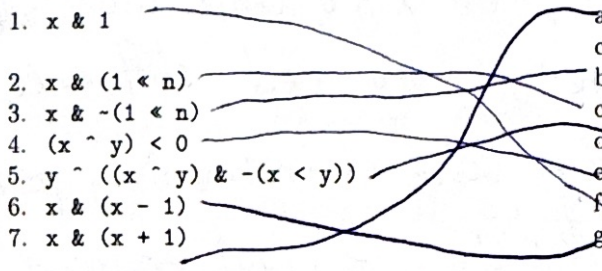
Hongdeo Meng

EL-GY 6483

---

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



Solution:

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 "optimizations" 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) Count down to zero, not up to N

- Why it helps: Some CPUs (e.g., ARM) have "decrement and test zero" instructions, so loop checks are cheaper.
- Where it doesn't help: Modern compilers often optimize both ways; on x86 and most 32/64-bit CPUs, there's no real difference.
- Improvement: Usually save 1 instruction per loop on small MCUs; trival on some modern CPUs.

(b) Avoid the % operation

- Why it helps: % use division, which is very slow on many MCUs (tens to hundreds of cycles)
- Where it doesn't help: If the modulus is a power of two (compiler replaces with &), or if the CPU has fast hardware division, benefit is small.
- Improvement: Huge on CPUs without hardware division; small on CPUs with division instructions

(c) Use an 8-bit unsigned char

- Why it helps: On 8-bit MCUs, it matches the native word size, saving memory and cycles; also reduces memory usage for large arrays.
- Where it doesn't help: On 32/64-bit CPUs, small integers are promoted to 32-bit for arithmetic, sometimes causing extra instructions and even worse performance.
- Improvement: Useful on 8-bit MCUs or large arrays; negligible or negative for small variables on modern CPUs.

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) Recursion is not allowed because it makes it impossible to guarantee fixed loop bounds and stack usage. This can lead to unpredictable execution time or stack overflow.

(b) Dynamic memory allocation is disallowed after initialization bcs allocators and garbage collection are unpredictable. They can cause fragmentation, timing delays, or memory errors.

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

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

(b) In _____ arithmetic, the overflow flag (V in CPSR) does not indicate anything meaningful about the result of the operation.

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

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

Solution:
(a) signed (c) Unsigned
(b) 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) The add operation without the s suffix doesn't update CPSR flags

(b) CMN works like ADD but doesn't save result.
0xffffffff + 0x1 = 0x0 → Flags : N=0, Z=1, C=1, V=0.

(c) ADDS with 0xFFFFFFFF + 1 also give 0x0, Flags: N=0, Z=1, C=1, V=0
- Signed view: -1+1 = 0
- Unsigned view: 4294967295 + 1 = 0
- Both stored as 0x0

(d) ADD without S again → no flags updated.

(e) 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE

Signed = overflow → 2

Unsigned = 4294967294

Flags = N=1, Z=0, C=0, V=1

Rule for V flag: If operands' sign bits are the same and the result's sign differs, V=1

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

```
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 instructions:

```
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:

1st iteration:

```
while (a != b) { //54 != 24
    if a>b) a = a-b;    a = 54-24=30
    else    b = b- a; }
```

2 nd:

```
while (a != b) {
    if (a>b)        30-24=6
        a = a-b;
    else   b= b-a}
```

3 rd

```
while (a != b) {    6 != 24
    if (a>b)  a=a-b;
    else   b= b-a ; }   b=24-6=18
```

4th:

$b = 18 - 6 = 12$

5 th: $b = 12 - 6 = 6$

6th: $a = b = 6$ return 6

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

```
1st:
gcd   CMP   r1,r2      ; do 54 -24 , C flag set to 1
      BEQ   end        ;
      BLT   lessthan   ;
      SUB   r1,r1,r2   ; do 54 -24, store 30 in r1
      B     gcd        ; back to gcd
2nd:
      CMP   r1,r2      ; do 30 -24 , C f set to 1
      BEQ   end        ;
      BLT   lessthan   ;
      SUB   r1,r1,r2   ; do 30-24, store 6 in r1
      B     gcd        ; go back to gcd
3rd
gcd   CMP   r1, r2     ;   6 -24 , N set 1
      BEQ   end;
      BLT   lessthan   ;   go to lessthan
      SUB   r1, r2;
      B     gcd
lessthan SUB  r2, r2, r1 ; do 24-6, store 18 in r2
      B     gcd        ;   go back gcd
```

4th : 6-18 , N set to 1
go to less than
SUB r3, r2, r1; 18-6 , store 12 in r2
B gcd ;

5th 6-12 N to 1
12-6 store 6 in r2

6th. 6-6 C to 1
go to end

finished gcd was 6

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

1st
gcd CMP r1, r3; 54-24 , C to 1
SUBGT r1, r1, r2 ; store 30 in r1
SUBLT r2, r2, r1;
BNE gcd ; go back

2nd 30-24 ; C to 1
store 6 in r1
go back

3rd 6-24 , N to 1
store 18 in r2
go back

4th: 6-18 , N to 1
store 12 in r2
go back
5th. 6-12 , N to 1
store 6 in r2
go back
6th. 6-6 , C to 1
end with gcd being 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 ARM, we find:
- CMP, SUB take 1 cycle (do SUBLT, SUBGT)
- BEQ, BLT, BNE take 1 cycle if not executed and 1+P cycles if executed
- B takes 1+P cycles

P is the num of cycles required for pipeline refill. range from 1-3.
1st 2nd: 5+P cycles each
3rd, 4th, 5th: 5+2P cycles each
6th: 2+P cycles each
for a total 27+9P cycles:
- 1st → 5th: 4+P cycles each
- 6th: 4 cycles

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