EL-GY 6483

Aditi Shukla

---

1. Using some test cases, match these bit operations to their associated function:

1. x & 1	a) Return x without trailing 1s (e.g. 11011111 be- comes 11000000)
2. x & (1 « n)	b) Unset the n_{th} bit
3. x & ~(1 « n)	c) Return true if n_{th} bit is set
4. (x ˆ y) < 0	d) Return the minimum of x and y
5. y ˆ ((x ˆ y) & -(x < y))	e) Return true if x and y have opposite signs
6. x & (x – 1)	f) Return true if x is odd, false if x is even
7. x & (x + 1)	g) Return 0 if x is a power of 2 for $x > 0$

Solution:

I found out the answers to the above questions using c programming.

1. f

2. c

3. b

4. e

5. d

6. g

7. a

2. The following C “optimizations” are said to improve the performance of embedded systems. In reality, some of them are useless or even counterproductive on certain architectures. For each of the “optimiza- tions” given,

   - Find out why it optimizes performance on some architectures
   - Find out if there are any targets on which it does not improve performance, or decreases performance
   - On the architectures on which it improves performance, how great is the improvement? (e.g., one instruction overall, one instruction per iteration of a loop, etc.) Is the improvement significant or trivial?

   Here are the “optimizations”:

   (a) Count down to zero, not up to N, in for() loops

   (b) Avoid the % operation

   (c) Use an 8-bit unsigned char whenever you have a value that you know won’t go beyond 0-255 (e.g., some loop index variables)

Solution:

a. **i)** Some modern RISC machines, like PPC or SPARC, which reserve a whole register to be always zero, it is cheaper to compare something with zero than to compare with anything else. Hence,

- We might save a register.
- We might get a compare instruction with a smaller binary encoding.
- If a previous instruction happens to set a flag (likely only on x86 family machines), we might not even need an explicit compare instruction.

The forward loop of i from 0 to N will be translated in most processors to machine code as:

- Load i
- Compare and jump if Less than or Equal zero

But the reverse loop of i from N to 0 needs to load N from Memory each time:

- Load i
- Load N
- Sub i and N
- Compare and jump if Less than or Equal zero

So it is not because of counting down or up but because of how our code will be translated into machine code.

ii) It does not decrease performance. In some hardware, with the incrementing loop we have to test i<N each time round the loop. For the decrementing version, the zero flag may automatically tell us if i>=0. This saves a test per time round the loop. However, on modern pipelined processor hardware, this is certainly irrelevant as there isn't a simple one-one mapping from instructions to clock cycles.

What matters much more than whether we're increasing or decreasing our counter is whether we're going up memory or down memory. Most caches are optimized for going up memory, not down memory. Since memory access time is the bottleneck that most programs today face, this means that performance may certainly decrease even if we are going down the loop.

iii) Counting down to zero will have one instruction less per iteration of a loop. But, it is very insignificant.

b) i) & ii) C = A % B is equivalent to C = A – B * (A / B). In other words the modulus operator is functionally equivalent to three operations. As a result, the code that uses the modulus operator can take a long time to execute. This is why avoiding the % operator optimizes the performance of embedded systems. Some processors won’t have modulus instruction at hardware level. In such case the compilers will insert stubs (predefined functions) to perform modulus. It reduces performance.

iii) If we avoid using the modulus operation, this will reduce our runtime greatly. Binary division is literally long division, so it takes many more steps than comparable arithmetic operators. This is why, who've worked on embedded systems will remember implementing division through bit-shifting (often in assembly) as an optimization. However, modern CPUs use specialized division circuits with lookup tables and nowadays they're so good that we'll gain nothing in speed by using bit-shifting. By doing modulo to powers of two we can replace with bitwise 'and'.

ie

x % 256

is same as

x & 255

But we might not have that luxury of always having 16 or 32 etc items.

C) i) In C, the unsigned char data type is the only data type that has all the following three properties simultaneously:

- it has no padding bits, that is where all storage bits contribute to the value of the data.
- no bitwise operation starting from a value of that type, when converted back into that type, can produce overflow, trap representations or undefined behavior.
- it may alias other data types without violating the "aliasing rules", that is that access to the same data through a pointer that is typed differently will be guaranteed to see all modifications.

The range of unsigned char is 0 to 255 and when the value of variable in loop will cross over 255, value will be 0 and again same process will happen.

ii) It does not decrease performance.

iii) For a signed integer bit, one bit is used to store the sign and the rest are used to store the data.
For a unsigned all the bits are used to store the data. So we will save on memory.

3. Refer to the JPL Institutional Coding Standard for the C Programming Language (http://lars- lab.jpl.nasa.gov/JPL_Coding_Standard_ext.pdf). This standard describes their rules for mission critical flight software written in the C programming language. (The NASA Jet Propulsion Laboratory was responsible for the Mars Curiosity rover.)

   (a) Why is recursion not permitted in mission critical flight software?

   (b) Why is dynamic memory allocation disallowed after task initialization in mission critical flight software?

Solution:

a) The presence of statically verifiable loop bounds and the absence of recursion prevent runaway code, and help to secure predictable performance for all tasks. The absence of recursion also simplifies the task of deriving reliable bounds on stack use. The two rules combined secure a strictly acyclic function call graph and control-flow structure, which in turn enhances the capabilities for static checking tools to catch a broad range of coding defects.

b) There shall be no use of dynamic memory allocation after task initialization. This rule disallows the use of malloc(), sbrk(), alloca(), and similar routines, after task initialization. This rule is common for safety and mission critical software and appears in most coding guidelines. The reason is simple: memory allocators and garbage collectors often have unpredictable behavior that can significantly impact performance. A notable class of coding errors stems from mishandling memory allocation and free routines: forgetting to free memory or continuing to use memory after it was freed, attempting to allocate more memory than physically available, overstepping boundaries on allocated memory, using stray pointers into dynamically allocated memory, etc. Forcing all applications to live within a fixed, pre-allocated, area of memory can eliminate many of these problems and make it simpler to verify safe memory use.

4. Fill in the blanks with the word “signed” or “unsigned”:

(a) In **signed** arithmetic, if the overflow flag (V in CPSR) is set on an operation, the result is wrong.

(b) In **unsigned** arithmetic, the overflow flag (V in CPSR) does not indicate anything mean- ingful about the result of the operation.

(c) In **unsigned** arithmetic, if the carry flag (C in CPSR) is set on an operation, the result is wrong.

(d) In **signed** arithmetic, the carry flag (C in CPSR) does not indicate anything meaningful about the result of the operation.

Solution:

a) Signed

b) Unsigned

c) Unsigned

d) Signed

5. Describe the status of the N, Z, C, and V flags of the CPSR after each of the following:

(a)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
add     r0, r1, r2
```
(b)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
cmn     r1, r2
```
(c)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
adds    r0, r1, r2
```
(d)
```
ldr     r1, =0xffffffff
ldr     r2, =0x00000001
addeq   r0, r1, r2
```
(e)
```
ldr     r1, =0x7fffffff
ldr     r2, =0x7fffffff
adds    r0, r1, r2
```

Solution:

a) All the flags remain unchanged.

b) N = unchanged, Z = 1, C = 1, V = 0

c) N = unchanged, Z = 1, C = 1, V = 0

d) N = C = Z = V = 0 as the “equal” condition did not become active.

e) N = 1, C = Z = unchanged, V = 1

6. The following C code implements the Euclid algorithm for calculating the greatest common divisor:

```c
int gcd(int a, int b)
{
    while  (a  != b)
      {
        if  (a  > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

Here is an equivalent ARM assembly routine that only uses conditional execution on the branch instruc- tions:

```
gcd
        CMP        r1,  r2
        BEQ        end
        BLT        lessthan
        SUB        r1, r1, r2
        B          gcd
lessthan
        SUB        r2, r2, r1

        B          gcd
end
        ...
```

And here is an equivalent ARM assembly routine that uses full conditional execution :

```
gcd     CMP       r1, r2
        SUBGT     r1, r1, r2
        SUBLT     r2, r2, r1
        BNE       gcd
```

Assume a is 54 and is loaded into r1, b is 24 and is loaded into r2.

(a) Run through the C algorithm until its completion to find the greatest common divisor.

Solution:

	a	**b**	**a!=b**	**a=a-b**	**b=b-a**	
First Iteration	54	24	true	30	-	
Second Iteration	30	24	true	6	-	
Third Iteration	6	24	true	-	18	
Fourth Iteration	6	18	true	-	12	
Fifth Iteration		6	12	true	-	6

Now a = b = 6, hence the condition (a != b) becomes false, therefore a=6 is returned.

GCD = 6

(b) Run through the ARM assembly version without full conditional execution.

Solution:

Begin with: r1=54 and r2=24,

Loop 1

•CMP r1, r2 r1 is greater

• SUB r1, r1, r2 r1 = r1-r2 = 54-24 = 30

• Branch to gcd

Loop 2

• CMP r1, r2 r1 is greater

• SUB r1, r1, r2 r1= r1-r2 =30-24 = 6

• Branch to gcd

Loop 3

• CMP r1, r2 r2 is greater

• BLT less than branch to less than

• SUB r2, r2, r1 r2=r2-r1 = 24-6 =18

• Branch to gcd

Loop 4

• CMP r1, r2 r2 is greater

• BLT less than branch to less than

• SUB r2, r2, r1 r2=r2-r1 = 18-6 =12

• Branch to gcd

Loop 5

- CMP r1, r2 r2 is greater
- BLT less than branch to less than
- SUB r2, r2, r1 r2=r2-r1 = 12-6 =6
- Branch to gcd
- CMP r1, r2 Both r1, r2 become equal
- BEQ end Branch to end
- end

GCD is 6

(c) Run through the ARM assembly version with full conditional execution.

Solution:

Initially r1 = 54 and r2 =24

- CMP r1, r2 r1 is greater than r2
- SUBGT r1, r1, r2 r1 = r1-r2 = 54-24 = 30
- SUBLT will not execute because r1 >r2
- BNE gcd branch back to gcd since r1 != r2
- CMP r1, r2 r1 is greater than r2
- SUBGT r1, r1, r2 r1 = r1-r2 = 30-24 = 6
- SUBLT r2, r2, r1 r2 = r2-r1 = 24-6 = 18
- BNE gcd branch back to gcd since r1 != r2
- CMP r1,r2 r2 is greater than r1
- SUBGT will not execute because r2 > r1
- SUBLT r2, r2, r1 r2 = r2-r1 = 18-6=12
- BNE gcd branch back to gcd since r1 != r2
- CMP r1, r2 r2 is greater than r1

• SUBGT will not execute because r2 > r1

• SUBLT r2, r2, r1 r2 = r2-r1 = 12-6=6

• BNE gcd will not execute since r1=r2

The GCD is 6.

(d) Refer to the ARM Cortex-M4 Technical Reference Manual (available online) to find out the timing of each instruction. How many cycles does the first ARM routine take? How many cycles does the second ARM routine take?

Solution:

From the ARM Technical Reference Manual, we find that:

• CMP, SUB take 1 cycle (as do SUBLT, SUBGT)

• BEQ, BLT, BNE take 1 cycle if not executed and 1 + P cycles if executed

• B takes 1 + P cycles

Where, P is the number of cycles required for a pipeline refill. This ranges from 1 to 3.

The first routine takes

• First, second iterations: 5 + P cycles each

• Third, fourth, fifth iterations: 5 + 2P cycles each

• Sixth iteration: 2 + P cycles

for a total of 27 + 9P cycles.

The second routine takes

• First through fifth iterations: 4 + P cycles each

• Sixth iteration: 4 cycles

for a total of 24 + 5P cycles (3 + 4P cycles fewer than the previous routine).