The Alpha AXP, part 2: Integer calculations

devblogs.microsoft.com/oldnewthing/20170808-00

August 8, 2017

Raymond Chen

Here are some of the integer computational operations available on the Alpha AXP. I'm going to cover only the instructions used in general-purpose programming, since that's the sort of thing you're most likely to encounter when debugging everyday application code. In particular, I'm not going to cover the various multimedia instructions.

Integer arithmetic operations come in two flavors, one that operates on the full 64-bit register, and another that operates on the least significant 32 bits of the value. As I noted earlier, the general rule in the Alpha AXP is that if the result of an operation is a 32-bit value and the destination is a register, then the value is sign-extended to a 64-bit value. This means that if you use the 32-bit versions of these instructions, the results will be sign-extended to 64-bit values.

The general notation for calculations is to provide the source operands first, and the destination operand last.

```
ADDQ    Ra, Rb, Rc  ; Rc = Ra + Rb
ADDQ    Ra, #b, Rc  ; Rc = Ra + b

ADDL    Ra, Rb, Rc  ; Rc = (int64_t)((int32_t)Ra + (int32_t)Rb)
ADDL    Ra, #b, Rc  ; Rc = (int64_t)((int32_t)Ra +           b)
```

The `ADD` instruction has four variants. The 64-bit versions add the two source values and puts the result in the destination *Rc* register. The 32-bit versions add the least significant 32-bit values in the source registers, calculates a 32-bit result, and then sign extends that result to a 64-bit value, putting the final result in the *Rc* register. You can add two registers, or you can add a register and a small constant in the range 0 to 255.

In the future, I'm going to write `x` to mean "`L` or `Q`", and `Rb/#b` to mean "a register (*Rb*) or a small constant in the range 0 to 255."

```
SUBx    Ra, Rb/#b, Rc  ; Rc = Ra - Rb
MULx    Ra, Rb/#b, Rc  ; Rc = Ra * Rb
UMULH   Ra, Rb/#b, Rc  ; Rc = (Ra *U Rb) >> 64
```

The `SUB` instructions perform subtraction, and the `MUL` instructions perform multiplication. The `UMULH` instruction performs a 64×64 unsigned multiplication, and stores the high 64 bits of the 128-bit intermediate result. (If you want the low 64 bits, then use the regular `MULQ` instruction.)

Note that there is no integer division operation. There are three common workarounds:

- Use a helper function.
- If dividing by a constant *n*, you may be able to use the `UMULH` instruction to multiply by $(2^{64} \div n)$ and then extract the high 64 bits (which means to divide by 2^{64}).
- Convert both values to floating point, perform a floating point division, and then convert the result back to an integer.

So hopefully you don't do a lot of integer division.

```
S4ADDx  Ra, Rb/#b, Rc  ; Rc = Ra * 4 + Rb/#b
S8ADDx  Ra, Rb/#b, Rc  ; Rc = Ra * 8 + Rb/#b

S4SUBx  Ra, Rb/#b, Rc  ; Rc = Ra * 4 - Rb/#b
S8SUBx  Ra, Rb/#b, Rc  ; Rc = Ra * 8 - Rb/#b
```

The scaled addition and subtraction instructions multiply *Ra* by 4 or 8 before adding or subtracting *Rb*/#b. These are commonly used to calculate effective addresses as part of an array indexing operation.

Next come the bit-twiddling instructions. Note that these instructions always operate on full 64-bit registers. (But if both inputs are in canonical form, then so too will the result.)

```
AND     Ra, Rb/#b, Rc  ; Rc = Ra &  Rb/#b
BIS     Ra, Rb/#b, Rc  ; Rc = Ra |  Rb/#b "bit set"
XOR     Ra, Rb/#b, Rc  ; Rc = Ra ^  Rb/#b
BIC     Ra, Rb/#b, Rc  ; Rc = Ra & ~Rb/#b "bit clear"
ORNOT   Ra, Rb/#b, Rc  ; Rc = Ra | ~Rb/#b
EQV     Ra, Rb/#b, Rc  ; Rc = Ra ^ ~Rb/#b "bit equivalence"
```

Officially, the `C` in `BIC` stands for "complement", but I find it easier to remember if I pretend that it stands for "clear", because it clears the bits in `Ra` as selected by `Rb/#b`. For example,

```
BIC     t0, #3, t2     ; clear bottom two bits of t0
```

This takes the value in *t0*, clears the bottom two bits (#3), and puts the result into *t2*.

The `EQV` and `ORNOT` instructions are not widely used, but I included them for completeness.

There are three bit-shifting instructions.

```
SLL     Ra, Rb/#b, Rc  ; Rc =           Ra << (Rb/#b % 64)
SRL     Ra, Rb/#b, Rc  ; Rc = (uint64_t)Ra >> (Rb/#b % 64)
SRA     Ra, Rb/#b, Rc  ; Rc = ( int64_t)Ra >> (Rb/#b % 64)
```

The right-shift has two variants, depending on whether you want the shifted value to be zero-filled (unsigned, or logical shift) or sign-filled (signed, or arithmetic shift). Note that there are no 32-bit versions of the bit shifting instructions. They always operate on the full 64-bit register.

There are some rarely-used computation instructions that I'm not going to go into, like "count number of leading zero bits" and all the multimedia instructions. There are also some other computation instructions that are closely related to other functions of the processor, so I'll defer those to the appropriate section. Next time, we'll look at memory access, including the computation instructions tailored to support memory operations.

Bonus chatter: There are a number of idioms that let you express other concepts in terms of the instructions above.

```
BIS     zero, zero, zero    ; NOP (writes to zero are ignored)
BIS     zero, zero, Rc      ; Set Rc to zero
ADDL    zero, #b, Rc        ; Set Rc to a small constant
SUBL    zero, #b, Rc        ; Set Rc to a small negative constant
BIS     Ra, Ra, Rc          ; Copy Ra to Rc
BIS     zero, Ra, Rc        ; Copy Ra to Rc
BIS     Ra, zero, Rc        ; Copy Ra to Rc
SUBx    zero, Ra, Rc        ; Rc = -Ra
ORNOTx  zero, Ra, Rc        ; Rc = ~Ra
ADDL    zero, Rb, Rc        ; Rc = (int64_t)(int32_t)Rb
```

Note that I gave three ways to copy one register to another. The first is the one recommended by DEC. The second is the one the Microsoft compiler generates. Windows NT requires that copying registers in function prologues and epilogues must be performed with one of the three given formats in order for the instruction to be unwindable.

I showed idioms for loading small positive and negative constants, but we'll see next time that there's something that works for medium-sized constants.

The last idiom is an important one because it forces a 32-bit value into canonical form. This is useful when there isn't a 32-bit version of the instruction you want, such as a shift instruction.

Raymond Chen

Follow