Endianness
or
Where is Byte 0?

White Paper

Bertrand Blanc – 3B Consultancy – bertrand.blanc@3B-Consultancy.com
Bob Maaraoui – Texas Instruments – bmaa@ti.com

Endianness Keywords: *endianness, little-endian, big-endian, endian-neutral, byte 0 location, Least Significant Byte, Most Significant Byte, communication, endian conversions.*

Enhanced Endianness Keywords: *enhanced endianness, endianness, little-endian, big-endian, middle-endian, endian-neutral, byte 0 location, Least Significant Element/Unit, Most Significant Element/Unit, data width, data atomic element/unit width, least addressable element/unit, byte-swap, bit-swap, communication, endian conversions.*

Preliminaries 2
Assumptions 3
Sub-Width 5
Endianness 6
Byte Swap 7
Bit Swap 8
Integration of Heterogeneous Components 9
Exchanged Data Study 13
Little-Endian towards Big-Endian.....13
Big-Endian towards Little-Endian.....13
Little-Endian towards Little-Endian.....13
Big-Endian towards Big-Endian.....13
Summary 15

Preliminaries

This document aims to present how Endianness is willing to be taken into consideration to let Endian specific system inter-operate sharing data without misinterpret value.

Intel introduces their white paper with the following sentence:
"Endianness describes how multi-byte data is represented by a computer system and is dictated by the CPU architecture of the system. Unfortunately not all computer systems are designed with the same Endian-architecture. The difference in Endian-architecture is an issue when software or data is shared between computer systems. An analysis of the computer system and its interfaces will determine the requirements of the Endian implementation of the software. "

This paper targets CPU-based Endianness raising software issues. We will try to reinterpret these information at register/system-level. Indeed, a system is composed of components seen at system-level as black-boxes defined elsewhere (legacy component, reused component, purchased component, …). All these components were defined with a basic Endian mode which may differ once integrated together. We can easily understand that a data send in little-endian format will be interpreted in a wrong way once caught by a module big-Endian-based.

Intel gives the following definition:
"Endianness is the format to how multi-byte data is stored in computer memory. It describes the location of the most significant byte (MSB) and least significant byte (LSB) of an address in memory. Endianness is dictated by the CPU architecture implementation of the system. The operating system does not dictate the endian model implemented, but rather the endian model of the CPU architecture dictates how the operating system is implemented.

Representing these two storage formats are two types of Endianness-architecture, Big-Endian and Little-Endian. There are benefits to both of these endian architectures. Big-Endian stores the MSB at the lowest memory address. Little-Endian stores the LSB at the lowest memory address. The lowest memory address of multi-byte data is considered the starting address of the data."

This definition seems fine, but a few assumptions are to be established to fit register/system-level Endianness. We will present in a first part what these assumptions are, followed with major Endian formats met: big-endian and little-endian.

We will hence address consequences of Endianness, enriched with bit-order and byte-swap. New features will be introduced beyond basic Endianness, leading to a model of Enhanced Endianness.

Note to the Reader: This study is part of a set of studies, some background is hence missing to accurately understand all aspects of this document, especially unified component foundations. Nevertheless, beyond some obfuscated terms coming from the background, this document targets Enhanced Endianness and should be self-containing to address this topic by itself.

Assumptions

We presented in a previous document that a system, whatever it may be (infinite hierarchal systems or raw-register leaves), is defined as

$$(\mathbf{S_n, R_m, W_{m'}, I_{m'}, RST_r, V, B, O, M, T, Z_n})$$

with	S_n	a set of switch control-signal
	R_m	an ordered set of m Read-access lines triggered by S_n
	$W_{m'}$	an ordered set of m' Write-access lines triggered by S_n
	$I_{m'}$	an ordered set of m' input data bus lines triggered by S_n associated to each $W_{m'}$ lines
	RST_r	a set of reset control-signals (power-on-resets, software resets, …) preempting regular behaviors
	V	an inner retained sized data
	B	a set of behaviors triggered by S_n, R_m, $W_{m'}$, RST_r control-signals, $I_{m'}$ and V data values as well
	O	output data bus
	M	a set of inner sub-components triggered by S_n
	T	a memory allocation table triggered by S_n, sorting M allocated components
	Z_n	an ordered set of component size triggered by S_n

Note 1: a raw-register (final system leaf) is seen as above, with assumptions: $M = \{\}$, and $T = \{\}$.
Note 2: a system is seen as above, with assumptions: $B = \{\}$ (inferred: $M \neq \{\}$, and $T \neq \{\}$)

S_n may compute data coming from outside the current pertained component. $I_{m'}$, V, O are storing or carrying data. Z_n fix component size. These data have not been considered up to now since we especially targeted control-oriented signals giving up any data meaning. However, interpretation of these data is mandatory to correctly compute trigger signals, or broadcasting data with a known format in order to enable other modules to correctly interpret these data.

Example 1: according to data representation, 0xAB and 0xBA may represent the same data.

Example 2: according to data representation, algorithms may be more efficient or not. An addition should compute least significant bit first in order to broadcast carry if needed, whilst a < operation should compute most significant bit first in order to minimize computations.

Example 3: despite a component is defined and designed to fit a given sub-width, modules outside may themselves be designed with another sub-width creating incoherencies within Endianness understandings between these modules.

System definition above is therefore enriched with following data representation features:

1. sub-width $\mathbf{ZS_n}$ defines in a given mode, what is basic physical register width this system is made of, targeting physical memory mapping into a given memory
2. Endianness $\mathbf{E_n}$ defines in a given mode, how data are handled within a given sub-width atomic data item
3. bit-order $\mathbf{bS_n}$ defines in a given mode, which bit is significant first, within a sub-width atomic data item
4. byte-order $\mathbf{BS_n}$ defines in a given mode, which byte is significant first, within a sub-width atomic data item

Note 1: In a given mode, we notice that data are represented with unique sub-width, unique Endianness representation, unique bit-order and unique byte-order.

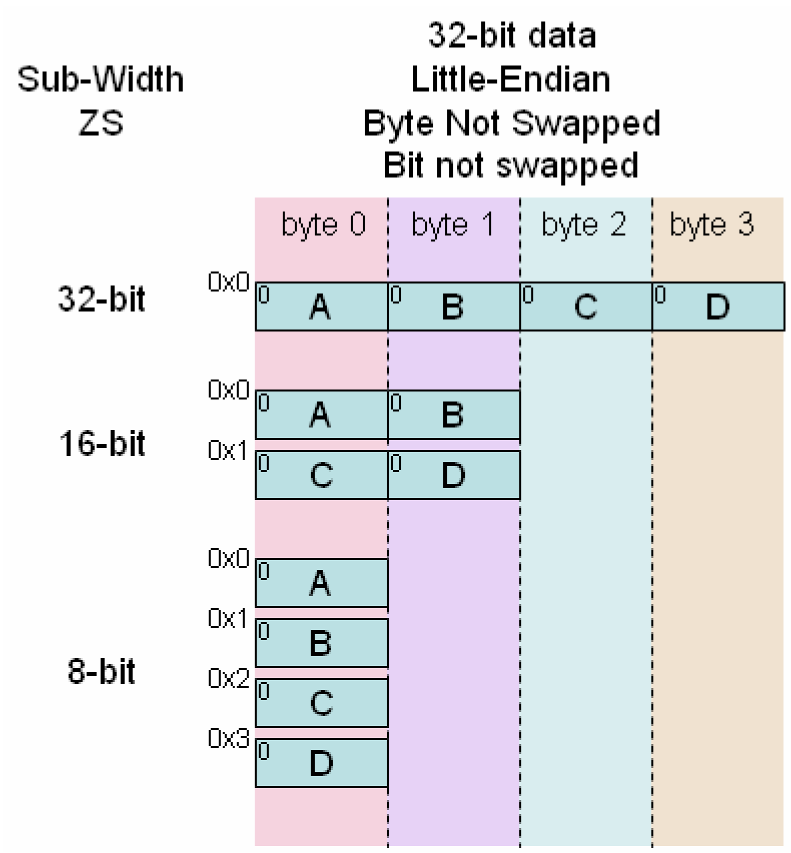
A container will be called packet likewise, composed of ZS bytes. We should hence say which byte is the first one within containers. In the same way we should say which bit is the first one within bytes.

Endianness targets containers order within a Z-width data, stating which container is the first one located at the lowest address.

Byte-swap targets bytes order within a ZS-width container, stating which byte is the first one located at the lowest address.

Bit-swap targets bits order within a byte, stating which bit is the first one.

Ordering convention will be as depicted below, tagged as a 32-bit data, composed of ZS-width containers, in little-endian mode, without any byte-swap, without any bit-swap: 32/ZS/*little*/*no-byte-swap*/*no-bit-swap*. This representation is the underlying standard one.



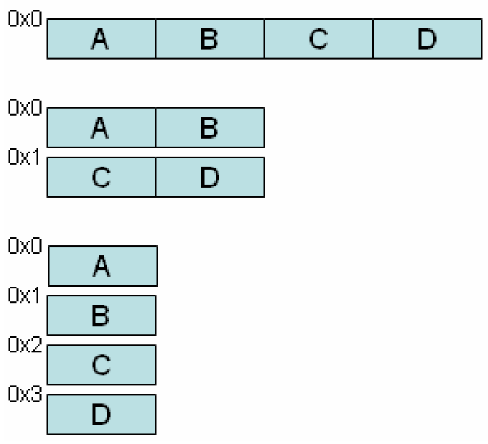
Sub-Width

Synonyms: Atomic Element size/width, Least Addressable Unit (LAU)

We can ask ourselves whether these 4 features are necessary, whether only handling Endianness is not enough.

First of all, despite byte is the smaller addressable memory item, some memories may not be byte-based in their addressable memory address computation system. The may be 16-bit, 32-bit, … memory addressable. Sub-width information is hence mandatory to get how memory addresses are incremented.

According to physical memory address system, a 32-bit register may be seen as:



In first layout 32-bit sub-width-based, the 32-bit register is represented in an atomic memory space at address 0x0, next register may be located at address 0x1. In second layout 16-bit sub-width based, the 32-bit register is represented in 2 separate contiguous memory space at addresses 0x0 and 0x1, next register will be at address 0x2. In 3rd layout 8-bit sub-width-based, the 32-bit register is represented in separate contiguous 4 atomic address spaces at addresses 0x0, 0x1, 0x2 and 0x3, next register may be located at address 0x4. We can notice that 3rd data representation is the smaller one, since based on byte-based representation.

Example: we assume the same 32-bit register is represented with 32-bit sub-width in mode M32, with 16-bit sub-width in mode M16, and with 8-bit sub-width in standard mode:

- S_n = {*std*, M32, M16}
- Z_n = {*std* → 32-bit, M32 → 32-bit, M16 → 4-byte} = {32-bit}
- ZS_n = {*std* → 1-byte, M32 → 4-byte, M16 → 16-bit}

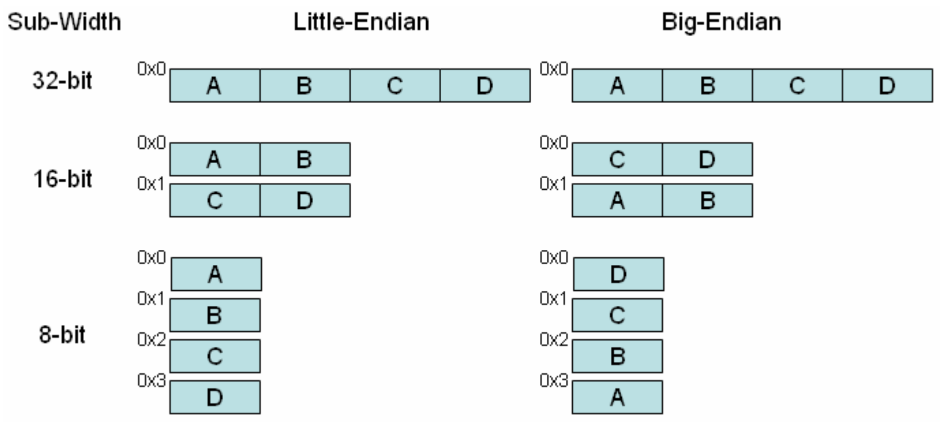
Endianness

Thru sub-width feature we can sort atomic sub-widths items within a given register. Commonly admitted definition of Endianness is

1. in Big-Endian, Most-Significant-Byte (MSB) is stored at lowest address
2. in Little-Endian, Least-Significant-Byte (LSB) is stored at lowest address

This couple of definition byte-based describes a given static 8-bit sub-width. However, we noticed that sub-width may be 8-bit-based, but may be 16-bit-based, … This definition hence becomes inaccurate. Nevertheless, for historical reasons, we will still employ MSB and LSB words, keeping in mind "B" standing for "byte" must be seen as "sub-width".

Example 1: The picture below shows how data are ordered for a 32-bit register, in both little-endian and big-endian representation layout, for 32-bit, 16-bit and 8-bit sub-widths. Byte order within containers, and bit order within bytes is not significant.



Note: These translations are called "butterfly".

Example 2: We assume the same 32-bit register is represented with 32-bit sub-width in mode M32, with 16-bit sub-width in mode M16, and with 8-bit sub-width in standard mode. The component is defined in standard mode with little-endian layout, and in big-endian in M32 and M16.

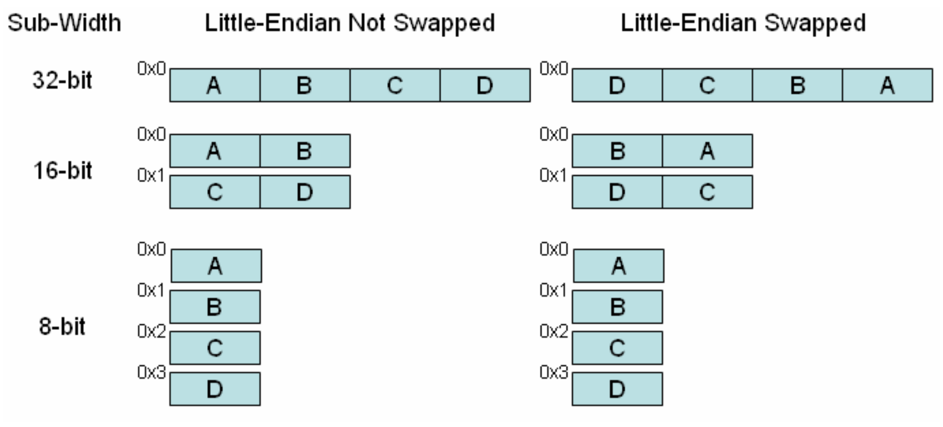
This system is defined as:

- S_n = {*std*, M32, M16}
- Z_n = {*std* → 32-bit, M32 → 32-bit, M16 → 4-byte} = {32-bit}
- ZS_n = {*std* → 1-byte, M32 → 4-byte, M16 → 16-bit}
- E_n = {*std* → *little*, M32 → *big*, M16 → *big*} = {*std* → *little*, (M32, M16 → *big*)}

Byte Swap

This feature gives an order on bytes within a same atomic sub-with item to denote which byte is the most/least significant. This feature fits more MSL/LSB definitions given in Endianness section. Byte-swap is sometimes called middle-endian.

Example 1: The picture below shows how data are ordered for a 32-bit register, in little-endian format, handling both swapped and not-swapped byte order, for 32-bit, 16-bit and 8-bit sub-widths.



Example 2: We assume the same 32-bit register is represented with 32-bit sub-width in mode M32, with 16-bit sub-width in mode M16, and with 8-bit sub-width in standard mode. The component is defined in standard mode with little-endian layout, and in big-endian in M32 and M16. Byte-swap is only consistent in M16 mode.

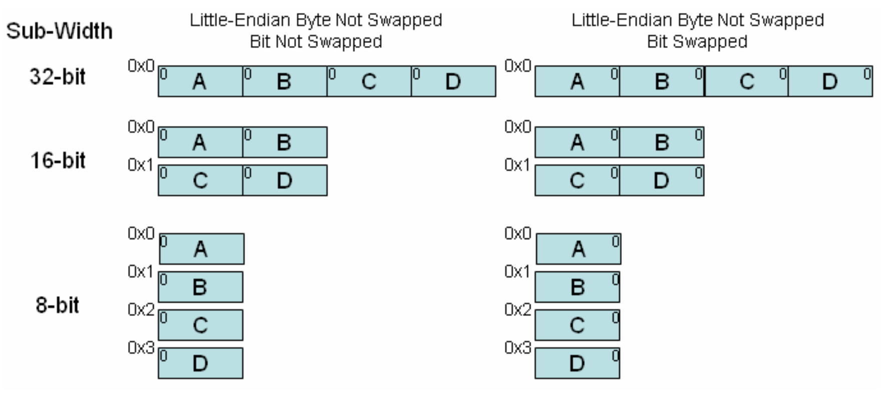
This system is defined as:

- S_n = {*std*, M32, M16}
- Z_n = {*std* → 32-bit, M32 → 32-bit, M16 → 4-byte} = {32-bit}
- ZS_n = {*std* → 1-byte, M32 → 4-byte, M16 → 16-bit}
- E_n = {*std* → *little*, M32 → *big*, M16 → *big*} = {*std* → *little*, (M32, M16 → *big*)}
- BS_n = {*std* → *not-swapped*, M32 → *not-swapped*, M16 → *swapped*}

Bit Swap

This feature denotes which bit is the most/least significant within a byte. Indeed, address 0x0 can start with bit0, or bit7 of the same byte.

Example 1: The picture below shows how data are ordered for a 32-bit register, in little-endian format, bytes are not swapped, handling both swapped and not-swapped bit order, for 32-bit, 16-bit and 8-bit sub-widths.



Example 2: We assume the same 32-bit register is represented with 32-bit sub-width in mode M32, with 16-bit sub-width in mode M16, and with 8-bit sub-width in standard mode. The component is defined in standard mode with little-endian layout, and in big-endian in M32 and M16. Byte-swap is only consistent in M16 mode. Bit-swap only targets M32 mode.

This system is defined as:

- S_n = {*std*, M32, M16}
- Z_n = {*std* → 32-bit, M32 → 32-bit, M16 → 4-byte} = {32-bit}
- ZS_n = {*std* → 1-byte, M32 → 4-byte, M16 → 16-bit}
- E_n = {*std* → *little*, M32 → *big*, M16 → *big*} = {*std* → *little*, (M32, M16 → *big*)}
- BS_n = {*std* → *not-swapped*, M32 → *not-swapped*, M16 → *swapped*}
- bS_n = {*std* → *not-swapped*, M32 → *swapped*, M16 → not-*swapped*}

Note: To ease and shorten component size representation (size, sub-width, Endianness, byte-swap, bit-swap) contained in separate sets of features (Z_n, ZS_n, E_n, BS_n, bS_n) triggered by modes in S_n, we will note:

$$S_n \rightarrow Z_n/ZS_n/E_n/BS_n/bS_n$$

Besides, *little-endian* and *big-endian* are shortened into **l** and **b**; *swapped* and *not-swapped* are shortened into **S** and **nS**.

Example:

- M32 → 32-bit/4-byte/b/nS/S
- M16 → 4-byte/16-bit/b/S/nS
- *std* → 32-bit/1-byte/l/nS/nS

Note: most of registers are captured with standard view: **32-bit/8-bit/l/nS/nS**.

Integration of Heterogeneous Components

We saw each component is designed setting up a common context ($S_n \rightarrow Z_n/ZS_n/E_n/BS_n/bS_n$) in the scope of each of them. We will present in this section how to handle data coherency between each of them once integrated.

Be a system C = (S_n, R_m, $W_{m'}$, $I_{m'}$, RST_r, V, B, O, M, T, $Z_n/ZS_n/E_n/BS_n/bS_n$), M = $\{M_i\}_k$.

As it is defined (including default settings) for each sub-components from M, $Z_n/ZS_n/E_n/BS_n/bS_n$ features are set up for C as well.

These data-layout features are visible at C boundary hiding encapsulated ones mapped in C.T from C.M. The context set up at C level must hence be reestablished between C.M allocated components to ensure data coherency.

Example 1: M1 is 32-bit/32-bit/l/nS/nS, M2 is 32-bit/8-bit/b/nS/nS. M1 and M2 communicate sharing data: M1 sends data 0xABCD, M2 hence receives data 0xABCD and translate it into 0xDCBA before computing it.

Fixing data format features at C-level will inform outside this module (seen as a black-box) how data are managed within it (white-box), and hence how to keep data meaning between heterogeneous components.

Example 2: A 4-KB component D is designed as 32-bit/16-bit/l/nS/nS by a 3rd vendor. This system is integrated within a 32-bit/8-bit/X/Y/Z environment wrapped by component C. We propose to wrap this legacy component to propose four separate memory spaces fitting each possibility based on an 8-bit sub-width (feature coming from C):

1. little-endian, bits not swapped
2. little-endian, bits swapped
3. big-endian, bits not swapped
4. big-endian, bits swapped

These wrappers will hence be declared as:

1. 32-bit/8-bit/l/nS/nS = 32-bit/16-bit/l/nS/nS = 32-bit/32-bit/l/nS/nS
2. 32-bit/8-bit/l/nS/S = 32-bit/16-bit/l/nS/S = 32-bit/32-bit/l/nS/S
3. 32-bit/8-bit/b/nS/nS = 32-bit/16-bit/b/S/nS = 32-bit/32-bit/l/S/nS
4. 32-bit/8-bit/b/nS/S = 32-bit/16-bit/b/S/S = 32-bit/32-bit/l/S/S

Note 1: byte-order is close to Endianness handling, especially when sub-widths are different supposed to keep coherency between exchanged data.

Note 2: bit-order is fully decoupled in the model having no interaction with Endianness management.

C is hence designed as shown below. C will be seen from outside as a full 32-bit/8-bit/l/nS/nS component, but will offer the opportunity to have data available in each various layout according to which memory space is targeted.

1. C = (S_n, R_m, $W_{m'}$, $I_{m'}$, RST_r, V, B, O, M, T, {32-bit/8-bit/X/X/X})
2. M = {M_1, M_2, M_3, M_4}
3. M_1 = ($S1_n$, $R1_m$, $W1_{m'}$, $I1_{m'}$, $RST1_r$, V1, B1, O1, M1, T1, {32-bit/8-bit/l/nS/nS}), M1 = {D}
4. M_2 = ($S2_n$, $R2_m$, $W2_{m'}$, $I2_{m'}$, $RST2_r$, V2, B2, O2, M2, T2, {32-bit/8-bit/l/nS/S}), M2 = {D}
5. M_3 = ($S3_n$, $R3_m$, $W3_{m'}$, $I3_{m'}$, $RST3_r$, V3, B3, O3, M3, T3, {32-bit/8-bit/b/nS/nS}), M3 = {D}
6. M_4 = ($S4_n$, $R4_m$, $W4_{m'}$, $I4_{m'}$, $RST4_r$, V4, B4, O4, M4, T4, {32-bit/8-bit/b/nS/S}), M4 = {D}
7. T = {0x0 $\leftarrow M_1$, 4-KB $\leftarrow M_2$, 8-KB $\leftarrow M_3$, 12-KB $\leftarrow M_4$}

Note 1: We notice C features for Endianness, byte-swap and bit-swap are left blank thru 'X'. This means component C is designed Endianness-free relegating Endianness handling to sub-components level. Module C can be seen at its boundary as 32-bit/8-bit/l/nS/nS, 32-bit/8-bit/l/nS/S, 32-bit/8-bit/b/nS/nS and 32-bit/8-bit/b/nS/S.

Note 2: in previous documents 4-KB in 6th item would have been written "f(4-KB)". Function f can be removed now since it was supposed to wrap ($S_n \rightarrow Z_n/ZS_n/E_n/BS_n/bS_n$) topic, which has just been sorted out.

From note 1 we may propose a methodological rule using features provided by this model: Endianness features of an Endianness-free module are constrained by switches computing Command Bus data *Cmd* (used to select the address matching T allocated addresses).

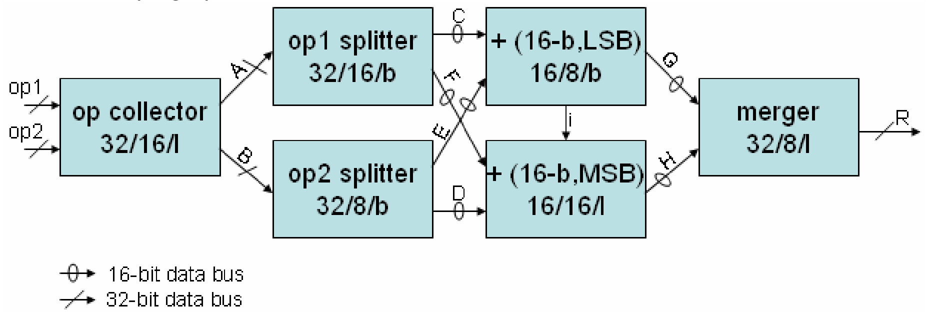
1. $C = (S_n, R_m, W_{m'}, I_{m'}, RST_r, V, B, O, M, T, Z_n/ZS_n/E_n/BS_n/bS_n)$
2. $S_n = S_{n'} \cup \{S1 \leftarrow Cmd \geq 0x0$ and $Cmd < 4\text{-KB}$,
   $S2 \leftarrow Cmd \geq 4\text{-KB}$ and $Cmd < 8\text{-KB}$,
   $S3 \leftarrow Cmd \geq 8\text{-KB}$ and $Cmd < 12\text{-KB}$,
   $S2 \leftarrow Cmd \geq 12\text{-KB}\}$
3. $M = \{M_1, M_2, M_3, M_4\}$
4. $Z_n/ZS_n/E_n/BS_n/bS_n = \{S1 \rightarrow$ 32-bit/8-bit/l/nS/nS,
   $S2 \rightarrow$ 32-bit/8-bit/l/nS/S,
   $S3 \rightarrow$ 32-bit/8-bit/b/nS/nS,
   $S4 \rightarrow$ 32-bit/8-bit/b/nS/S$\}$
5. $M_1 = (S1_n, R1_m, W1_{m'}, I1_{m'}, RST1_r, V1, B1, O1, M1, T1, \{$32-bit/8-bit/l/nS/nS$\})$, $M1 = \{D\}$
6. $M_2 = (S2_n, R2_m, W2_{m'}, I2_{m'}, RST2_r, V2, B2, O2, M2, T2, \{$32-bit/8-bit/l/nS/S$\})$, $M2 = \{D\}$
7. $M_3 = (S3_n, R3_m, W3_{m'}, I3_{m'}, RST3_r, V3, B3, O3, M3, T3, \{$32-bit/8-bit/b/nS/nS$\})$, $M3 = \{D\}$
8. $M_4 = (S4_n, R4_m, W4_{m'}, I4_{m'}, RST4_r, V4, B4, O4, M4, T4, \{$32-bit/8-bit/b/nS/S$\})$, $M4 = \{D\}$
9. $T = \{0x0 \leftarrow M_1, 4\text{-KB} \leftarrow M_2, 8\text{-KB} \leftarrow M_3, 12\text{-KB} \leftarrow M_4\}$

Note 1: we can infer a coherency property targeting Endianness features, between signal enabling conditions and focused sub-components in such modes.

Note 2: we notice that M set do not trigger inner sub-modules since they are living at the mean time within the component. This is full allowed since there is no overlapping between each allocated components.

Example 3: Consider following system expected to perform a 32-bit addition. Some components are used to build this system over existing components, each of them are evolving with different endian contexts. The table after, describes how 32-bit raw data, op1 = op2 = 0x0A0B0C0D represented in standard representation, are evolving throughout this system to produce awaited right result 0x1416181A (standard representation).

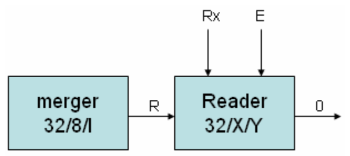
Column "E" highlights which endian-conversion was requested: little-to-big, big-to-little or big-to-big. We notice that big-to-big endian conversions may happen, targeting data with different sub-width containers: in this case a byte-swap within containers is performed (column "BS"). Little-to-little endian conversion only request data repackaging without modifying byte order.



id	E	BS	data	description	Data layout within targeted component 1st row orders containers, 0 is affected to least significant (LS) container			
op1	yes	no	0x0A0B0C0D	Raw 1st operand data	0		1	
					0x0C0D		0x0A0B	
op2	yes	no	0x0A0B0C0D	Raw 2nd operand data	0		1	
					0x0C0D		0x0A0B	
A	yes	no	0x0C0D_0A0B	1st operand transmitted to 1st splitter	0		1	
					0x0A0B		0x0C0D	
B	yes	yes	0x0D0C_0B0A	2nd operand transmitted to 2nd splitter	0	1	2	3
					0xA	0xB	0xC	0xD
C	no	yes	0x0D0C	1st operand LS16-bit sent to 1st 16-b add	0		1	
					0x0C		0x0D	
D	yes	no	0x0A_0B	2nd operand MS16-bit sent to 2n	0			

				16-b add	0x0B0A			
E	no	no	0x0D_0C	2nd operand LS16-bit sent to 1s 16-b add	0		1	
					0x0C		0x0D	
F	no	no	0x0A0B	1st operand MS16-bit sent to 2n 16-b add	0			
					0x0A0B			
G	yes	no	0x1A_18	computed LS16-bit	0	1	2	3
					0x1A	0x18	0x00	0x00
H	no	no	0x1416	computed MS16-bit	0	1	2	3
					0x16	0x14	0x00	0x00
i	-	-	0b0	1-bit carry	-			
R	no	no	0x14_16_18_1A	result	0	1	2	3
					0x1A	0x18	0x16	0x14

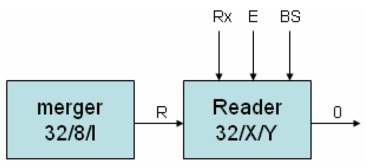
Assuming now a component Reader is plugged after the component Merger providing a way to access R data in various formats. We consider 0x1416181A is carried out by R. The table below shows how R is transformed according to variations on (X, Y, Rx, E) to output O. First Rx operation is performed, returning first x bits from bit 0, after having performed E-endianness switch, if needed, on (X, Y).



Rx	E	(X, Y), X in bits, Y is either little-endian (l) or big-endian (b)					
		(8, l)	(8, b)	(16, l)	(16, b)	(32, l)	(32, b)
R 8-bit	*little*	0x1A	0x1A	0x1A	0x18	0x1A	0x14
R 16-bit	*little*	0x181A	0x181A	0x181A	0x1A18	0x181A	0x1614
R 32-bit	*little*	0x1416181A	0x1416181A	0x1416181A	0x16141A18	0x1416181A	0x1A181614
R 8-bit	*big*	0x14	0x14	0x16	0x14	0x1A	0x14
R 16-bit	*big*	0x1614	0x1614	0x1416	0x1614	0x181A	0x1614
R 32-bit	*big*	0x1A181614	0x1A181614	0x181A1416	0x1A181614	0x1416181A	0x1A181614

We notice that according to X, outputted data O can bring 3 different values (see colors): bytes layouts are different.

We add now a byte-swap input BS as shown in the picture below, computing the following table. Bytes are swapped within X ranges, when an endian conversion is required (cells in bold within the table).



Rx	E	(X, Y), X in bits, Y is either little-endian (l) or big-endian (b)					
		(8, l)	(8, b)	(16, l)	(16, b)	(32, l)	(32, b)
R 8-bit	*little*	0x1A	**0x1A**	0x1A	**0x1A**	0x1A	**0x1A**
R 16-bit	*little*	0x181A	**0x181A**	0x181A	**0x181A**	0x181A	**0x181A**
R 32-bit	*little*	0x1416181A	**0x1416181A**	0x1416181A	**0x1416181A**	0x1416181A	**0x1416181A**
R 8-bit	*big*	**0x14**	0x14	**0x14**	0x14	**0x14**	0x14
R 16-bit	*big*	**0x1614**	0x1614	**0x1611**	0x1614	**0x1614**	0x1614
R 32-bit	*big*	**0x1A181614**	0x1A181614	**0x1A181614**	0x1A181614	**0x1A181614**	0x1A181614

We notice that byte-swap within a sub-width range when an endian conversion is required align outputs regardless of its inner data layout representation. Conversion table can hence be reduced into:

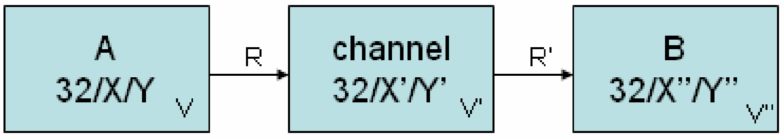
Rx	E	(X, Y), X in bits, Y is either little-endian (l) or big-endian (b)
R 8-bit	*little*	0x1A
R 16-bit	*little*	0x181A
R 32-bit	*little*	0x1416181A
R 8-bit	*big*	0x14
R 16-bit	*big*	0x1614
R 32-bit	*big*	0x1A181614

More generally, byte can be swapped within a sub-width range whenever needed, independently of any endianness consideration, nor any endian needed conversion. We used this byte-swap feature above to compute outputted data removing data storage layout within "Reader" component to avoid any misinterpretation leading to drastic side-effects in the interpretation of the read data.

Each time byte-swap BS is enabled, we can speak about *middle-endian*, however this confusing term does not mention whether basic component Endianness context was either little-endian, or big-endian, leading to misinterpretation of data as well.

Exchanged Data Study

We will study here communications between a couple of 32-bit components A and B, where sub-widths (X and X'') and endianness (Y and Y'') may vary independently within both. Communication channel may be as complex as we want, and hence will be tweaked with different sub-width (X') and endianness likewise (Y').



X', Y'	V (X, Y) = (8-bit, little)	V'	V'' (X'', Y'') = (8-bit, little)
8-bit, little	0x14_16_18_1A	0x14_16_18_1A	0x14_16_18_1A
16-bit, little	0x14_16_18_1A	0x1416_181A	0x14_16_18_1A
32-bit, little	0x14_16_18_1A	0x1416181A	0x14_16_18_1A
8-bit, big	0x14_16_18_1A	0x1A_18_16_14	0x14_16_18_1A
16-bit, big	0x14_16_18_1A	0x181A_1416	0x14_16_18_1A
32-bit, big	0x14_16_18_1A	0x1416181A	0x14_16_18_1A

Comment: The channel is aware of A features and is hence able to compute data provided by A in functions of A and its endianness features. According to its sub-width, the channel will thereafter pack data. In a similar way, B can act in the same way getting data from the channel.

Note 1: We can notice for (X', Y') = (16-bit, big) that little-endian data V are packed in 16-bit packets and then swapped (endian-conversion). Data contained within these packets are not swapped, despite they were in an 8-bit sub-width layout representation (X' = 8-bit).

Note 2: we notice that an endian-conversion is a reflexive operation: (X, Y) = (X'', Y'') implies V = V'', whatever communication channel may be.

Little-Endian towards Big-Endian

Case: Z/ZS/l to Z'/ZS'/b

1. Incoming little-endian data are resized: either packed (ZS < ZS'), split (ZS > ZS') or unchanged (ZS = ZS')
2. Endian conversion (l to b) is applied on data gathered in [1]

Big-Endian towards Little-Endian

Case: Z/ZS/b to Z'/ZS'/l

1. Endian conversion (b to l) is applied on incoming big-endian data, swapping ZS sub-width packets
2. Data computed in [1] are resized: either packed (ZS < ZS'), split (ZS > ZS') or unchanged (ZS = ZS')

Little-Endian towards Little-Endian

Case: Z/ZS/l to Z'/ZS'/l

1. Incoming little-endian data are resized: either packed (ZS < ZS'), split (ZS > ZS') or unchanged (ZS = ZS')
2.

Big-Endian towards Big-Endian

Case: Z/ZS/b to Z'/ZS'/b

1. ZS = ZS' : incoming big-endian data are unchanged
2. ZS < ZS' : incoming big-endian data are packed and modified
    a. ZS-width packets are gathered into ZS'-width bigger packets
    b. ZS-width packets within ZS'-width packets are swapped

3. ZS > ZS' : incoming big-endian data are modified and unpacked
    a. Each ZS'-width packet is split into ZS-width packets
    b. ZS-width packets are swapped within ZS'-packets
    c. All ZS'-width packets are unpacked

Follow a few words about byte-swap operation BS_n in $Z_n/ZS_n/E_n/BS_n/bS_n$. Endian conversions can be achieved according to algorithms presented above when a couple of components are communicating full aware of endianness of the other. In this way either the master or the slave can compute data in the right way.

Moreover, components can be designed in an Endian-free, or Endian-neutral, way, without any Endian information (little or big) provided by the component. Endian-free way is a strong forte for bi-endian components, avoiding to set up a common context, or avoiding to mess up usage of the component letting integrators mixing up what is what. Endian-free systems propose multiple Endian-tagged interfaces enabling integrator to plug components on the right interface natively fitting plugged components Endianness features.

To complete this goal, we saw that bytes may be swapped or not within sub-width packets according to data sub-width callees. Provided interfaces might hence be:

1. $Z_n/ZS_n/l_n/nS_n/bS_n$ — ZS_n sub-width is kept as it for little-endian (no swapped operation is done)
2. $Z_n/ZS_n/b_n/nS_n/bS_n$ — ZS_n sub-width is kept as it for big-endian (no swapped operation is done)
3. $Z_n/ZS_n/E_n/S_n/bS_n$ — ZS_n sub-width is fictitiously kept since swap operation is done

Note 1: First couple of cases is defined with algorithms presented above

Note 2: 3rd case is more ambiguous as it since keeping data coherency using this feature depends of both Endianness and sub-widths. To avoid any issue, we suggest to set up ZS to 8-bit, which is the unit targeted by byte-swap feature. Callers will hence have to resize these 8-bit based data according to their own sub-width.

Note 3: This note is a consequence of the note above. We may remove byte-swap feature providing every interface for various ZS. However this remark is not really consistent since we cannot foresee future usage of designed system especially for unforeseen ZS (64-bit, 128-bit, …).

Note 4: bit-swap bS does not influence Endianness-based features, but will be relevant as soon as we ask "where is first bit within a byte".

Summary

We enriched system model (S_n, R_m, $W_{m'}$, $I_{m'}$, RST_r, V, B, O, M, T, Z_n) with information targeting data layout. Enhanced definition is:

$$(S_n, R_m, W_{m'}, I_{m'}, RST_r, V, B, O, M, T, Z_n/ZS_n/E_n/BS_n/bS_n)$$

With

Z_n	an ordered set of component size triggered by S_n
ZS_n	an ordered set of component sub-width triggered by S_n
E_n	an ordered set of component Endianness triggered by S_n. It can be either little-endian (l) or big-endian (b), or Endian-free (X). Endianness is based on ZS_n
BS_n	an ordered set of component byte-swap triggered by S_n, allowing full Endianness handling between components exchanged data, not covered by local Endianness handled thru sub-width settings. It can be either byte-swapped (S), or not byte-swapped (nS) or byte-swapp-free (X)
bS_n	an ordered set of component bit-swap triggered by S_n stating which bit is the first one within an atomic sub-width or byte item. It can be either bit-swapped (S), or not bit-swapped (nS) or bit-swapp-free (X)

An Endian-free context (based on $E_n/BS_n/bS_n$ features) relies on sub-components Endian context. Once computed a component cannot stay in a full Endian-free context. This means, either from each sub-component Endian features a common Endian sub-set can be inferred at module-level, or component Endian features are left blank under the assumption sub-components have their own exclusive Endian features compatibles.

Endianness features, covered by ($S_n \rightarrow Z_n/ZS_n/E_n/BS_n/bS_n$), allow compositional architecture of heterogeneous systems designed within their context or Endian-scope, without loosing information about

- data layout which may vary according to switches or built-in chosen technology
- address arithmetic which may vary according to switches or built-in chosen technology
- bi-Endian components (like ARM), and more generally about Endian-free components

An addition (+) will be designed with a little-endian approach, whilst a logics less (<) will be more efficiently designed with a big-endian approach.