The Alpha AXP, part 8: Memory access, storing bytes and words and unaligned data

devblogs.microsoft.com/oldnewthing/20170816-00

August 16, 2017

Raymond Chen

Storing a byte and word requires a series of three operations: Read the original data, modify the original data to incorporate the byte or word, then write the modified data back to memory.

To assist with the modification are two groups of instructions known as insertion and masking.

```
INSBL   Ra, Rb/#b, Rc  ; Rc =  (uint8_t)Ra << (Rb/#b * 8 % 64)
INSWL   Ra, Rb/#b, Rc  ; Rc = (uint16_t)Ra << (Rb/#b * 8 % 64)
INSLL   Ra, Rb/#b, Rc  ; Rc = (uint32_t)Ra << (Rb/#b * 8 % 64)
INSQL   Ra, Rb/#b, Rc  ; Rc = (uint64_t)Ra << (Rb/#b * 8 % 64)

INSWH   Ra, Rb/#b, Rc  ; Rc = (uint16_t)Ra >> ((64 - Rb/#b * 8) % 64)
INSLH   Ra, Rb/#b, Rc  ; Rc = (uint32_t)Ra >> ((64 - Rb/#b * 8) % 64)
INSQH   Ra, Rb/#b, Rc  ; Rc = (uint64_t)Ra >> ((64 - Rb/#b * 8) % 64)
```

These are the inverse of the extraction instructions. Instead of extracting data from a 128-bit value, they move the data into position within a 128-bit value. For example, here's a diagram of inserting the long `FGHI` into a 128-bit value:

```
high part  low part
--------- ---------
0000 0FGH           -- INSLH
          I000 0000 -- INSLL
```

The last piece of the puzzle is the masking instructions.

```
MSKBL   Ra, Rb/#b, Rc  ; Rc = Ra & ~( (uint8_t)~0 << (Rb/#b * 8 % 64))
MSKWL   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint16_t)~0 << (Rb/#b * 8 % 64))
MSKWL   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint32_t)~0 << (Rb/#b * 8 % 64))
MSKWL   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint64_t)~0 << (Rb/#b * 8 % 64))

MSKWH   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint16_t)~0 >> ((64 - Rb/#b * 8) % 64))
MSKWH   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint32_t)~0 >> ((64 - Rb/#b * 8) % 64))
MSKWH   Ra, Rb/#b, Rc  ; Rc = Ra & ~((uint64_t)~0 >> ((64 - Rb/#b * 8) % 64))
```

These instructions zero out the bytes of a 128-bit value that are about to be replaced by an insertion.

For example, here's how the masking of a long would work:

```
high part  low part
--------- ---------
ABCD EFGH IJKL MNOP -- 16-byte value
      ^^^ ^         -- 4 bytes to be inserted here
ABCD E000           -- MSKLH
          0JKL MNOP -- MSKLL
```

Putting the pieces together, we see that in order to replace a long in the middle of a 128-bit value, you would use the insertion instructions to place the new value in the correct position, the masking instructions to zero out the bits that used to be there, and then "or" the pieces together.

```
; store an unaligned long in t1 to (t0)
; first read the 128-bit value currently in memory
LDQ_U   t2,3(t0)                    ; t2 = yyyy yyyD
LDQ_U   t5,(t0)                     ; t5 =           CBAx xxxx

; build the values to insert
INSLH   t1,t0,t4                    ; t4 = 0000 000d
INSLL   t1,t0,t3                    ; t3 =           cba0 0000

; mask out the values to be replaced
MSKLH   t2,t0,t2                    ; t2 = yyyy yyy0
MSKLL   t5,t0,t5                    ; t5 =           000x xxxx

; "or" the new values into place
BIS     t2,t4,t2                    ; t2 = yyyy yyyd
BIS     t5,t3,t5                    ; t5 =           cbax xxxx

; and write the results back out
STQ_U   t2,3(t0)                    ; must store high then low
STQ_U   t5,(t0)                     ; in case there was no straddling
```

Extending this pattern to quads and words is left as an exercise.

Notice that in the case where *t0* does not straddle two quads, we perform two reads from the same location, and two writes to the same location. Let's walk through what happens:

```
; first read the 128-bit value currently in memory
; (which is really the same 64-bit value twice)
LDQ_U   t2,3(t0)                    ; t2 = yyDC BAxx
LDQ_U   t5,(t0)                     ; t5 = yyDC BAxx

; build the values to insert
INSLH   t1,t0,t4                    ; t4 = 00dc ba00
INSLL   t1,t0,t3                    ; t3 = 0000 0000

; mask out the values to be replaced
MSKLH   t2,t0,t2                    ; t2 = yy00 00xx
MSKLL   t5,t0,t5                    ; t5 = yyDC BAxx

; "or" the new values into place
BIS     t2,t4,t2                    ; t2 = yydc baxx
BIS     t5,t3,t5                    ; t5 = yyDC BAxx

; and write the results back out
STQ_U   t2,3(t0)                    ; write same value back
STQ_U   t5,(t0)                     ; write updated value
```

This highlights some of the weird memory effects of the Alpha AXP. If another thread snuck in and modified the memory at *t0 & ~7*, those changes would be reverted at the first `STQ_U`, and then the updated value gets written next. This means that the value changes from `yyyyDCBAxx` to `zzzzDCBAww`, and then back to `yyyyDCBAxx`, and then finally to `yyyydcbaxx`. The value changes, and then appears to change back to the old value, before finally being updated to a new (sort-of) value.

We'll learn more about the Alpha AXP memory model later.

In the case where you are writing a word and you know that it is aligned, then you can avoid having to deal with the 128-bit value and operate within a 64-bit value (because an aligned word will never straddle two quads).

```
; store an aligned word in t1 to (t0)
; first read the 64-bit value currently in memory
LDQ_U   t5,(t0)                     t5 = yyBA xxxx

; build the value to insert
INSWL   t1,t0,t3                    t3 = 00ba 0000

; mask out the values to be replaced
MSKWL   t5,t0,t5                    t5 = yy00 xxxx

; "or" the new values into place
BIS     t5,t3,t5                    t5 = yyba xxxx

; and write the results back out
STQ_U   t5,(t0)
```

Okay, but what about bytes? Well, bytes can never be misaligned, so we always go through the “known aligned” shortcut.

```
; store a byte in t1 to (t0)
; first read the 64-bit value currently in memory
LDQ_U   t5,(t0)                     t5 = yyyA xxxx

; build the value to insert
INSBL   t1,t0,t3                    t3 = 000a 0000

; mask out the values to be replaced
MSKBL   t5,t0,t5                    t5 = yyy0 xxxx

; "or" the new values into place
BIS     t5,t3,t5                    t5 = yyya xxxx

; and write the results back out
STQ_U   t5,(t0)
```

Dealing with unaligned memory on the Alpha AXP is very annoying. Notice that updates to words and bytes, even aligned words, is not atomic. We read the entire quad from memory, perform some register calculations, and then write the entire quad back out. If somebody made a change to another byte within the quad, we will wipe out that change when we complete our word or byte update.

Next time, we’ll look at atomic memory operations.

Bonus chatter: There is one more pair of instructions which operate on the bytes within a register: `ZAP` and `ZAPNOT`.

```
ZAP     Ra, Rb/#b, Rc  ; Rc = Ra after zeroing the bytes selected by Rb/#b
ZAPNOT  Ra, Rb/#b, Rc  ; Rc = Ra after zeroing the bytes selected by ~Rb/#b
```

The `ZAP` and `ZAPNOT` instructions treat the low-order 8 bits of the second parameter as references to the corresponding bytes of the *Ra* register: Bit *n* of *Rb*/#b corresponds to bits *N* × 8 through *N* × 8 + 7. The `ZAP` instruction sets the byte to zero if the corresponding bit is set; the `ZAPNOT` instruction sets the byte to zero if the corresponding bit is clear. The other 56 bits of the second parameter are ignored.

For example, `ZAP v0, #128, v0` clears the top byte of *v0*, and `ZAPNOT v0, #128, v0` clears all but the top byte of *v0*. (For some reason, I had trouble remembering which way is which. My trick was to pretend that the `ZAPNOT` instruction is called `KEEP`.)

As a special case, these instructions provide a handy way to zero-extend a register.

```
ZAPNOT  Ra, #1, Rc  ; zero-extend byte from Ra to Rc
ZAPNOT  Ra, #3, Rc  ; zero-extend word from Ra to Rc
ZAPNOT  Ra, #15, Rc ; zero-extend long from Ra to Rc
```

Note that in the last case, zero-extending a negative long will result in a 32-bit value in non-canonical form. But you hopefully were expecting that; if you want to sign-extend the value (in order to ensure a value in canonical form), you would have done `ADDL Ra, #0, Rc`.

Raymond Chen

Follow