# The Alpha AXP, part 6: Memory access, basics

devblogs.microsoft.com/oldnewthing/20170814-00

August 14, 2017

Raymond Chen

The Alpha AXP has only one memory addressing mode: Indexed indirect.

```
; load memory to register
LDL     Ra, disp16(Rb)  ; result is sign-extended to 64 bits
LDQ     Ra, disp16(Rb)

; store register to memory
STL     Ra, disp16(Rb)
STQ     Ra, disp16(Rb)
```

In all cases, the address of the memory is `(int16_t)disp16 + Rb`.[1] As we learned when we discussed integer constants, the displacement is a 16-bit signed value, so it has a reach of ±32KB. By convention, a displacement of zero can be omitted.

The `L` suffix loads/stores a 32-bit value (long) and the `Q` suffix loads/stores a 64-bit value (quad).

If the memory address is not suitably aligned, then an alignment fault is generated. By default, Windows NT handles the alignment fault by emulating the unaligned load for you, then resuming execution. However, this is a ridiculously huge performance penalty, so you don't want to make it a habit.

Note the absence of byte and word memory access.[2] You'll have to construct those yourself. And as noted above, you'll also construct unaligned memory access yourself if you know what's good for you.

The special `_U` versions of the load and store instructions are useful for constructing byte access, word access, and unaligned memory access.

```
LDQ_U   Ra, disp16(Rb)
STQ_U   Ra, disp16(Rb)
```

These special unaligned instructions are available only for quads. They ignore the bottom 3 bits of the effective address when calculating the address to load from. For example, if the current value of the *t2* register were `0x1234`, then the instruction

```
LDQ_U    t1, 3(t2)
```

would take the value in the *t2* register, add `3` to it, resulting in `0x1237`, and then round the value down to the nearest multiple of 8, producing `0x1230`. It would then load the 8-byte value starting at `0x1230` and put into the *t1* register.

There are a set of extra computational instructions to assist in taking apart and reassembling integers from their containing quads. They are the extraction, insertion, and masking opcodes.

Here goes extraction:

```
EXTBL   Ra, Rb/#b, Rc  ; Rc =  (uint8_t)(Ra >> (Rb/#b * 8 % 64))
EXTWL   Ra, Rb/#b, Rc  ; Rc = (uint16_t)(Ra >> (Rb/#b * 8 % 64))
EXTLL   Ra, Rb/#b, Rc  ; Rc = (uint32_t)(Ra >> (Rb/#b * 8 % 64))
EXTQL   Ra, Rb/#b, Rc  ; Rc = (uint64_t)(Ra >> (Rb/#b * 8 % 64))

EXTWH   Ra, Rb/#b, Rc  ; Rc = (uint16_t)(Ra << ((64 - Rb/#b * 8) % 64))
EXTLH   Ra, Rb/#b, Rc  ; Rc = (uint32_t)(Ra << ((64 - Rb/#b * 8) % 64))
EXTQH   Ra, Rb/#b, Rc  ; Rc = (uint64_t)(Ra << ((64 - Rb/#b * 8) % 64))
```

These are weird to write out in formulas, but they are easy to explain. You want to read these mnemonics as “extract byte/word/long/quad low/high”. For example, `EXTWL` is “extract word low”.

To perform the extraction, you shift the first source parameter right (if extracting low) or left (if extracting high) by the number of bytes controlled by the second parameter. (More precisely, specified by the least significant 3 bits of the second parameter.) And then you extract the low-order byte/word/long/quad from the result.

Note that these are fully 64-bit instructions, so there is no sign extension in the `EXTLx` instructions.

For example, if *t1* is `0x7531`, then

```
EXTWH   t0, t1, t2
```

goes like this: The shift amount is 7 bytes because the least significant three bits of *t1* are `001`, and we are extracting the high part. So take the value in *t0*, shift it left 56 bits (7 bytes), and then extract the least significant 16 bits, zero-extending the result to 64 bits.

The way to think of these instructions is that the extract a byte, word, long, or quad from a 128-bit value. The “low” version extracts the value from the least-significant 64 bits of the 128-bit value, and the “high” version extracts the value from the most-significant 64 bits of the 128-bit value. Both instructions position the extracted value so the two pieces can be “or”d together.

```
high part  low part
--------- ---------
ABCD EFGH IJKL MNOP -- 16-byte value
      ^^^ ^         -- 4 bytes extracted at this position
    EXTLH EXTLL
     FGH0 000I
```

Note that this is *not* how the instructions actually operate, because there are edge conditions when the shift amount is an exact multiple of 8, but it's a nice way to help remember how the instructions work.

Anyway, with the extraction instruction, we can load a single byte of memory, even if not aligned:

```
LDQ_U  t1, (t0)
EXTBL  t1, t0, t1
```

To see how this works, let's number the bytes in a 64-bit value from least significant to most significant, zero through seven.

If *to* were `0x9731`, then the `LDQ_U` loads 8 bytes of memory from `0x9730`. The least significant byte (index 0) contains the value of the byte `0x9730`, and the next-least-significant byte (index 1) contains the value of the byte `0x9731`, which is the one we want. And by an amazing non-coincidence, the `EXTBL` instruction selects the byte at index `0x9731 & 7 == 1`, which is exactly the byte we want.

Loading signed data is a bit more work because you have to sign-extend the result. The simple way of doing this is to add

```
SLL     t1, #56, t1    ; shift byte all the way up to index 7
SRA     t1, #56, t1    ; shift it all the way back to index 0
                       ; but with sign extension
```

However, DEC recommends this alternative four-instruction sequence:

```
LDQ_U   t1, (t0)       ; load the entire enclosing quad
LDA     t2, 1(t0)      ; sneaky trick to extract at index 7
EXTQH   t1, t2, t1     ; shift the desired byte into index 7
SRA     t1, #56, t1    ; signed shift right to move to index 0
```

The basic idea here is to extract the desired byte into index 7 and then use a signed shift right to shift it into index 0 with sign extension. If we hadn't added 1 to *to*, then the byte we wanted would have shifted off the end of of the register (into imaginary index 8);[3] adding 1 means that the `EXTQH` will shift one fewer index, so that instead of shifting into imaginary index 8, the value we want ends up in index 7.

I'm guessing that DEC recommends the latter sequence because it has a slightly shorter dependency chain, but at the cost of an extra register.

Of course, if you know ahead of time what the alignment of *to* is, then you can avoid having to calculate the shift amount in *t2* and could just hard-code `( to + 1) % 8` as the second parameter to the `EXTQH`.

The standard sequence for loading an aligned word is

```
LDQ_U   t1, (t0)    ; load the entire enclosing quad
EXTWL   t1, t0, t1  ; extract the appropriate word, zero-extended
```

And if you want sign extension, you use the same trick:

```
LDQ_U   t1, (t0)    ; load the entire enclosing quad
LDA     t2, 2(t0)   ; sneaky trick to extract at index 6+7
EXTQH   t1, t2, r1  ; shift the desired bytes to index 6+7
SRA     t1, #48, t1 ; signed right shift to move to index 0+1
```

These sequences are designed for properly word-aligned addresses. But what if it's not correctly-aligned?

Let's find out. Suppose that *to* is `0x1357`. Let's say that the word pointed to by *to* is `XXYY`, with `YY` stored at `0x1357` and `XX` stored at `0x1358`. But we erroneously treat it as an aligned address and execute the standard aligned word load sequence:

```
LDQ_U   t1, (t0)    ; loads the quad at 0x1350
                    ; t1 = YY??????`????????
EXTWL   t1, t0, t1  ; shifts the value in t1 right by 7 bytes
                    ;      00000000`000000YY
                    ; and then extracts the least-significant word
                    ; t1 = 00000000`000000YY
```

Oops, we read only the least significant byte of the value; the `XX` was not loaded at all.

If you sit down and work it out, the aligned word load sequence produces the desired results for most unaligned addresses, but if the word crosses a quad boundary, the code executes without any faults but produces incorrect results. This is worse than crashing!

On the Alpha AXP, it is absolutely essential that you accurately declare the fact that a pointer may point to misaligned data. If you forget, then depending on the size of the data you are accessing and the precise nature of the misalignment, you might get away with it, or you might crash, or (worst case) you will proceed with incorrect data.

So far, we've been looking only at loading aligned bytes and words. Next time, we'll look at unaligned accesses, as well as storing bytes and words, as well as storing unaligned longs and quads.

[1] Note that the base register must be a general-purpose register. This means no PC-relative addressing, because the program counter is not a general-purpose register. This is another strike against storing constants in the code stream.

[2] Later versions of the Alpha AXP added instructions for aligned byte and word access, but Windows NT didn't require that you had one of those newer processors. Consequently, you were unlikely to encounter them in practice unless you had code that did processor feature detection and had separate code paths for processors with and without support for byte and word access instructions.

However, the proof-of-concept Alpha AXP 64-bit edition of Windows did require a processor that supported the byte and word memory access instructions, so you would see those instructions if you had to debug the Alpha AXP 64-bit verison of Windows. (And by "you" I mean "me," because that version of Windows never shipped.)

[3] Well, except that if *to* had been an exact multiple of 8, then the `EXTQH` wouldn't have shifted anything at all. But you can work it out with pencil and paper and convince yourself that the right thing happens even if *to* is an exact multiple of 8.

Raymond Chen

**Follow**