Unimal 2.0

Application note 3

Computing a remainder modulo a constant

Documentation revision 2.00

Techniques:
Develop an algorithm, then implement it at compile time

MacroExpressions
http://www.macroexpressions.com

Table of contents

Calculating a remainder on a small microcontroller: a problem

This note is for mostly for illustration purposes: A practical value for larger number ranges is questionable due to the size of the table. However, the technique may be useful.

Assume that we have to compute the remainder of an unsigned integral variable `x` divided by `N`, an unsigned constant. A natural C implementation is simply to write:

```
x%N
```

A typical application: Assume that we have an array of software timer counters `TimerCounter[]`, which are incremented and reset according to certain rules. Assume, further, that we need to do certain processing on every `N`-th tick of this timer, where `N` is a constant. A "natural" C implementation would look like this:

```
if(TimerCounter[y]%N == 0) {
     do_processing(y);
}
```

The problem is that if we have a fairly small microcontroller at hand (including some RISC machines), there might be no division instruction, and the remainder operator (%) can become costly, especially if this calculation occurs often.

To remedy the situation, we can trade space for time as described in the following section.

An approach to optimization

Yes, we are going to show how Unimal can help in this situation.

However, Unimal will not invent an algorithm for you; it will only help implement your algorithm at compile time. So, we have to come up with an algorithm and then see if and how it can be implemented in Unimal.

So the first order of business is to develop an algorithm for a "modulo constant" operation.

Let's assume that we know a number y, which is a multiple of N and differs from x by less than N. Then the signed remainder is x-y, and if we need the non-negative remainder, we add the N to x-y if the latter is negative.

To find a y, let's simply use a table of the multiples of N for the whole range of possible values of x.
We are going to create this table in such a way that finding the right entry is computationally very easy.

Let M be a number not greater than N, and let the array MultN_M be such that its n-th element (0-based), MultN_M[n], is the greatest multiple of N less than (n+1)*M. In other words (or, rather, symbols),

```
MultN_M[n]=floor(((n+1)*M-1)/N)*N,
```

where floor(x) is the largest integer not exceeding x.

Then, as long as $n*M \le x < (n+1)*M$, MultN_M[n] is the number y we are looking for.

(Indeed, since $(n+1)*M-N \le MultN\_M[n] < (n+1)*M$, or

$$-(n+1)*M < -MultN\_M[n] \le -(n+1)*M+N$$

and

$$n*M \le x < (n+1)*M,$$

it follows that

$$-N \le -M = -(n+1)*M+n*M < x-MultN\_M[n] < (n+1)*M+N-(n+1)*M = N.)$$

It doesn't look like we made our job easier yet, because we need to find the n such that $n*M \le x < (n+1)*M$, which means that we need to divide x by M. I.e., we are again at square one. Notice, however, that we can choose the M to our liking. If we take it to be a power of 2, then division by M reduces to the right shift by the number of bits equal to the exponent of the power of 2. This is easily handled by most processors.

For the example, consider finding remainders from division over N=20. Let's assume that x is an 8-bit variable, so its range is from 0 to 255. The greatest power of 2 not exceeding N is M=16, and thus the exponent is 4 ($2^4=16$). The maximum quotient is therefore 255/16+1=16, so we need to tabulate the 16 multiples of 20:

```
static const unsigned char Mult20_16 = {
   0, 20, 40,60,60,80,100,120,140,140,160,180,200,220,220,240,
};
```

Just to make sure it's clear, Mult20_16[4]=60 because (4+1)*16=80 and the greatest multiple of 20 less than 80 is 60.

For a variable *x*, the closest multiple of 20 is found at offset `x/16`, or `x>>4` (right shift 4 bits), and the (signed) remainder is therefore `x – Mult20_16[x>>4].` If the remainder is negative, we can add 20 to it to get an unsigned remainder. Putting this all together, we come up with the following simple function:

```
int remainder20(unsigned char x)
{
  static const unsigned char Mult20_16[] = {
    0, 20, 40,60,60,80,100,120,140,140,160,180,200,220,220,240,
  };
  int remainder;
  remainder = (signed)x - Mult20_16[x>>4];
  if(remainder<0) remainder += 20;
  return remainder;
}
```

This is something any machine handles easily. Moreover, for efficiency, you may use inlining or convert this to a C macro.

This example shows that even though we could choose any power of 2 as the `M` (not to exceed `N`), the size of the table of quotients is `256/M`, so the greater the `M`, the better.

Solution analysis and a maintainable implementation

Now that we know what we want to get in the end, we want to automate the process so as to eliminate project maintenance efforts. That is, when we know the divisor N at compile time, we want a remainder<N> function created automatically.

To do so, we will make Unimal repeat the steps we accomplished manually in the example.

First, given N, we want to find M, the largest power of 2 not exceeding N and the exponent of this power, such that 1<<exponent == M.

Here is a simple macro to accomplish that:

```
#MP Macro Log2 ;(divisor)
#MP For n = 0, 32 ;any large enough number
#MP  M = 1<<n
#MP  If 2*M>#1#
#MP    Set exponent=n
#MP    n=2000 ;break the loop
#MP  Endif
#MP Endfor
#MP Endm
```

(See Samples\AppNotes\3\remainder.u)

Note that we needed to manipulate the loop counter (n) to break out of the loop.

The second thing worth wrapping in a macro is populating the table of the exact multiples. This is a typical example of a tabulated function

```
((n*M-1)/N)*N
```

Here is the macro that takes the array size, N and M as arguments. It simply computes and renders the value in a loop.

```
#MP Macro TableInit ;(size, N, M)
#MP For n = 1, #1#
#MP  Set Temp = ((n*#3#-1)/#2#)*#2#
  #mp%uTemp,
#MP Endfor
#MP Endm
```

Finally, here is a function generator (Unimal macro parameters in the target language interface are italicized for clarity):

```
#MP Set N=20 ;constant divisor
/* generate the remainder function (most importantly, the table)
*/
#MP Expand Log2(N)
int remainder#mp%uN(unsigned char x)
{
    static const unsigned char bucket#mp%uN[] = {
```

```
    #MP Expand TableInit(256/M, N, M)
    };
  int remainder;
  remainder = (signed)x - bucket#mp%uN[x>>#mp%uexponent];
  if(remainder<0) remainder += #mp%uN;
  return remainder;
}
```

If we run

```
Unimal remainder.u
```

we'll get the output we expected:

```
/* generate the remainder function (most importantly, the table)
*/
int remainder20(unsigned char x)
{
   static const unsigned char bucket20[] = {
     0,
     20,
     40,
     60,
     60,
     80,
     100,
     120,
     140,
     140,
     160,
     180,
     200,
     220,
     220,
     240,
  };
  int remainder;
  remainder = (signed)x - bucket20[x>>4];
  if(remainder<0) remainder += 20;
  return remainder;
}
```