The Alpha AXP, part 16: What are the dire consequences of having 32-bit values in non-canonical form?

devblogs.microsoft.com/oldnewthing/20170829-00

August 29, 2017

Raymond Chen

On the Alpha AXP, 32-bit values are typically represented in so-called canonical form. But what happens if you use a non-canonical representation?

Well, it depends on what instruction consumes the non-canonical representation.

If the consuming instruction is an explicit 32-bit instruction, such as `ADDL` or `STL`, then the upper 32 bits are ignored, and the operation proceeds with the lower 32 bits. In that case, the non-canonical representation causes no harm. For example, consider this calculation:

```
; Calculate Rc = Ra + Rb + 0x1234 (32-bit result)
LDA     Rc, 0x1234(zero)    ; Rc = 0x00000000`00001234
ADDL    Rc, Rb, Rc          ; Rc = Rb + 0x1234
ADDL    Rc, Ra, Rc          ; Rc = Ra + Rb + 0x1234
```

If we are willing to use a non-canonical form temporarily, we could simplify this to

```
; Calculate Rc = Ra + Rb + 0x1234 (32-bit result)
LDA     Rc, 0x1234(Rb)      ; Rc = Rb + 0x1234 (64-bit intermediate)
ADDL    Rc, Ra, Rc          ; Rc = Ra + Rb + 0x1234 (32-bit result)
```

The `LDA` will put *Rc* into non-canonical 32-bit form if *Rb* is in the range `0x7FFFEDCC` to `0x7FFFFFFF` because the `LDA` instruction is 64-bit only, and the result would be in the range ``0x00000000`80000000`` through ``0x00000000`80001233``, which are non-canonical. But all is forgiven at the `ADDL` instruction, since it considers only the 32-bit portion of the addends (ignoring the non-canonical part) and generates a 32-bit result in canonical form.

On the other hand, if the instruction that consumes the non-canonical 32-bit value is a 64-bit instruction, then the non-canonical value will cause trouble.

Consider this simple function:

```
void f(int x)
{
    if (x == 0) DoSomething();
}
```

The Windows ABI for Alpha AXP requires that all 32-bit values be passed and returned in canonical form. You are welcome to use non-canonical values inside your function, but all communication with the outside world must use canonical form for 32-bit values.

This function might assemble to something like this:

```
    BEQ     a0, DoSomething ; tail call optimization
    RET     zero, (ra), 1   ; return without doing anything
```

The first instruction checks whether *x* is zero. If so, then it jumps directly to the `DoSomething` function, leaving the return address unchanged, so that when `DoSomething` returns, it returns to the caller of `f`. (This is a tail call optimization.)

If the value is not zero, then it returns to the caller.

There is no 32-bit version of the `BEQ` instruction; it always tests the full 64 bits.

If the value of *x* were not canonical, then the branch instruction could suffer false negatives: Even though the lower 32 bits are zero, there may be nonzero bits set in the upper half. That cause the `BEQ` to report "sorry, not zero" even though the 32-bit part of *a0* was zero.

There are a number of instructions which do not have a 32-bit version and which always operate on the full 64-bit register value. Another example:

```
void f(int x, int y)
{
    if (x < y) DoSomething();
}
```

This function might assemble to something like this:

```
    CMPLT   a0, a1, t0      ; t0 = 1 if a0 < a1
    BNE     t0, DoSomething ; tail call optimization
    RET     zero, (ra), 1   ; return without doing anything
```

In this version, the compiler performs a signed less-than operation and branches based on the result. The `CMPLT` instruction always operates on the full 64-bit register value; there is no 32-bit version. Consequently, passing a non-canonical value can result in the debugger reporting strange things like "Well, even though you passed $x = 1$ and $y = 2$, the less-than comparison returned false because *x* was passed in the non-canonical form of `0xFFFFFFFF`00000001`.

Using sign-extended values for canonical form for 32-bit values has the nice property that signed and unsigned comparisons of 32-bit values have the same results as signed and unsigned comparisons of their corresponding canonical forms.

If zero-extension had been used for canonical form, then unsigned comparisons would be preserved, but signed comparisons would not agree: The 32-bit signed comparison of `0x00000000` with `0xFFFFFFFF` would report that the first value is larger (0 > −1) but the 64-bit signed comparison ``0x00000000`00000000`` with ``0x00000000`FFFFFFFF`` of the corresponding zero-extended values would report that the second value is larger (0 < 4,294,967,295).

I'm pretty sure this was not a coincidence.

Bonus chatter: Non-canonical values introduce another case where uninitialized variables can result in strange behavior. Consider:

```
int f()
{
    int v;
    ... a bunch of code that somehow forgot to set v ...
    ... but in a complicated way that eluded code flow analysis ...
    return (v < 0) ? -1 : 0;
}
```

This might get compiled to the following:

```
    ; compiler chooses t0 to represent v
    ...
    SRA     t0, #32, v0     ; v0 = 0xFFFFFFFF`FFFFFFFF if t0 was negative
                            ; v0 = 0x00000000`00000000 if t0 was nonnegative
    RET     zero, (ra), 1   ; return the result
```

If the code forgets to assign a value to *v*, then it will have the value left over from whatever code ran earlier. Suppose that leftover value happened to be the non-canonical value ``0x12345678`12345678``. In that case, the result of the `SRA` would be ``0x00000000`12345678``, and the function `f` ends up returning some value that seems to be impossible from reading the code: According to the code, the function always returns either `-1` or `0`, yet sometimes we crash because it returned the crazy value `0x12345678`!

Raymond Chen

Follow