How to compare two packed bitfields without having to unpack each field

devblogs.microsoft.com/oldnewthing/20190301-00

March 1, 2019

Raymond Chen

Suppose you are packing multiple bitfields into a single integer. Let's say you have a 16-bit integer that you have packed three bitfields into:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
r					g						b				

Suppose you have two of these packed bitfields, *x* and *y*,

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xr					xg						xb				
yr					yg						yb				

and you want to know whether every field in *x* is greater than or equal the corresponding field in *y*. I.e., you want to determine whether $xr \geq yr$, $xg \geq yg$, and $xb \geq yb$.

One way would be to unpack the bitfields.

```
bool IsEveryComponentGreaterThanOrEqual(uint16_t x, uint16_t y)
{
 auto xr = x >> 11;
 auto yr = y >> 11;
 if (xr < yr) return false;

 auto xg = (x >> 5) & 0x3F;
 auto yg = (y >> 5) & 0x3F;
 if (xg < yg) return false;

 auto xb = x & 0x1F;
 auto yb = y & 0x1F;
 if (xb < yb) return false;

 return true;
}
```

There's an obvious optimization here, which is to avoid the extra shifting.

```
bool IsEveryComponentGreaterThanOrEqual(uint16_t x, uint16_t y)
{
 auto xr = x & 0xF100;
 auto yr = y & 0xF100;
 if (xr < yr) return false;

 auto xg = x & 0x07E0;
 auto yg = y & 0x07E0;
 if (xg < yg) return false;

 auto xb = x & 0x001F;
 auto yb = y & 0x001F;
 if (xb < yb) return false;

 return true;
}
```

But suppose this comparison is part of your program's inner loop, so you're hoping for something better.

Well, if you had planned ahead and inserted a zero padding bit at the front of each field:

18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	r					0	g						0	b				

then you could subtract the two values and see if any padding bit became set, which indicates that an underflow occurred somewhere to the right.

```
bool IsEveryComponentGreaterThanOrEqual(uint32_t x, uint32_t y)
{
 auto m = (x - y) & ((1 << 18) | (1 << 12) | (1 << 5));
 return m == 0;
}
```

However, this forces you to reserve padding bits, and it seems silly to have padding bits all over your data just for this purpose. I mean, those are bits that could've been doing something useful!

In our example, those three extra bits forced us to use a larger integral type, which means our memory usage doubled.

Can you do it without inserting padding bits?

Indeed you can, thanks to a trick from emulator master Darek Mihocka: The carry-out vector.

You can read the paper or take the easier route and read the presentation.

In this case, we want the subtraction carry-out vector (which is really the borrow vector). The formula is right here in the Bochs emulator source code.

```
#define SUB_COUT_VEC(op1, op2, result) \
  (((~(op1)) & (op2)) | ((~((op1) ^ (op2))) & (result)))
```

In the subtraction carry-out vector, a bit is set if the subtraction resulted in a borrow at that position. We then check whether there was a borrow at the corresponding high bits 4, 10, or 15.

Here we go:

```
bool IsEveryComponentGreaterThanOrEqual(uint16_t x, uint16_t y)
{
 auto c = ((~x & y) | (~(x ^ y) & (x - y));
 c &= 0x8410;
 return c == 0;
}
```

Slide 13 of the presentation linked above shows how this technique can be used to implement saturating bitfield arithmetic in general-purpose registers. Who needs SIMD registers!

The carry-out vector is truly magical.

Bonus reading: How Bochs Works Under the Hood. The "Lazy flags handling" section has a useful diagram.

Raymond Chen

Follow