Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de la Matanza
ISSN: 2525-1333
Vol.: 7 - Nro. 1 (AGOSTO 2022)

Artículo original

Penalidades en lecturas no alineadas dentro de Microcontroladores RISC-V

Misaligned memory loads penalties in RISC-V Microcontrollers

Edgardo Gho

Universidad Nacional de La Matanza
Grupo de investigación en lógica programable
egho@unlam.edu.ar

Universidad Abierta Interamericana
EdgardoAlberto.Gho@alumnos.uai.edu.ar

Resumen:
La arquitectura RISC-V fue concebida con el fin de evitar los problemas de sobrecarga de instrucciones de las arquitecturas x86 y ARM. Su definición es abierta dejando librado los detalles de la microarquitectura al diseñador del procesador. Las implementaciones de microcontroladores RISC-V se comportan de manera distinta en cuanto a los accesos a datos de forma no alineada. Si bien los compiladores buscan evitar este tipo de accesos, determinadas estructuras de datos requieren los mismos en ámbitos donde la memoria es limitada. En este artículo se estudia la implementación de tres microarquitecturas RISC-V en cuanto a los accesos a

memoria no alineados y se plantea un código que permite salvar la ejecución de programas que realizan accesos no alineados cuando la microarquitectura no tiene soporte para los mismos. En los casos donde la microarquitectura soporta accesos no alineados se estudia el impacto en la eficiencia de ejecución de instrucciones.

Abstract:
The RISC-V instruction set architecture is designed to overcome current problems of bloated instruction sets present in other architectures like x86 and ARM. The ISA does not restrict micro architecture implementations leaving those details free to the chip designer. RISC-V microcontrollers can then provide support for misaligned memory loads or depend on software emulation. Compilers will try to prevent these types of memory accesses nevertheless some data structures require them mostly on memory constraint systems. This article studies three different RISC-V microarchitecture implementations related to misaligned accesses and provides code to perform software emulation when the microarchitecture does not support them. Instruction execution penalties are studied comparing them to aligned memory accesses.

Palabras Clave: Arquitectura de computadoras, RISC-V, estructura de datos
Key Words: Computer architecture, RISC-V, data structures
Colaboradores: Carlos Maidana, Jair Hnatiuk

I. INTRODUCTION

A computer is composed of three main components, being the Central Processing Unit (CPU), Memory Unit and Input/Output(I/O). Memory content is defined by the user depending on the application. Programs designed to be executed by the CPU need to be stored in memory for the CPU to access and execute. This is done by storing the program instructions on a memory section, which the CPU accesses sequentially during the execution of the program. Program data, usually referred to as variables, are also stored in memory and accessed by the CPU in the same manner. Computer architecture defines a data unit named *word* which represents the size of a single unit of information. The unit size is usually tailored to the application. Most modern computer architectures [1] have *word* sizes larger than one byte, being 4 bytes (32 bits) and 8 bytes (64 bits) the more common. Some computer architectures, in particular Harvard-based ones, define different memory ranges to differentiate between program instructions and program data. These ranges need to be known in order to properly use the computer. This information is usually represented on a memory map which lists the ranges where a program instructions and data can be stored. The memory addressing is part of the computer architecture and virtually all computer architectures [2] use byte addressing. In essence, byte addressing allows the user to reference a single byte inside a *word*. Nevertheless, memories are organized as block arrays and are meant to be accessed one block at a time. The block size is related to cache memory line size [3], in particular L1. So even though CPUs support byte addressing, the program information (being instruction or data) is usually word aligned, so much so that some CPU architectures enforce a restriction preventing memory accesses that cross the *word* boundary. These types of accesses are usually referred to as unaligned or misaligned accesses.

In [3] the authors study the impact of different unaligned memory accesses on different x64 ISA (instruction set architecture) instructions. Depending on the boundary crossed the time penalty can be up to 1800%.

The RISC-V is an open standard instruction set architecture that is not bound to a particular implementation [4]. This means that while the computer architecture is standardized and clearly defined, manufacturers have no restrictions on how to implement the ISA. While this gives an enormous amount of freedom to the designer it can create fragmentation among different implementations of the same ISA [5]. Being an open architecture RISC-V allows microarchitecture changes to improve security while maintaining the base instruction set compatibility.

This paper studies the microarchitecture differences within memory access alignment in three different RISC-V CPUs. Although the three CPUs implement the same basic RV32I instruction set as a minimum, some implement a more complete RV32IMAC set including integer multiplication/division, atomic access and compressed instructions. The focus of the paper is comparing how the three handle misaligned load access to data memory. As a result, we propose a simple code to handle misaligned loads for *word* size data, both in RV32I and RV32IMAC sets. A benchmark is performed to measure the execution time penalty.

It is important to highlight that most compilers will try to prevent misaligned access to memory. One known case is the GCC [6] compiler which adds padding to data structures in order to keep load access to the data structure element aligned. Even though this will prevent misaligned access, the padding added to the data structure consumes memory. On CPU implementations designed for embedded systems and not general purpose computers memory is usually constrained and using padding in data structures will waste this resource.

II. BACKGROUND

A. RISC-V

The base RISC-V ISA [7] is a fixed 32 bit instruction length. However, it is designed to be extended to a variable length set using 16 bits parcels. Therefore, instruction alignment needs to be naturally aligned to the 16 bit parcel size. If the implementation is restricted to the basic RV32I set, then a 32 bit natural alignment is needed for instructions, although it is very common to find RV32IC which supports compressed instructions changing the requirement to a 16 bit natural alignment. In fact, the macro-op fusion benefits [8] from using the compressed instructions to achieve better performance than other ISAs like x86 or ARM.

Data access on the other hand is byte addressed and there are no restrictions by the ISA in terms of alignment. The authors describe that for best performance data access should be naturally aligned with the data size. This means that words (loaded with LW) should be aligned to 32 bits and half words (loaded with LHW or LHU) should be aligned to 16 bits. Even though this is suggested for best performance, it is ultimately up to the chip designer to support misaligned access. The RISC-V ISA does not impose a restriction on the microarchitecture, so chip designers are free to choose to support it or not. Nevertheless, the ISA defines a trap mechanism to emulate the access in software. This means that it is ultimately possible to support these types of misaligned access but with a time penalty.

B. Compilers

A typical compiler like GCC is normally aware of the microarchitecture implementation allowing the programmer to remain ignorant about it. In the case of RISC-V, there is a set of registers accessible using the CSR* instructions which describe the microarchitecture. The *misa* register will describe which of the extensions the CPU supports. Therefore, it is possible for a compiler to prepare code for several microarchitectures and select the correct one during runtime. Unfortunately, this will create bloated binaries [9] which are not memory efficient on memory constrained devices like microcontrollers. A clear example of this is the M extension. This extension provides hardware implementation for multiply and divide instructions which should be faster than a software implementation. Hence the compiler can prepare the code to detect if the M extension is supported during runtime and implement a software emulation alternative in case it does not. To facilitate this process the RISC-V architecture provides traps for invalid instructions. This would allow the compiler to provide a trap handler that would handle the missing instructions on the basic RV32I set. There is of course a penalty for this but other than the trap handler the original code remains the same.

The same technique can be used to handle other faults related to the microarchitecture. In this paper a proposed software emulation for a misaligned load is provided.

```
struct Data {
    uint8_t   A;
    uint32_t B;
    int16_t   C;
    int8_t     D;
    uint8_t   E;
};
```

Fig. 1 Misaligned Data Structure

C. Data Structures

It is normal for programmers to group related information using data structures. The C programming language struct is a good example of this. Looking at Figure 1, the data structure is composed of an 8 bit unsigned integer named A, a 32 bit unsigned integer named B, a 16 bit signed integer named C, an 8 bit signed integer named D and finally an 8 bit unsigned integer named E.

If the programmer instantiates an element with this data structure the compiler will be aware that the access to certain elements will be misaligned in 32 bits systems [10]. If the structure is stored on a byte addressing computer in a way that naturally aligns the first element (A), it will produce a misaligned access to the B and C elements.

A programmer that is aware of the microarchitecture limitations has the option to arrange the data in a way to force natural alignment for the elements that are bigger than 8 bits. So, Figure 2 would be a possible re-ordering of the original structure in a way that all access to the structure elements are naturally aligned.

```
struct Data {
    uint32_t B;
    int16_t   C;
    uint8_t   A;
    int8_t     D;
    uint8_t   E;
};
```

Fig. 2 Aligned Data Structure

Unfortunately requiring this change would be cumbersome to the programmer since it needs to manually align the elements of the data structure, but it's still not good enough. The re-arrangement will work for a single instance of the structure assuming the instance is stored in a naturally aligned address but since the number of bytes composing the structure is odd, it will not allow saving two consecutive instances like an array. The second instance will be misaligned.

For this reason, compilers will not require programmers to naturally align elements inside a structure and will provide a solution to the alignment access by adding padding to the structure. On Figure 1, the compiler will reserve three consecutive bytes after element A effectively expanding A to be 32 bits. That way the structure is now naturally aligned for all elements access but now it uses 12 bytes instead of the original 9 bytes. This is a +33% increase penalty to keep alignment. Furthermore, if the structure would be composed only by element A and B, the added padding would increase the original structure to a +60%.

Compilers provide ways to restrict padding by forcing the data to be packed, which does not add any padding, but it does not enforce alignment. It is up to the programmer to choose whether to save memory space or improve speed by avoiding misaligned access which might incur in time penalties.

III. RELATED WORK

The Linux kernel recently added support for software emulated misaligned accesses [11], but this implementation requires running the Linux kernel which is not always an option specially on systems that have no

Table 1 RISC-V Microcontrollers

CPU	**FE310-G002**	**ESP32C3**	**RISC-Vp**
Execution	In-order pipelined	In-order pipelined	In-order no pipelined
Extensions	RV32IMAC	RV32IMC	RV32I
Memory	16KB data. 16KB instruction. Instruction memory expandable using external Flash.	400 KB shared for data and instructions. Instruction memory expandable using external Flash.	Up to 540KB using FPGA BlockRAM shared data and instruction. No expansion.
Misalignment	Software emulation	Hardware support	Hardware support

MMU and have fixed limited memory. Microcontrollers like the ones studied on this paper usually implement software in bare-metal or using a RTOS. The provided code in this paper can be easily adapted to these chips without imposing a full Linux kernel implementation.

The authors in [12] study the effects of misaligned memory access in microcontrollers but it is not restricted to RISC-V. Their study also includes ARM and MIPS architectures. Their study is restricted in some cases to byte access since not all platforms allow misaligned access for words. This paper focuses on word load penalty when address is misaligned and provides software emulation when hardware does not support this type of access.

Misaligned accesses become important based on [3] Heap randomization can be improved in architectures that support misaligned accesses.

IV. EVALUATION

Three different RISC-V microarchitectures were studied in terms of their respective misaligned loads. All three processors are oriented to microcontroller applications, therefore memory is usually fixed size and as a scarce resource it is important to maximize its usage. Hence using padding in data structures is not recommended. Table 1 represents the instruction set extensions, memory size and microarchitecture details of the three processors.

A. FE310-G002

This RISC-V microcontroller [13] only supports software emulated misaligned loads of words. The code present in Figure 3 shows how to force a misaligned load using pointers.

The C compiler translates the misaligned load of pWord pointer into loadVariable with the ASM listed on Figure 4.

The middle ASM instruction in the group is the actual misaligned load. Since it's a full 32 bit word load with offset 0 based on the address pointed by a5 into a5, this instruction can be compressed into 16 bits.

In order to test time penalties, a trap handler was written to software-emulate the misaligned access. The code for

```
struct Data {
   uint32_t A;
   uint32_t B;
};
struct Data data;
data.A=0x12345678;
data.B=0x90ABCDEF;
uint8_t *pByte = (uint8_t*)&data;
uint32_t *pWord = (uint32_t*)(pByte+4);
uint32_t loadVariable = *pWord; //Aligned
pWord = (uint32_t*)(pByte+1);
loadVariable = *pWord; //Misaligned
```

Fig. 3 Misaligned pointer access

the handler is provided in [14]. This handler only supports LW instructions, although it can be easily extended to handle LH and LHU misaligned access as well. The handler supports both RV32I and RV32IC extensions, detecting at runtime if the offending instruction is LW or C.LW (compressed LW).

The handler reads the *mcause* register to check if it is a load alignment issue. If that is the case, then it recovers the address of the offending instruction from the *mepc* register. Since the instruction can be either LW or C.LW, it can be aligned to 32 or 16 bits, so the handler supports the possible misaligned instruction access. Upon reading the instruction it calculates the return address for the trap (done later with *mret*) and the destination register for the load. In order to emulate the access, several registers are temporarily used, and their previous content is saved on the stack. Upon return the original content of those will be recovered, except for the destination register of the misaligned access which should have the misaligned data. This is true for all cases except when the destination register is the stack pointer itself. In this case, there needs to be an extra scratch register or fixed memory location

```
lw    a5, -24(s0)
lw    a5,    0(a5)
sw   a5, -28(s0)
```

Fig. 4 ASM translation

used to store the misaligned value prior to calling *mret*. In the case of this handler, it will scratch the T5 register, but this can be modified to use a fixed memory location and avoid using T5.

The software emulation executed 92 extra instructions for the access listed on Figure 4. This number of extra instructions will depend on how the original offending instruction is coded, what type of misalignment it performs and most important what is the destination register, being the higher ones from x0 to x31 the worst ones.

B. ESP32C3

This RISC-V microcontroller [15] supports hardware misaligned loads. There is no need to provide a trap handler since the hardware supports the misaligned access. Since ESP32C3 has a pipelined microarchitecture with data and instruction caches, using the cycle performance counter (CSR 0x7e2 for this chip) might yield different values depending on the pipeline stage and cache state. Therefore, a loop doing 100 aligned access followed by another loop with 100 misaligned access were executed while saving the cycle performance counter. The results yielded 807 cycles for aligned access and 1906 cycles for the misaligned ones. These numbers include the loop instructions and performance counter access overhead but since it is the same overhead on both loops the comparison is still valid. The misaligned access incurs a 137% penalty.

Table 2 Results

CPU	**FE310-G002**	**ESP32C3**	**RISC-Vp**
Penalty	9200%	137%	75%
CPI	1	1	4
C100LW	9300	237	700

C. RISC-Vp

This RISC-V implementation [16] also supports hardware misaligned loads. Since this microarchitecture was designed as an academic example of a RISC-V implementation it is possible to predict the penalties in misaligned access.

A proper aligned load uses 4 clock cycles while a misaligned load consumes 7 clock cycles. This is a 75% penalty when compared with an aligned load. This microarchitecture executes on a fixed clock per instruction since it has no pipeline and no cache, therefore there is no need to execute loops or use performance counters.

V. CONCLUSIONS

Table 2 shows that software emulation can result in a big time penalty versus the hardware implementation. The penalty for FE310-G002 is estimated since the trap handler efficiency varies depending on the offending load. Both LW and C.LW were tested. The CPI (clocks per instruction) represents the non-pipelined architecture of RISC-Vp with a fixed 4 clock per instruction. The CPI listed for FE310-G002 and ESP32C3 is also estimated since it would depend on the pipeline state depending on the code being executed but the goal for the pipeline would be one clock per instruction.

Even though RISC-Vp has a lower time penalty than ESP32C3 which also supports hardware misaligned loads, ultimately the number of clocks per 100 misaligned loads (C100LW) ends up being less for ESP32C3 due to the pipeline implementation.

The result clearly indicates that software emulation of misaligned load can incur severe time penalties affecting performance. Systems that require misaligned access should try to select a microarchitecture that can perform these without software emulation.

The provided sample code [14] can be expanded to support misaligned stores. This code is a good example of software emulation for missing microarchitecture features proving that RISC-V was designed in a way that simpler microarchitectures can execute code for more complete implementations. In this test scenario the FE310-G002 was limited in terms of misaligned access but this limitation can be overcome via software emulation at a high time penalty.

VI. REFERENCES

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach Fifth Edition. 2007.

[2] D. M. Harris and S. L. Harris, Digital design and computer architecture, 2nd edition. 2012. doi: 10.1016/C2011-0-04377-6.

[3] D. Jang, J. Kim, M. Park, Y. Jung, H. Lee, and B. B. Kang, “Rethinking Misalignment to Raise the Bar for Heap Pointer Corruption.” arXiv, 2018. doi: 10.48550/ARXIV.1807.01023.

[4] K. Asanović and D. Patterson, “RISC-V: An Open Standard for SoCs | EE Times,” EE Times, 2014.

[5] Agam Shah, “RISC-V takes steps to minimize fragmentation” https://www.theregister.com/2022/04/01/riscv_fragmentation/, Apr. 01, 2022.

[6] R. M. Stallman and T. G. D. Community, “Using the GNU Compiler Collection,” Development, vol. 2. 2012.

[7] A. Waterman et al., “The RISC-V instruction set manual,” Volume I: User-Level ISA’, version, vol. 2, 2014.

[8] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, “The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V.” arXiv, 2016. doi: 10.48550/ARXIV.1607.02318.

[9] A. Singh, Mac OS X Internals: A Systems Approach (paperback). Addison-Wesley Professional, 2006.

[10] Daniel Lemire, “Data alignment for speed: myth or reality?,” https://lemire.me/blog/2012/05/31/data-alignment-for-speed-myth-or-reality/, May 31, 2012.

[11] Damien Le Moal, “[v2,1/9] riscv: Unaligned load/store handling for M_MODE” https://patchwork.kernel.org/project/linux-riscv/patch/20200312051107.1454880-2-damien.lemoal@wdc.com/, Mar. 12, 2020.

[12] M. Hubacz and B. Trybus, “Data Alignment on Embedded CPUs for Programmable Control Devices,” Electronics (Basel), vol. 11, no. 14, 2022, doi: 10.3390/electronics11142174.

[13] SiFive Inc, “SiFive FE310-G002 Manual v1p4,” 2019.

[14] Edgardo Gho, “RISC-V Traps,” https://github.com/edgardogho/RISC-V-Traps, Jul. 18, 2022.

[15] Espressif Systems, “ESP32-C3 Series Datasheet,” 2022.

[16] Edgardo Gho, “RiscVP,” https://github.com/edgardogho/RiscVP, Mar. 04, 2021.

Recibido: 2022-07-20
Aprobado: 2022-08-02
Hipervínculo Permanente: https://doi.org/10.54789/reddi.7.1.2
Datos de edición: Vol. 7 - Nro. 1 - Art. 2
Fecha de edición: 2022-08-10