

# Implementation Techniques: Part 5

- ▶ One can think of **computer arithmetic** at three different levels:
  - ▶ **Low-level** bit manipulation managed by hardware; for example, the full and half adder circuits in an ALU.
  - ▶ **Mid-level** word manipulation managed by programmers; for example, a standard add expression in C.
  - ▶ **High-level** mathematical objects managed by libraries; for example, addition of data structures such as matrices.
  - ▶ These levels are stacked on top of each other like a pyramid, each one depends on efficiency in the lower levels.
- ▶ The rough goal of this lecture is to demonstrate that by considering and using low-level arithmetic, you can improve **implementation quality**.
- ▶ This is a broad topic, we concentrate on a few specific areas:
  - ▶ **Bit manipulation** and use in constructing bit-sets.
  - ▶ **Constant multiply**, something like  $a = b * 9$ .
  - ▶ **Constant division**, something like  $a = b / 9$ .

# Implementation Techniques: Part 5

- ▶ In this lecture we will mainly consider the construction of efficient mid-level operations by using low-level operations more directly.
- ▶ Some of this might seem obvious, you might have seen it before:
  - ▶ History has demonstrated that the topic isn't well understood ...
  - ▶ ... even if you do, this should act as a useful recap !
- ▶ To fix notation, we make the following definitions:
  - ▶ A **byte** is a group of 8-bits.
  - ▶ A **word** is a group of  $w$ -bits; we try be general and talk about  $w$ -bit words because different processors have different natural word sizes ...
  - ▶ Keep in mind that if a processor is described as 32-bit, we are roughly saying that  $w = 32$ .
- ▶ We deal only with **unsigned** values to make things easier.

# Bit-sets (1)

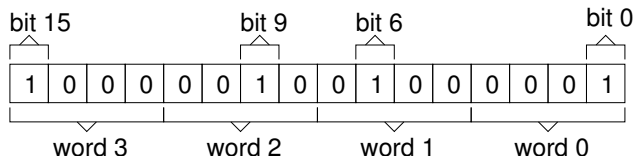
- ▶ Hopefully everyone knows what a **set** is: an unordered collection of elements where each element can only occur once.
- ▶ One can perform some standard operations on such an object:
  - ▶ Control set **membership**, for example check if an element is in the set or add an element to the set.
  - ▶ Check the **size** of the set, that is count how many elements are in the set.
  - ▶ Perform **arithmetic** on sets, for example calculate set intersection and union.
- ▶ Java has `java.util.HashSet` and `java.util.TreeSet`, for example, that capture meaning of a set **and** the content:
  - ▶ The data structure holds references to elements in the set.
  - ▶ Any set operations will manipulate the underlying data structure to enforce the correct semantics.
- ▶ Consider implementing a set using a **linked-list** as the underlying data structure:
  - ▶ Adding an element means appending it to end of list ...
  - ▶ ... removing an element means deleting it from the list.

## Bit-sets (2)

- ▶ But what if we don't need to hold the elements themselves; maybe we have a fixed universe of elements and just need to represent their membership in various sets.
- ▶ Instead of holding the actual elements, a **bit-set** refers to them using integer identifiers; think of each element being hashed onto an integer.
- ▶ A bit-set captures **only** the meaning of the set, not the content ...
- ▶ ... but the trade-off is that the actual set operations are faster and the data structure is very compact.
  - ▶ We hold the actual elements in some other data structure, for example an array or a list.
  - ▶ Using the bit-set we can perform (limited) operations such as: “check if object 1 is in the set” or “put object 8 into the set”.
- ▶ Java has a concrete class to implement bit-sets (but which doesn't use the normal set interface) called `java.util.BitSet`.

## Bit-sets (3)

- ▶ A bit-set is simply an **array of words**; if there are  $n$  words in the array, there are  $n \cdot w$  bits in total.
- ▶ Consider this set of 16 values, where  $w = 4$  to make things easier:



- ▶ If bit  $i$  is 1 then element  $i$  is a member of the set; if bit  $i$  is 0 then it isn't.

## Bit-sets (4) – Membership

- ▶ Our first goal is membership testing, i.e. check if element  $i$  is in the set: we need to test if bit  $i$  of bit-set  $A$  is 1.

```
if( A[i/w] & ( 1 << (i%w) ) )  
{  
    ... bit i is set ...  
}
```

- ▶ Note that since division and remainder by  $w$  are **cheap** since  $w$  is a power-of-two. For example, if  $w = 4$  we have that:
  - ▶  $i/w \rightarrow i \gg 2$ .
  - ▶  $i\%w \rightarrow i \& 3$ .
- ▶ Many processors have dedicated instructions to test if bit  $i$  of some word  $x$  is set, for example 80x86 has `bt` or **bit test** ...
- ▶ ... this reduces the cost **if** the compiler can use it; often you need to do this by hand using assembly language !

## Bit-sets (5) – Membership

- ▶ Next, we'd like to be able to add or remove elements to and from the set: we need to force bit  $i$  of bit-set  $A$  to be 1 or 0:

```
A[i/w] |= ( 1 << (i%w) ); // force bit to 1
A[i/w] &= ( ~( 1 << (i%w) ) ); // force bit to 0
```

- ▶ The mask  $( 1 << (i\%w) )$  has 1 in the right place, **ORing** it forces the bit to 1; the mask  $\sim( 1 << (i\%w) )$  is all 1 except for a 0 in the right place, **ANDing** it forces the bit to 0.
  - ▶ Consider bit  $i = 1$  for example,  $A[i/w]$  is  $A[0]$ ; imagine this word is  $1100_2$  to start with.
  - ▶ Then  $( 1 << (i\%w) )$  is  $0010_2$  so  $1100_2 \vee 0010_2 = 1110_2$  and we've set the bit to 1;  $\sim( 1 << (i\%w) )$  is  $1101_2$  so  $1110_2 \wedge 1101_2 = 1100_2$  and we've set the bit back to 0.
- ▶ Again, many processors have dedicated instructions to do this, for example 80x86 has `bts` and `btr` or **bit test-and-set** and **bit test-and-reset**.

## Bit-sets (6) – Size

- ▶ Next, we'd like to count the number of elements in the set: we need to count the number of bits set to 1 in each word of  $A$  ...
- ▶ ... consider counting the bits only in word  $j$ , to do the whole set we just loop through all the words.

```
t = A[j];  
  
for( c = 0, k = 0; k < w; k++ )  
{  
    if( ( t >> k ) & 1 )  
        c = c + 1;  
}
```

- ▶ This approach is quite **wasteful** when implemented; the overhead of the loop is large in comparison to the content.
- ▶ A more efficient method is to consider **population count**:
  - ▶ The operation counts the number of bits set in a word; this is often called the **Hamming weight** of a word.
  - ▶ Again, this is an actual instruction on some processors.

## Bit-sets (7) – Size

- ▶ We can use a **divide-and-conquer** approach to population count.
- ▶ Consider the case where  $w = 16$  to make things easier:
  - ▶ We split the problem into 2-bit, 4-bit and then 8-bit fields.
  - ▶ On each step, we add the number of bits from previous steps.
- ▶ Here is an example for  $x = 1011110001100001_2$ :

1	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1
0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	1
0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1
0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

- ▶ The question then is to implement these steps efficiently.

## Bit-sets (8) – Size

- ▶ So to start with  $x = 1011110001100001_2$  and for the next step we do:  
 $x = (x \& 0x5555) + ((x \gg 1) \& 0x5555);$
- ▶ This works because:  
 $t0 = (x) \& 0x5555 \rightarrow 0001010001000001_2$   
 $t1 = (x \gg 1) \& 0x5555 \rightarrow 0101010000010000_2$   
 $t0 + t1 \rightarrow 0110100001010001_2$
- ▶ For the next steps we do:  
 $x = (x \& 0x3333) + ((x \gg 2) \& 0x3333);$   
 $x = (x \& 0x0F0F) + ((x \gg 4) \& 0x0F0F);$   
 $x = (x \& 0x00FF) + ((x \gg 8) \& 0x00FF);$
- ▶ So, in general we **mask** the bits we want, **shift** them by the right offset and then add the counts together.
- ▶ Notice that the code is straight-line: it has no **branches** and no **loop overhead**.

## Bit-sets (9) – Intersection and Union

- ▶ Finally, we'd like to calculate the intersection or union of two sets.
- ▶ For example build a bit-set  $C$  so that it contains elements that are in both  $A$  and  $B$  (i.e. intersection) ...
- ▶ ... or build a bit-set  $D$  so that it contains elements that are in either  $B$  or  $C$  (i.e. union).

```
for( i = 0; i < n; i++ )  
{  
    C[ i ] = A[ i ] & B[ i ];  
}
```

```
for( i = 0; i < n; i++ )  
{  
    D[ i ] = A[ i ] | B[ i ];  
}
```

- ▶ This is intuitive since logical AND is the intersection operation; logical OR is the union operation.

## Constant Multiply (1)

- ▶ You may have already seen the concept of left-shift used for the special case of multiplication by a power-of-two:
  - ▶ Say we want to multiply an integer  $a$  by 2; we can just shift  $a$  left one bit.
  - ▶ In C this looks like  $a = a \ll 1$ ;
  - ▶ More generally, multiplying  $a$  by  $2^k$  means left-shifting  $a$  by  $k$  bits.
- ▶ This is obvious if you look at the binary expansion of  $a$

$$a = \sum_{i=0}^{i < n} a_i \cdot 2^i$$

since by shifting left one bit we are moving bit  $i$  to bit  $i + 1$ , i.e.

$$a' = \sum_{i=0}^{i < n} a_i \cdot 2^{i+1}$$

which clearly means that  $a' = 2 \cdot a$ .

- ▶ Shifts are generally less expensive to compute and are often serviced by a different functional unit; but what about **non-power-of-two** ?

## Constant Multiply (2)

- ▶ We have already seen the idea of **addition-chains** in exponentiation; more obviously, they can be used to perform efficient multiplication.
  - ▶ Make an addition chain that adds **powers-of-two** of the input.
  - ▶ Generating powers-of-two is cheap, we just use shifts, so the whole multiplication is cheap.
- ▶ For example, to perform the operation  $a \leftarrow b \cdot 13$  in C we could replace the statement `a = b * 13;` with:

```
t0 = b;  
t1 = b << 2;  
t2 = b << 3;  
a  = t0 + t1 + t2;
```
- ▶ For **small** or **low-weight** constants, this method will be faster than a general multiply:
  - ▶ It doesn't **seem** much of an improvement, however if the multiply is used millions of times, we improve noticeably ...
  - ▶ ... for example, in a tight loop the difference might be apparent

# Constant Divide (1)

- ▶ We can use a similar approach to division by constants although things now get more complicated ... actually, a lot more complicated !
- ▶ However, division is even more expensive than multiplication so the rewards are greater if we succeed.
- ▶ The general approach is this, say we want to compute  $a \leftarrow \lfloor n/d \rfloor$ :
  - ▶ We estimate  $1/d$ , the **reciprocal** of  $d$ .
  - ▶ Then we multiply the reciprocal with  $n$  to get the result.
  - ▶ The trick is to perform this multiplication efficiently given the known value of  $1/d$ .
- ▶ Consider the case where we want to compute  $a \leftarrow \lfloor n/3 \rfloor$ .
  - ▶ In binary, we find that  $1/3 \simeq 0.0101\dots01_2$ .
  - ▶ Or more simply  $1/3 \simeq 1/2^2 + 1/2^4 + 1/2^6 + \dots + 1/2^{30}$ .
  - ▶ Hence the division  $n/3 \simeq n/2^2 + n/2^4 + n/2^6 + \dots + n/2^{30}$ .
- ▶ This looks **exceptionally** unpleasant, but actually it is okay since we already know how to deal with powers-of-two efficiently !
  - ▶ We divide by the powers-of-two using right shifts of  $n$ .
  - ▶ Add the shifted values together to get the final result.

## Constant Divide (2)

- ▶ In fact, we can do even better than that; using C we can write down our formula very efficiently using just 9 operations:

```
q =      ( n >> 2 ) +  
        ( n >> 4 ) ;  
q = q + ( q >> 4 ) ;  
q = q + ( q >> 8 ) ;  
q = q + ( q >> 16 ) ;
```

- ▶ However, we have a problem of **accuracy** in our calculation:
  - ▶ We are limited to 32-bit operations where as the reciprocal is infinitely long ...
  - ▶ ... we are **throwing away** some of the reciprocal with each right shift we perform !
- ▶ It turns out the maximum error for our short method is 5; this was found empirically, I'm not sure how to prove it.

## Constant Divide (3)

- ▶ We therefore term  $q$  the **estimated** quotient: it is always smaller than the actual quotient, basically because we are throwing away terms than would otherwise contribute.
- ▶ To **compensate** for our earlier errors, we first calculate the remainder  $r \leftarrow n - q \cdot 3$ ; since the error is at most 5, the remainder is at most  $5 \cdot 3 = 15$ .
- ▶ So to correct the error we need to add any **erroneous** part of the remainder back to the quotient:
  - ▶ That is,  $q \leftarrow q + (r/3)$ , where  $0 \leq r \leq 15$ .
- ▶ It looks like we are back to the problem we started with !
  - ▶ However, the restricted range of  $r$  saves us.
  - ▶ Again approximating, we multiply  $r$  by  $1/3 \simeq 11/32$ .
  - ▶ We select this approximation over since division by 32 is easy.
- ▶ At long last, the correction we want is  $q \leftarrow q + ((r \cdot 11)/32)$ .

## Constant Divide (4)

- ▶ In C, the whole operation looks like this; note we could go further by optimising the multiplication and division by constants !

```
unsigned int div3( unsigned int n )
{
    unsigned int q;
    unsigned int r;

    q =      ( n >> 2 ) +
           ( n >> 4 ) ;
    q = q + ( q >> 4 ) ;
    q = q + ( q >> 8 ) ;
    q = q + ( q >> 16 ) ;

    r = n - q * 3;

    return q + ( ( r * 11 ) / 32 );
}
```

## Constant Divide (5)

- ▶ As an example of this monstrosity in action, consider the case where  $n = 3267 = 110011001100_2$

$$\begin{array}{r} 001100110011_2 \quad n \gg 2 \\ 000011001100_2 \quad n \gg 4 \\ \hline 001111111111_2 \quad q \\ 000000111111_2 \quad q \gg 4 \\ \hline 010000111110_2 \quad q \\ 000000000100_2 \quad q \gg 8 \\ \hline 010001000010_2 \quad q \\ 000000000000_2 \quad q \gg 16 \\ \hline 010001000010_2 \quad q \end{array}$$

- ▶ So after the initial part have that:
  - ▶ ... the estimated quotient  $q = 010001000010_2 = 1090$ .
  - ▶ ... the remainder  $r = n - q \cdot 3 = 3276 - (1090 \cdot 3) = 6$ .
- ▶ Clearly  $r/3 = 2$  so the real quotient is  $q = q + (r/3) = 1092$  would be right if we can cope with a division by three with  $0 \leq r \leq 15$ .
- ▶ Our correction  $\lfloor (6 \cdot 11)/32 \rfloor = 2$  is also right; we are done !

# Conclusions

- ▶ Some of this probably seems way off the radar in terms of what you want to do in typical implementation tasks ...
- ▶ ... but good programmers solve hard problems: implementing some algorithm on a processor without a division instruction **needs** well engineered software !
- ▶ The **bottom line** is this:
  - ▶ This whole exercise is to illustrate a single, simple concept.
  - ▶ Thinking in terms of **bits** rather than **words** (i.e. at a low-level rather than a high-level) can improve performance.
  - ▶ It can pay to map a problem onto the capabilities of the processor rather than using an intuitive method.
- ▶ This sort of approach is sometimes called **bit-twiddling**:
  - ▶ Playing with the value of individual bits in order to achieve a higher-level operation more efficiently; essentially bypassing the overhead of niceties introduced by high-level languages !
  - ▶ Knowing where to **stop** is important; large changes for small gain are not a good idea, especially as maintainability often suffers.

## Further Reading

- ▶ H.S. Warren Jr. Hackers Delight. Addison-Wesley, 2003. ISBN: 0-201-91465-4.