Implementing Sub-word Operations in Word-oriented Architectures[*]

Geraldo Lino de Campos
Escola Politécnica da Universidade de São Paulo
São Paulo 05508-900, Brazil
Phone: + 55 11 8159322 ext 5288
e-mail: RTC@FPSP.FAPESP.BR

Abstract

This paper presents a proposal for handling sub-words (bytes, 16 and 32 bit-fields) in word-oriented architectures. The proposal is based on using addresses expressed in units of bytes, and on keeping the low-order bits of the address attached to words read from memory. These bits are used by especial instructions to do the extract/insert operations. The solution holds for any unaligned accesses to sub-words even when the architecture supports only aligned accesses to words. Performance implications for machines with or without caches are examined, and extensions to arbitrary bit fields are proposed.

Keywords

byte operations, partial words, caches

* This research was partially supported by the CNPQ grant 501971/91-6

1 Introduction and notation

The current trend in high-performance computers is to define word-oriented architectures ; this trend can be seen, for instance, in the Digital’s ALPHA Architecture [DEC92]. This trend is motivated by performance requirements, since the existence of an alignment network contributes to lengthening the clock cycle. The same performance requirements dictates large word sizes, typically 64 bits.

Operations on smaller data fields are still required, however. We shall distinguish between arbitrarily sized fields, usually called subfields, and sizes that are multiples of bytes, that we will call subwords in this paper. This paper is concerned only with subwords, although most of the ideas can be extended to handling arbitrary subfields, if the architecture is properly designed.

The most relevant type of subword is the byte, for its importance in character operations. However, it can be anticipated that 2 bytes fields will become equally important with the future adoption of UNICODE for representing characters in an international environment, and a general mechanism encompassing one, two or four bytes subwords are required in other contexts. The implementation proposed in this paper cover all those cases.

Many architectures offer only partial solutions to the handling of subwords. ALPHA, for instance, only offers a good solution for the case of fully aligned strings, which certainly is not the general situation, since there are no published data supporting the concept that compilers can always make strings word-aligned. The possibility of unaligned loads alleviate somewhat this problem, but still it is not a general solution.

In this paper, word will be used for designing the physical word of the machine, supposed with at least 64 bits, with doubleword meaning two consecutive machine words, and byte, dibyte, tetrabyte, ... to refer to subwords of size 8, 16, 32, ... bits. A memory access to a subword s in memory address x will be called aligned if x mod s = 0, and unaligned otherwise.

In this paper an implementation of subword operations based on keeping the low-order bits of the address with every word fetched from memory is presented. The implementation proposed does not use dedicated alignment networks, but uses specific ALU operations, built upon shifts and logical operations that are present in almost any conceivable ALU, and allows unaligned references to subwords even if the architecture supports only aligned word references. The key elements are:

- use addresses expressed in bytes, even in word-oriented architectures;
- ignore the low-order bits of the address when doing a memory fetch for a word, but preserve and attach them to the data received from memory;
- this lower-order address bits are used by especial instructions to extract/insert the sub-word; they are hidden for other instructions, except that they are copied by move register instructions.

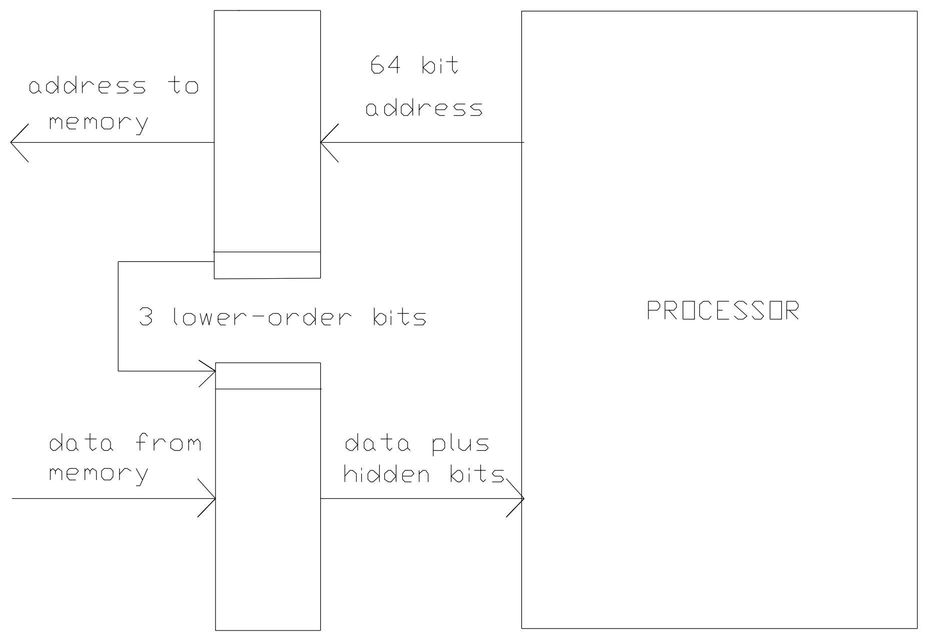

Figure 1 - Proposed architecture

In the case of 64 bit word size, when a request to address x is sent to memory subsystem, the 3 lower bits are kept, and the remaining bits are effectively sent to the memory subsystem. When the corresponding data arrives, it is tagged with the 3 lower-order bits of the original address, and sent to the specified destination. These bits will remain hidden, will be called index and referred as I (Rx), where Rx is the register number , and will be used only by the prosed ExtractX and InsertX instructions. In these instructions, which will be detailed later, X specifies the number of bits extracted/inserted by the instructions. In most of this paper the allowed values of X are restricted to powers of 2, starting at 8 (8, 16, 32, ...). For sake of simplicity, the word size will be considered to 64 bits in all examples.

Fields within words will be described as [a : b], where a designates the leftmost bit affected, and b is the number of bits. The least significant bit is 0. The notation A << B means A shifted left logically by B bits. ONES will stand for a word with all bits on.

Other menmonics for instructions are taken from the machine DLX, described in [Hen90].

This paper is organized as follows: section 2 presents the working of the proposed instructions in a machine which uses a two-way associative cache and aligned references, showing the basic principle; section 3 studies the more complex cases of write-thru caches and decoupled access/execute architectures. Section 4 concentrates cost considerations. Section 5 discusses the extensions for unaligned references and arbitrary partial words, and section 6 presents some concluding remarks.

2 The basic mechanism

In this section it supposed that all data is aligned, and that there is a two-way associative write-back cache, so repeated reads or writes to the same location take only one clock cycle. These restrictions will be eliminated in later sections.

To implement subword operations, the lower-order bits of the address are kept when the effective addresses sent to the memory (including cache) subsystem. These lower order address bits are appended to the word read; these bits will be hidden for all instructions except ExtractX and InsertX, and instructions that only move registers. Usually, immediately after reading the partial word will be extracted by a one of those instructions. Figure 1 shows the general mechanism of appending the lower order address bits to the read words.

The instruction ExtractX will use the index bits for extracting the addressed field, using the already present ALU logic. Different instructions should be provided for extracting fields of different sizes (8, 16, 32, ... bits). Depending on details of the architecture, we may have signed and unsigned options for each size.

The functionality of the ExtractX Rs, Rd instruction is:

Rd = Rs [I (Rs) * 8 + (X-1) : X]

A typical loop for comparing null-terminated strings of bytes starting at arbitrary addresses in R1 and R4 would be

```
Test:          LOAD         (R1),R2
               Extract8     R2, R3
               LOAD         (R4),R5
               Extract8     R5, R6
               CMP               R3, R6
               BNE               DifferentStrings
               CMP               R3, 0
               BZ                EqualStr
               INC               R1, 1
               INC               R4, 1
               JUMP         Test
EqualStr: ...
```

By changing Extract8 and incrementing by the proper amount, the above code compares strings of any subword size.

After the first read of a word, subsequent accesses will be from cache, and the code will not incur in any special time penalty for the repeated readings.

Write operations follow the same general pattern, but the destination word must be read in every execution of the loop, since this is cheaper than testing for word boundary, due to the fact that the destination word is already on the cache.

The functionality of the InsertX Rb, Rs, Rd instruction is

Rd = Rb AND ((ONES AND 0 [X - 1 : X] << I(Rb)*8) OR (Rs [X-1:X] << I(Rb)*8))

In the example below, the null-terminated string of subwords of size X at an arbitrary address pointed to by R1 is copied to the address contained in R4:

```
Copy:       LOAD        (R1),R2
            ExtractX        R2,  R3
            LOAD        (R4),R5
            InsertX         R5, R3, R5
            Store           (R4),R5
            BZ              Copied
            INC             R1,size(X)
            INC             R4,size(X)
            JUMP        Copy
Copied:     ...
```

3 System considerations

From a software standpoint, the above solutions work with any architecture. The efficiency, however, may be quite different for different memory subsystems. In section 2, the underlying architecture was supposed to have a two-way associative cache, where the proposed solution fits naturally, since there are no penalties for reading and writing repeatedly to the same location. In this section, we will examine the situation with other architectu-res, where reading or writing continuously on the same location may cause performance problems.

The following cases are worth considering:

3.1– Conventional cacheless architectures

This case is almost irrelevant today, when even microprocessors have caches. In this case, every subword access requires a memory operation; nevertheless, since this is exactly what would happen if the architecture

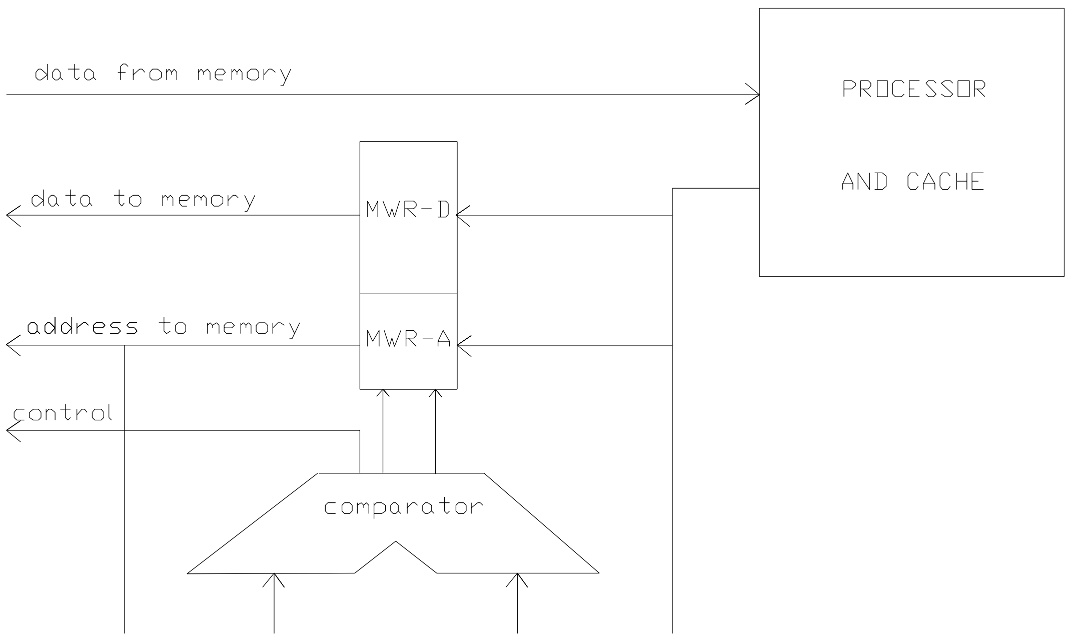

Figure 2 - Structure for avoiding repeated writes

allowed partial access to memory, there is neither improvement nor worsening in performance.

3.2– Architectures with write-back caches

This is the case used as example in section 2; since write-back caches send their contents to main more only upon demand, the process of repeated writes does not pose any additional problem.

3.3– Architectures with write-through caches

The proposed solution may be quite inefficient when writing strings on machines with pure write-through caches, since repeated writing to the same location will generate repeated writes to the same location in main memory, causing the processor to stall waiting for the memory subsystem. This may require some additional hardware, again with no impact on the cycle time.

This can be solved by including an intermediate register in the write path from the cache to the maim memory, as shown in figure 2. Every time a word is written in the cache, it is also copied to MWR-D, and the memory address is written to MWR-A, but these values are not sent to the memory subsystem. In subsequent writes, the memory address is compared to the address already present in the MWR-A; if they are different, the values present in MWR are sent to the memory subsystem; otherwise, they are simply updated.

If the machine already has a write queue, is should be easy to implement this mechanism; the details depend on the exact architecture of the write queue.

3.4– Decoupled access/execute

Decoupled access/execute architectures, originally proposed in [Smi84], try to anticipate memory accesses as much as possible to hide memory latency, and usually uses this mechanism instead of a cache. If a cache is used anyway with a decoupled access/execute architecture, the problem is already solved by one of the above mechanisms; otherwise, mechanisms should be designed to handle efficiently the repeated reads and writes to the same location.

One of the key problems to be solved for a good decoupled access/execute architecture is the synchronization of reads and writes to the same memory location. When a word is written to memory, it is necessary to check if a read to the same location has been issued and is still active. In this case, the newly generated value must be written in the corresponding position of the input queue, and the value that will came from the memory subsystem should be discarded when it arrives.

If a decoupled access/execute architecture already has the above mechanism, the proposed solution will work correctly, even with overlapped strings, and the reading performance will be as good as in a system with write-back caches.

Writes can be handled in the same way as with write-trough caches, since the situation is the same.

4 Cost considerations

The proposed instructions are a straightforward extension of shift/merge instructions already present in most architectures. The hardware cost of its implementation should be a moderate increase in control logic. It is certainly less expensive then the implementation of unaligned memory references, a competitive approach.

The real cost concern is with the index bits.

This solution has been developed for the micron project, [Cam92a, Cam92b], which is a tagged architecture. Tags are used to allow out-of-order execution and delayed interrupts. When a word contains valid data, there are available bits for storing the index, and hence the cost was zero. It is also a decoupled access/execute architecture. As in any real access/execute architecture, many memory operations can be outstanding, and so a large number of bits are already required for reordering and control; adding 3 bits for the index does not add to the complexity of control, only to the size of the control word.

That is not the general case, however, and the suitability of this proposal depends on solving the problem of the index bits storage when registers must be saved or restored. Making the memory 3 bits wider for this purpose is certainly wasteful; on the other hand, using two memory words in every save/restore operation has a significant performance penalty.

Registers may be saved and stored in blocks, as in task switching, or individually under program control. Procedure calls can be done either way -

machines with register windows save registers in blocks, while machines without register windows usually saves/restores registers individually.

In machines that save and restore in blocks even for procedure calls, it is easy to add the index fields packed in one word for a block (of 16) registers, and this is a small overhead. Given the number of registers in current machines, the save/restore of individual registers is a very infrequent operation, and two words of memory can be used without a significant performance penalty. Machines with register windows seem to be the current trend, exemplified by the SPARC architecture.

The situation is different for machines that predominantly save/re-store registers individually. In this case, this proposed scheme is suitable only when a peculiar feature (like the tag in the above example) offer an economic way to save/restore the indexes.

5 Operations with unaligned subwords

If the architecture supports register pairs operating as double precision operands, the operation with unaligned subwords would be a trivial extension of the functionality of ExtractX and InsertX instructions to operate with double word operands. The algorithms will remain quite simple: load the word with the base address in the low-order register, load the next word on the high-order register and apply the extended instruction.

However, this is unlikely to occur in machines with words of 64 or more bits. Additionally, a double-word access can cause very inconvenient access violations if the first word is in the last address in a page. In this case, a more complex sequence of instructions will be required, but since each instruction is independent, the general page fault mechanism will take care of access across page boundaries.

The instructions ExtractX (with X different of byte, of course) must be functionally extended to extract only the lower part of a subword, if a part lies outside it word argument, and must be complemented by a UpperExtratcX, that will AND an argument and the upper portion of the subword:

The meaning of the instruction ExtractX is preserved, with the additional condition that bit fields outside the word size should be considered to be 0. The meaning of UpperExtractX Rb, Rs, Rd is

```
if (Z = 8 * I(Rs) + X - 64) > 0 then Rd = Rb AND Rs [ Z-1 : Z ] << (X-Z)
                                else Rd = Rb
```

Analogously, the meaning of UpperInsertX Rb, Rs, Rd is

```
          if (Z = 8 * I(Rb) + X - 64) > 0
                  then Rd = Rb AND ((ONES AND 0 [Z - 1: Z ] ) OR (Rs [X - 1 :  Z]))
                  else Rd = Rb
```

The following example covers the case of moving subwords between two arbitrary and possibly unaligned addresses. R1 and R4 contains the source and destination addresses, respectively.

```
Copy:     LOAD          (R1),R2 % contains 1st source word
          ADD                R1,8,R3
          LOAD          (R3),R3 % contains 2nd source word
          ExtractX      R2, R8
          UpperExtractX R8, R3, R8 % R8 contains the subword
          LOAD          (R4),R5 % contains 1st dest. word
          ADD                R4, 8, R7
          LOAD          (R7),R3 % contains 2nd dest. word
          InsertX            R8, R5
          UpperinsertX  R8, R3, R3
          Store              (R4), R5
          Store              (R7), R3
          BZ                 Copied
          INC                R1, size(X)
          INC                R4, size(X)
          JUMP          Copy
Copied:        ...
```

Once operations with unaligned subwords are provided, it is somewhat straightforward to implement operations with arbitrary bit fields, but, depending on the specific ALU design, it might be more costly in terms of ALU hardware. Obviously, to address arbitrary subfields it is necessary to use address expressed in bits and not in bytes, like the former Burroughs B1000 series. This should not be problem with address size of 64 bits.

The instruction Extract should be extended to receive the field size as an operand, with perhaps signed and unsigned variants. Unfortunately, instructions should be provided for all the intended values of X for InsertX, since it already requires two arguments: the word to operate upon and the value to insert, and most architectures can only provide two explicit arguments to an instruction.

6 Conclusions

The proposed implementation of subword operations is quite general, does not require any significant additional hardware for machines with write-back caches and register windows, and can be implemented with a modest addition of hardware on many machines with write-trough caches or decoupled access/execute architeture. The concept can be easily extended for unaligned accesses.

It’s main advantages are:

- It works with any subfield - bytes, dibytes and tetrabytes - in an uniform and consistent way. Of special importance is the ability to use UNICODE coded texts;
- It doesn’t require the strigns to be aligned, but do not preclude optimizations if the compiler know they are;
- Depending on the underlying architecture, it can be implemented at a very low hardware cost.

7 Bibliography

[Cam92a] Campos, Geraldo L. Asynchronous Polycyclic Architecture: an overview. Information Processing 92, J. van Leewen (ed), vol 1, 518-524, Madrid, sep 1992.

[Cam92a] Campos, Geraldo L. Asynchronous Polycyclic Architecture. Parallel Processing: CONPAR 92-VAPP V (Lecture Notes in Computer Science, vol 634), Springer-Verlag, sep 1992.

[DEC92] Digital Equipment Corporation, "Alpha architecture handbook", 1992.

[Hen90] Hennessy, J. L. and Patterson, D. A. "Computer Architecture: A Quantitative Approach". Morgan, Kaufmann Publishers, California, 1990.

[Smi84] Smith, J. E.,"Decoupled Access/Execute Architecture Computer Architectures",ACM Trans. Computer Systems 2(4):298-308, Nov 1984.