2^t)$. An obvious advantage of this option is that it is easily implementable. Furthermore, no extra instructions are necessary to explicitly truncate (align-down) an address when handling addresses to the processor [17].

2. The most widely used option is to assume that a correct program will never produce such an access. The processor includes hardware to check that the t LSA bits are zero and, if any of these bits are one, then an exception is flagged to report that the program is erroneous.

In any of the previous options, when correct access to a word of 2^t bytes is carried out, the t LSA bits are meaningless and are not used. This paper introduces a third alternative behaviour by assigning semantics to those wasted bits that will be helpful when dealing with endianness conversion.

3. Semantics for the LSA bits

Our objective is to modify architectures that do not support unaligned accesses so that a LSA bit that is different from zero will no longer imply a programming error. Instead, it will indicate that the corresponding memory access must use a non-native byte order. We will define semantics for the LSA bits that will meet the following requirements:

1. If all the LSA bits are zero, then the memory access will use the byte order of the original architecture. This will ensure backward compatibility at binary-code level since no correct program written for the original architecture will produce a memory access with an LSA bit that differs from zero.
2. It must be possible to use several byte orders, including at least little-endian and big-endian, by choosing the values of the LSA bits of the memory access.
3. The modified architecture must be implementable without any time penalty with respect to the original architecture.

As in the Altivec ISA extension, when a word of size $N = 2^t$ is read or written using the effective address A, the starting address of the word will be $A - (A \bmod 2^t)$. However, in contrast to the Altivec ISA extension, the byte order will depend on the t LSA bits $A_{t-1} \dots A_1 A_0$. In particular, the permutation $Pl(A_{t-1} \dots A_1 A_0)$ will be used if the original architecture is little-endian, and the permutation $Pb(A_{t-1} \dots A_1 A_0)$ will be used if the original architecture is big-endian, where Pl and Pb are the functions defined by the Eqs. (1) and (2). Note that if every LSA bit is zero (i.e. $A_{t-1} \dots A_1 A_0 = 0 \dots 00$), then $Pl(A_{t-1} \dots A_1 A_0)(x) = x$ and $Pb(A_{t-1} \dots A_1 A_0)(x) = N-1-x$, and therefore the permutation of the original architecture is used and the first requirement is met. Furthermore, if every LSA bit is one (i.e. $A_{t-1} \dots A_1 A_0 = 1 \dots 11$), then $Pl(A_{t-1} \dots A_1 A_0)(x) = N-1-x$ and $Pb(A_{t-1} \dots A_1 A_0)(x) = x$ and hence the permutation used corresponds to big-endian (if the original architecture is little-endian) or little-endian (if the original architecture is big-endian) and the second requirement is met. Moreover, it is possible to carry out a memory access using any mixed-endian permutation. For example, in order to read or write a word of four bytes using the mixed-endian permutation P_{PDP4B} of the PDP-11, we simply have to make the LSA bits $A_1 A_0$ equal to 10 if the native byte order is little-endian, or 01 if the native byte order is big-endian.

Since no correct program written for the original architecture will produce a memory access with an LSA bit that differs from zero, the unaligned access exception can be eliminated in the modified architecture without losing binary-code compatibility. However, for debugging purposes, it could be desirable to introduce a way to mask that exception instead of eliminating it. For example, many architectures include special purpose registers to store information regarding the state of execution of the current process. Several bits of these registers often have no associated meaning and are reserved for future versions of the architecture. The proposed variation of the architecture may use one of these bits to specify whether the unaligned access exception should be raised when the value of an LSA bit is not zero. If the operating system keeps a separate value of this special purpose register for every process, then every process can enable or disable the exception separately. If the special purpose register can only be modified in supervisor mode, then an user process that wants to use a non native byte order would have to call the operating system to disable the unaligned access exception. However, this implies very little overhead since the user process only has to make the call once. The process does not need to make any other system calls to disable the new functionality before calling library functions written for the old version of the architecture, since these functions will work as before.

4. Implementation

The implementation of the proposed variation of an architecture is straightforward. It simply requires a modification of the interconnection of the load/store units with memory and simple changes in the exception handling. As an example, Fig. 2 shows a modification of the interconnection circuit of Fig. 1 to implement the new functionality. In general, if the general purpose registers of the modified architecture have a size of 2^r bytes, then the memory system can provide simultaneous access to 2^r consecutive memory locations. The load/store circuitry includes r layers of swappers labelled from 0 to $r-1$. Each swapper consists of two multiplexers/demultiplexers in parallel, as shown in Fig. 3. The set of swappers at level i will or will not swap both halves of each subword of size 2^{i+1} bytes depending on the value of the control signal S_i. If the native order of the architecture is little-endian, then each control signal S_i is connected directly to the address bit A_i. If the native order of the architecture is big-endian, then the value of each control signal S_i also depends on the size of the accessed word in the following way: if the accessed word has 2^t bytes then the value of each control signal S_i will be A_i if $i \geq t$ or $\overline{A_i}$ if $i < t$. Note that the new implementation introduces no time penalty since the delay of the interconnection circuits of Figs. 1 and 2 are the same, that is, twice the delay of a multiplexer/demultiplexer. Therefore, the third requirement described in the previous section is met.

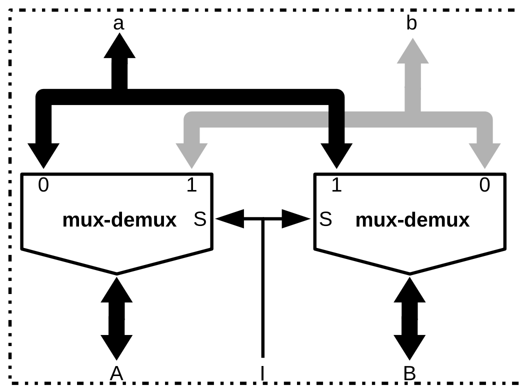


Fig. 3. Swapper circuit.

5. Software

In an architecture with the proposed functionality, each multibyte word W of size $N = 2^t$ stored in memory can be accessed using N different addresses that only differ in the t LSA bits. The lowest of these addresses must be used to read or write W using the native byte order of the architecture, and therefore we will call it *the native-endian address*. The highest of these addresses will be called *the reverse-endian address*. Hence, the address to be used for access in little-endian, that is, the little-endian address, is the native-endian address if the architecture is little-endian, but if the architecture is big-endian, then the little-endian address is the reverse-endian address. Analogously, the big-endian address is the native-endian address if the architecture is big-endian or the reverse-endian address if the architecture is little-endian. In general, any of the N mixed-endian byte orders can be used by selecting one of the N addresses of the word. It could be thought that taking advantage of this functionality is difficult since the programmer must use different addresses to access the same word depending on the byte order to be used. Fortunately, this can be easily solved in Reduced Instruction Set Computer (RISC) architectures by using assembler directives, since these architectures include the following features [18]:

- The only instructions with operands in memory are of load/store type.
- There are only a few data addressing modes, usually only immediate and register (which do not involve memory) and base plus displacement.

Therefore, to use a non-native byte order, it is only necessary to define a pseudoinstruction for every load/store instruction of the ISA. The assembler will replace every instance of the pseudoinstruction with the corresponding load/store instruction of the ISA with the same parameters, but a constant which depends on the desired byte order will be added to the offset. If this constant is simply the size of the accessed word in bytes minus one and it is added to a native-endian address, then the result will be the corresponding reverse-address. For example, suppose we have modified the OpenRISC 1000 32-bit big-endian architecture [19] with the proposed semantics. One of the instructions of its ISA is Load Half Word and Extend with Sign (`l.lhs`). We can use assembler directives to define an analogous little-endian pseudoinstruction `l.lelhs` so that every line in the form

```
l.lelhs rD,Ia(rA)
```

will be replaced with

```
l.lhs rD,Ib(rA)
```

where the offset `Ib` is simply `Ia` plus one (since the size of the access is two bytes). The same holds for store instructions, such as Store Single Word (`l.sw`): an analogous little-endian pseudoinstruction `l.lesw` can be defined so that every line in the form

```
l.lesw Ia(rA),rB
```

will be replaced with

```
l.sw Ib(rA),rB
```

where the offset `Ib` is `Ia` plus three (since the size of the access is four bytes). In this way, the endianness conversion is automatically carried out by the assembler. The programmer only needs to know that, as in the original architecture, every access must be aligned. Note that the memory access and the endianness conversion is carried out using a single assembler instruction.

Carrying out the endianness conversion in C is also very simple, even if the underlying architecture is not RISC, by using the macros of Fig. 4. The parameter of the `le` and `be` macros must be an l-value with a native endian address. The first parameter expands to an l-value with a little-endian address, while the second expands to an l-value with a big-endian address. They can be used to read variables in an alien byte order, but can also be used to write in an alien byte order when they are the left operand of an assignment. For example, suppose we want to assign to a variable `A` the value of another variable `B` multiplied by three. If both variables are stored using the native byte order, then the C code should be the following:

```
A=3*B;
```

If `B` is stored in little-endian and `A` must be stored in big-endian, then the code should be:

```
be(A)=3*le(B);
```

Note that, on the condition that the addresses of the arguments of the macros are known at compilation time, the corresponding binary code would have the same size and execution time and hence the endianness conversion introduces no penalty. To carry out endianness conversions involving arrays, the `lea` and `bea` macros can also be used. The parameter of these macros must be a pointer with a native-endian address to a word. The first macro expands to a pointer with little-endian address to the same word, while the second expands to a pointer with big-endian address to the same word. Again, when they are part of the left operand of an assignment, it is possible to write in an alien byte order. For example, suppose `A` is an one-dimensional array, `B` is a two-dimensional array, and the value `3*B[2][3]` must be assigned to `A[5]`. If both arrays are stored using the native byte order then the following code should be used:

```
A[5]=3*B[2][3];
```

If `B` is stored in little-endian and `A` must be stored in big-endian, then the code could be:

```
bea(A)[5]=3*lea(B)[2][3];
```

or alternatively:

```
be(A[5])=3*le(B[2][3]);
```

Again, the endianness conversion introduces no penalty. This can easily be extended to include any mixed-endian byte order. For example, the `pdp` and `pdpa` macros of Fig. 4 make it possible to use the byte order of the PDP-11 architecture.

More advantages arise when dealing with functions with parameters that are references. To exemplify this, suppose that a library has a function with the following interface:

```
#ifndef _AEBO_H
#define _AEBO_H
#ifndef _ENDIAN_H
#include <endian.h>
#endif
#if (__BYTE_ORDER != __LITTLE_ENDIAN && __BYTE_ORDER !=__BIG_ENDIAN)
#error "This version of the library does not support the host byte order yet."
#endif
#define __reverse_endian_address(x) ((typeof(x)) ((void *)(x) + sizeof(*(x)) - 1))
/*                                    ----------- ------------   -----------------
                                          (3)         (1)              (2)
   1. Convert to void* to have unit pointer arithmetic.
   2. Set less significant address bits to 1: use non-native (reverse) byte order.
   3. Restore the original pointer type.
*/
#define __reverse_endian(x) (*__reverse_endian_address(&(x)))
#if __BYTE_ORDER == __LITTLE_ENDIAN
#define le(x) (x)
#define lea(x) (x)
#define be(x) __reverse_endian(x)
#define bea(x) __reverse_endian_address(x)
#define pdpa(x) ((typeof(x)) ((void *)(x) + (sizeof(*(x))>2?2:0) ))
#define pdp(x) (*pdpa(&(x)))
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
#define le(x) __reverse_endian(x)
#define lea(x) __reverse_endian_address(x)
#define be(x) (x)
#define bea(x) (x)
#define pdpa(x) ((typeof(x)) ((void *)(x) + (sizeof(*(x))>1?1:0) ))
#define pdp(x) (*pdpa(&(x)))
#endif
#endif
```

Fig. 4. Macros defined in aebo.h.

```
void foo(long* bar, int table[],...);
```

This is a ‘normal’ function in the sense that the parameters use the native byte order of the architecture. If we want the function to read and/or write the values of the arguments `native_bar` and `native_table` in the native endianness, it could be called in this way:

```
foo(&native_bar,native_table,...);
```

In a conventional architecture, if we wanted the function to be able to read and/or write one or more arguments using different byte orders, then it would be necessary to introduce at least one new parameter in its interface in order to select the desired endianness. Therefore, the new version of the function could be called in this way:

```
biendian_foo(endian_select,&bar,table,...);
```

This has several drawbacks:

- Obviously, the new function must be written and compiled.
- The new function cannot replace the old function because they have different interfaces and hence both versions must be maintained.
- It is necessary to include code in the new function to evaluate the new parameter and to act thereon. This introduces a penalty even if the ISA includes instructions to read and write memory in an alien byte order.

In contrast, if the architecture were extended to support the proposed functionality, then there would be no need to rewrite nor even recompile the function to use selectable endianness as long as it always accessed each word as a whole. For example, the arguments `my_bar` and `my_table` would be accessed by the function in big-endian and little-endian respectively, simply by calling it in this way:

```
foo(bea(&my_bar),lea(my_table),...);
```

Once more, there would be no penalty in the execution time of the function for using an alien byte order.

```
fread(table,size,1,input_file);
offset=dc_offset(table,length);
add_dc(-offset,table,length);
fwrite(table, size, 1, output_file);
```

Fig. 5. Simplified software filter that process data in native byte order.

6. Results

In order to measure the impact of the introduced functionality, the proposed modification has been applied to the 32-bit OpenRISC 1000 architecture. This architecture was chosen for the following reasons:

- Although an OpenRISC 1000 implementation supporting unaligned accesses can be carried out, the architecture establishes that, by default, unaligned accesses are not allowed and should trigger an exception [19]. For this reason, the proposed modification can be applied.
- There are open implementations that can easily be modified and synthesized for the reconfigurable hardware available to the authors [20].
- There are unrestricted operating systems and tool-chains available for the architecture that will allow performance measurements [21,22].

The original and modified architectures have been implemented, and programs with identical functionality have been executed in both implementations. These programs are versions of a basic filter written in C for the original architecture, which have been improved to process data in big-endian and little-endian. The original filter simply removes the DC offset of a set of 32-bit samples written in the native byte order. A simplification of its code is shown in Fig. 5.

Table 3
Synthesis results.

Architecture	Clock frequency	LUTs	Registers
Original	Maximum (100 MHz)	10 619	7779
Modified	Maximum (100 MHz)	10 552	7778

Table 4
Execution time comparison.

	Original time (s) Mean (Std. Dev.)	Modified time (s) Mean (Std. Dev.)	Speed up
User	0.505 (0.023)	0.200 (0.012)	60.46%
System	0.344 (0.040)	0.345 (0.030)	−0.26%
Total	0.865 (0.043)	0.559 (0.032)	35.33%

```
fread(table,size,1,input_file);
# if __BYTE_ORDER == __BIG_ENDIAN
if(selected_endian==little)
    for(index=0;index<length;index++)
        table[index]=le32toh(table[index]);
# else//host byte order is __LITTLE_ENDIAN
if(selected_endian==big)
    for(index=0;index<length;index++)
        table[index]=be32toh(table[index]);
# endif
offset=dc_offset(table,length);
add_dc(-offset,table,length);
# if __BYTE_ORDER == __BIG_ENDIAN
if(selected_endian==little)
    for(index=0;index<length;index++)
        table[index]=htole32(table[index]);
# else//host byte order is __LITTLE_ENDIAN
if(selected_endian==big)
    for(index=0;index<length;index++)
        table[index]=htobe32(table[index]);
# endif
fwrite(table, size, 1, output_file);
```

Fig. 6. Simplified improved filter for the original architecture.

6.1. Synthesis

An open implementation of the original architecture called *mor1kx* [20] and a modification to support the proposed functionality have been synthesized for the Digilent Nexys A7 Board [23] featuring a Xilinx Artix XC7A100T-CSG324 FPGA chip [24]. The only part of the data unit modified to support the proposed functionality is the combinational logic of the load/store unit, and the only difference in the behaviour of the control unit is that the unaligned access exception is disabled and hence the number of clock cycles required to execute each instruction are the same in both implementations. The full projects, including the bit streams, can be downloaded from [25] and [26]. The default options of the Vivado tool are used for synthesis. Results are as shown in Table 3. Since both implementations reached the maximum operation frequency supported by the board, i.e. 100 MHz, the execution time of a program written for the original architecture is the same in both. The employed resources were slightly reduced in the modified version.

6.2. Software test bench

The performance measurement is carried out under the Linux kernel 4.4.0 [21]. Since the proposed semantics ensures backward compatibility at binary-code level, the implementation of the modified architecture successfully runs the operating system of the original architecture without modifications. Furthermore, the execution time of the software written for the original architecture turns out to be the same. The

```
fread(table,size,1,input_file);
offset=dc_offset(selected_endian==big?
    bea(table) : lea(table),length);
add_dc(-offset,selected_endian==big?
    bea(table) : lea(table),length);
fwrite(table, size, 1, output_file);
```

Fig. 7. Simplified improved filter for the modified architecture.

improved versions of the software filter used in the test have to deal with the fact that the data may be written in little-endian byte order. It must be noted that the little-endian byte order is optionally supported by the OpenRISC 1000 architecture. If an implementation supported said order, then the little-endian byte order would be activated by setting the Little Endian Enable (LEE) bit of the SR register. However, it would be helpless in this case since it would not be possible to selectively change the endianness used by each process, and mor1kx does not support this functionality. Hence, the improved filter for the original architecture will carry out the endianness conversion by software using the `endian` library of `glibc`. A modification of the `dc_offset` and `add_dc` library functions used in the program for the original architecture to deal with arguments in an alien byte order would be inefficient since the input data should be converted twice. Instead, the input data will be converted to the native byte order before processing, and output data will be converted to the target byte order as shown in Fig. 6. The improved filter for the modified architecture is much simpler, since only the invocation of the functions have been modified, as shown in Fig. 7. The full program can be downloaded from [27]. The programs were compiled using the 5.3.0 version of the `or1k-linux-musl-gcc` tool-chain [22]. The execution time of both programs with random files of 250 K bytes stored in RAM was measured 100 times. The measurements can be repeated by executing the `test_bench.sh` shell script available in [27]. The results are shown in Table 4. The total execution time was reduced by ca. 35% with the proposed functionality, while the user execution time was reduced by over 60%. This result was expected since the OpenRISC 1000 ISA does not include instructions to reorder the bytes of a register and therefore the program for the original architecture had to carry out each endianness conversion using several logic and shift instructions. In particular, for 4-byte words, the `endian` library uses the internal C function `__bswap32`, which uses many instructions as shown in Fig. 8. The overhead of these instructions was removed in the modified version and hence the execution time was remarkably reduced. The size of the executable was also reduced from 9428 bytes to 9112 bytes, that is, a reduction of 3.35%.

7. Discussion

The processor modification was very straightforward, required negligible engineering cost and does not impact the performance when executing legacy code. Because of that it may be desirable independently of the expected applications of the processor.

8. Conclusions

Semantics for the least significant address bits of multibyte accesses in architectures that only support aligned accesses have been proposed. They have the following advantages:

- Existing architectures can easily be modified to include the proposed semantics so that a single instruction can carry out a memory access and a byte order conversion.
- The modification does not require new instructions nor operation codes since the least significant address bits are employed to code the byte order to be used.

```
__bswap32:
.cfi_startproc
l.sw     -4(r1),r2
.cfi_offset 2, -4
l.addi   r2,r1,0
.cfi_def_cfa_register 2
l.addi r1,r1,-8
l.sw     -8(r2),r3
l.lwz    r3,-8(r2)
l.srli   r4,r3,24
l.lwz    r3,-8(r2)
l.srli   r3,r3,8
l.andi   r3,r3,65280
l.or     r4,r4,r3
l.lwz    r3,-8(r2)
l.slli   r3,r3,8
l.movhi  r5,hi(16711680)
l.and    r3,r3,r5
l.or     r4,r4,r3
l.lwz    r3,-8(r2)
l.slli   r3,r3,24
l.or     r3,r4,r3
l.ori    r11,r3,0
l.ori r1,r2,0
l.lwz    r2,-4(r1)
l.jr     r9
l.nop
.cfi_endproc
```

Fig. 8. Assembler code corresponding to the _bswap32 function.

- Any architecture modified to include the proposed semantics will be backwards compatible at binary-code level and hence their implementations will be able to run any software written for the original architecture.
- The modification does not introduce any penalty.
- Assemblers and C compilers of the original architecture can easily be employed to take advantage of the new functionality by defining a few simple preprocessor macros.
- With the new functionality, subroutines written for the original architecture will be able to process referenced data in an alien byte order without any penalty being incurred. There is no need to rewrite nor even recompile these subroutines to take advantage of this functionality.
- An implementation of a traditional architecture was modified to support the proposed functionality. The modification did not introduce any penalty in hardware resources nor in execution time.
- A test bench processing data in a non native byte order intensively was executed in the original implementation and in the modified implementation. There was a reduction of over 35% of the total execution time and over 60% of the user execution time in the modified implementation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been partially supported by the Ministerio de Economía, Industria y Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and by the European Regional Development Fund (ERDF).

References

[1] K. Cooper, L. Torczon, Engineering a Compiler, Elsevier Science, 2011, URL https://books.google.es/books?id=_tgh4bgQ6PAC.

[2] D. Cohen, On Holy Wars and a Plea for Peace, Computer 14 (10) (1981) 48–54, http://dx.doi.org/10.1109/C-M.1981.220208.

[3] IEEE standard for information technology–portable operating system interface (POSIX(R)) base specifications, Issue 7, in: IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), 2018, pp. 1–3951, http://dx.doi.org/10.1109/IEEESTD.2018.8277153.

[4] J.M. O'Connor, M. Tremblay, PicoJava-I: the Java virtual machine in hardware, IEEE Micro 17 (2) (1997) 45–53, http://dx.doi.org/10.1109/40.592314.

[5] M. Garcia, E. Francesquini, R. Azevedo, S. Rigo, HybridVerifier: A cross-platform verification framework for instruction set simulators, IEEE Embedded Syst. Lett. 9 (2) (2017) 25–28, http://dx.doi.org/10.1109/LES.2016.2626980.

[6] I. Horton, Beginning C++, Apress, Berkeley, CA, 2014, pp. 1–22, http://dx.doi.org/10.1007/978-1-4842-0007-0_1.

[7] M. Arora, The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits, Springer New York, New York, NY, 2012, pp. 155–168, http://dx.doi.org/10.1007/978-1-4614-0397-5_7.

[8] H.E. Yantir, A. Yurdakul, An efficient Heterogeneous register file implementation for FPGAs, in: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 293–298, http://dx.doi.org/10.1109/IPDPSW.2014.40.

[9] J.-J. Li, S.-C. Wang, P.-C. Hsu, P.-Y. Chen, J.K. Lee, A multi-core software API for Embedded MPSoC environments, in: Proceedings of the Second Russia-Taiwan Conference on Methods and Tools of Parallel Programming Multicomputers, in: MTPP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 40–50, URL http://dl.acm.org/citation.cfm?id=1927517.1927524.

[10] J. Henkel, S. Parameswaran (Eds.), Designing Embedded Processors: A Low Power Perspective, Springer Netherlands, 2007.

[11] M.A.A. Farhan, D.E. Keyes, Optimizations of unstructured aerodynamics computations for many-core architectures, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2317–2332, http://dx.doi.org/10.1109/TPDS.2018.2826533.

[12] R. Auler, E. Borin, The case for flexible ISAs: Unleashing hardware and software, in: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2017, pp. 65–72, http://dx.doi.org/10.1109/SBAC-PAD.2017.16.

[13] G. Kondoh, H. Komatsu, Dynamic Binary translation specialized for embedded systems, SIGPLAN Not. 45 (7) (2010) 157–166, http://dx.doi.org/10.1145/1837854.1736019, URL http://doi.acm.org/10.1145/1837854.1736019.

[14] M. Souza, D. Nicácio, G. Araújo, ISAMAP: Instruction Mapping driven by dynamic binary translation, in: A.L. Varbanescu, A. Molnos, R. van Nieuwpoort (Eds.), Computer Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 117–138.

[15] A. Sloss, D. Symes, C. Wright, in: M. Kaufmann (Ed.), ARM System Developer's Guide, 2004, p. 689.

[16] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scale, AltiVec extension to PowerPC accelerates media processing, IEEE Micro 20 (2) (2000) 85–95, http://dx.doi.org/10.1109/40.848475.

[17] J. Rentzsch, Data alignment: Straighten up and fly right, 2005, URL https://www.ibm.com/developerworks/library/pa-dalign/index.html.

[18] J.L. Hennessy, a.A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2011.

[19] OpenRISC 1000 Architecture Manual, 2012, URL http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf.

[20] mor1kx IP core specification, URL https://github.com/openrisc/mor1kx/blob/master/doc/mor1kx.asciidoc.

[21] Linux kernel 4.4.0 for OpenRISC, URL https://github.com/openrisc/linux/tree/for-next/kernel.

[22] or1k-linux-musl-gcc tool-chain, URL https://github.com/openrisc/musl-cross.

[23] Nexys A7 FPGA Trainer Board, URL https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/.

[24] Artix-7 devices, URL https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

[25] mor1kx synthesized for the Digilent Nexys 4 DDR board, URL https://gitlab.com/davidguerrero/mor1kx-synthesized-for-the-digilent-nexys-4-ddr.

[26] A mor1kx implementation modified to implement a variation of the 32 bit OpenRISC 1000 architecture with multiendian capabilities and synthesized for the Digilent Nexys 4 DDR board, URL https://gitlab.com/davidguerrero/mor1kx-multiendian.

[27] Software test bench to compare the 32 bit OpenRISC 1000 architecture with a variation with multiendian capabilities running Linux, URL https://gitlab.com/davidguerrero/openrisc-1000-multiendian-test-bench.

David Guerrero Martos received his Bsc degree and the Ph.D. degree in Computer Engineering from the University of Seville, Spain, in 2000 and 2012, respectively. Since 2002, he has been working as a lecturer in the Department of Electronics Technology of said university. His research interests include digital circuit synchronization, hardware implementation of numerical methods, and computer architecture. He has published several papers in journals and conferences.

German Cano received his Bsc degree in computing engineering from the University of Seville, Spain, in 2015. He has been a Ph.D. student in the Electronics Technology Department at this university since 2017. His research interests include Bootloaders, IoT, and SoC.

Dr. Jorge Juan received his Bsc degree (1994) and Ph.D. degree in Physics (2000) from the University of Seville, Spain. He is currently a lecturer in the Electronics Technology Department at this university where he is leading the Digital Research and Development Group. Dr. Juan has carried out research in the areas of metastability, delay modelling, timing and power simulation, and digital embedded systems.

Alejandro Millan was born in Seville in 1975. He received his M.Sc. in Computer Engineering in 1999 and his Ph.D. in 2008, from the University of Seville. He works as a Professor at its Department of Electronics Technology. Since 1999, he has taught at the School of Computer Engineering and the Polytechnic School of Engineering. He is a member of its ID2 Research Group where he has participated in 20 research projects, and he has published in excess of 50 conference papers and a total of 17 journal papers.

Dr. Manuel J. Bellido received his B.Sc. degree (1987) and Ph.D. degree (1994) in Physics from the University of Seville, Spain. He has been with the Electronics Technology Department at this university since 1990 where he holds a post as a Professor.

Julian Viejo received his M.Sc. and Ph.D. degrees in Computing Engineering from the University of Seville (Spain) in 2004, and 2011, respectively. He works as an assistant professor in the Department of Electronics Technology of that University and has contributed several research papers to international journals and conferences in the area of Digital Signal Processing and System-on-Chip design.

Paulino Ruiz-de-Clavijo received his Bsc degree (1999) and Ph.D. degree (2007) in computer science from the University of Seville (Spain). He has been with the Electronics Technology Department at this university since 1999 where he has held a post as an assistant professor since 1999. He was also with the Institute of Microelectronics of Seville, part of the National Centre of Microelectronics in Spain, from 1998 to 2004. His research work includes system-on-chip designs, Digital Signal Processing, and embedded microprocessors architecture: areas to which he has contributed in international conferences and workshops.

Enrique Ostua received his B.Sc. degree in computing engineering (2003) and his M.Sc. in computing & networking engineering (2015) from the University of Seville. He has been an associate professor in the Department of Electronics Technology, University of Seville, since 2003. His research interests include system-on-chip designs and embedded microprocessor architecture areas, to which he has contributed in international journals, conferences, and workshops.