Karan Vora (kv2154)
Real-Time Embedded Systems Assignment 2

Problem 1):

1) : f
2) : c
3) : b
4) : e
5) : d
6) : g
7) : a

==

Problem 2):

(A) Count down to zero loops make use of SUBS along with Z flag whereas if it is up to N loops then it uses ADD and CMP instruction pair along with an extra register to store N. Hence using count down saves an instruction as well as a register. It doesn't decrease performance, it actually saves an instruction.

Depending on the situation, however, it may or may not be necessary or helpful for the programmer to attempt this kind of optimization. For example, if the value of n for the incrementing version is known at compile time, and n is only a loop index variable that is not used anywhere else, then the compiler may perform this optimization itself.

(B) 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.

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) Use an 8-bit unsigned char whenever you have a value that you know won't go beyond 0-255.

Usually the stated goal of this advice is to save on memory space. If you are programming an 8-bit microcontroller, using 8-bit variables whenever possible will often be more efficient and save memory.

However, if you are using a 32-bit microcontroller, manipulating an 8-bit variable can be less efficient than a 32-bit one. The microcontroller may need to pad these variables before doing arithmetic on them, and depending on the address alignment, it may also need to waste space when storing them in memory.

===

Problem 3):

From the Document provided by JPL

(A) Recursion : 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) Dynamic Memory Allocation : There shall be no use of dynamic memory allocation 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.

===

Problem 4):

(A) : Signed
(B) : Unsigned
(C) : Unsigned
(D) : Signed

===

Problem 5):

(A) : All flags remains 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

===

Problem 6):

Solution (A):

Assume a = 54 loaded in r1, b = 24 loaded in r2

First Iteration ;

```
while (a != b) // 54 != 24
{
        if(a > b) // True
        {
                a = a – b // a = 54 -24 = 30
        }
        else // False
        {
                b = b - a
        }
} // Return to while loop
```

Second Iteration ;

```
while (a != b) // 30 != 24
{
        if(a > b) // True
        {
                a = a – b // a = 30 -24 = 6
        }
        else // False
        {
                b = b - a
        }
} // Return to while loop
```

Third Iteration ;

```
while (a != b) // 6 != 24
{
        if(a > b) // False
        {
                a = a – b
        }
        else // True
        {
                b = b – a // b =24 – 6 = 18
        }
} // Return to while loop
```

Fourth Iteration ;

```
while (a != b) // 6 != 18
{
        if(a > b) // False
        {
                a = a – b
        }
```

```
    else // True
    {
            b = b – a // b = 18 – 6 = 12
    }
} // Return to while loop
```

Fifth Iteration ;

```
while (a != b) // 6 != 12
{
    if(a > b) // False
    {
            a = a – b
    }
    else // True
    {
            b = b – a // b = 12 – 6 = 6
    }
} // Return to while loop
```

Sixth Iteration ;

```
while (a != b) // 6 == 6 loop breaks
{
    if(a > b)
    {
            a = a – b
    }
    else
    {
            b = b - a
    }
} // Exit from code
return a; // return 6
```

Solution B):

Assume a = 54 loaded in r1, b = 24 loaded in r2

First Iteration:

```
gcd
        CMP  r1, r2;       ; do 54 – 24, C flag set to 1
        BEQ  end           ; not executed
        BLT  lessthan      ; not executed
        SUB r1, r1, r2     ; do 54 – 24, store 30 in r1
        B  gcd             ; go back to gcd
```

Second Iteration:

```
gcd
        CMP  r1, r2;       ; do 30 – 24, C flag set to 1
        BEQ  end           ; not executed
        BLT  lessthan    ; not executed
        SUB r1, r1, r2    ; do 30 – 24, store 6 in r1
        B  gcd           ; go back to gcd
```

Third Iteration:

```
gcd
        CMP  r1, r2;      ; do 6 – 24, C flag set to 1
        BEQ  end          ; not executed
        BLT  lessthan    ; is executed, go to lessthan
        SUB r1, r1, r2    ;
        B  gcd           ;
lessthan
        SUB  r2, r2, r1   ; do 24 – 6, store 18 in r2
        B  gcd           ; go back to gcd
```

Fourth Iteration:

```
gcd
        CMP  r1, r2;      ; do 6 – 18, C flag set to 1
        BEQ  end          ; not executed
        BLT  lessthan    ; is executed, go to lessthan
        SUB r1, r1, r2    ;
        B  gcd           ;
lessthan
        SUB  r2, r2, r1   ; do 18 – 6, store 12 in r2
        B  gcd           ; go back to gcd
```

Fifth Iteration:

```
gcd
        CMP  r1, r2;      ; do 6 – 12, C flag set to 1
        BEQ  end          ; not executed
        BLT  lessthan    ; is executed, go to lessthan
        SUB r1, r1, r2    ;
        B  gcd           ;
lessthan
        SUB  r2, r2, r1   ; do 12 – 6, store 6 in r2
        B  gcd           ; go back to gcd
```

Sixth Iteration:

```
gcd
        CMP  r1, r2;      ; do 6 – 6, Z, C flag set to 1
        BEQ  end          ; executed
        BLT  lessthan    ;
        SUB r1, r1, r2    ;
        B  gcd           ;
lessthan
        SUB  r2, r2, r1   ;
        B  gcd           ;
end
                         ; finished, gcd was 6
```

Solution C):

Assume a = 54 loaded in r1, b = 24 loaded in r2

First Iteration:
```
gcd
          CMP  r1, r2;          ; do 54 – 24, C flag set to 1
          SUBGT  r1, r1, r2     ; this is executed, store 30 in r1
          SUBLT  r2, r2, r1     ; not executed
          BNE gcd               ; go back to gcd
```

Second Iteration:
```
gcd
          CMP  r1, r2;          ; do 30 – 24, C flag set to 1
          SUBGT  r1, r1, r2     ; this is executed, store 6 in r1
          SUBLT  r2, r2, r1     ; not executed
          BNE gcd               ; go back to gcd
```

Third Iteration:
```
gcd
          CMP  r1, r2;          ; do 6 – 24, C flag set to 1
          SUBGT  r1, r1, r2     ; not executed
          SUBLT  r2, r2, r1     ; this is executed, store 18 in r2
          BNE gcd               ; go back to gcd
```

Fourth Iteration:
```
gcd
          CMP  r1, r2;          ; do 6 – 18, C flag set to 1
          SUBGT  r1, r1, r2     ; not executed
          SUBLT  r2, r2, r1     ; this is executed, store 12 in r2
          BNE gcd               ; go back to gcd
```

Fifth Iteration:
```
gcd
          CMP  r1, r2;          ; do 6 – 12, C flag set to 1
          SUBGT  r1, r1, r2     ; not executed
          SUBLT  r2, r2, r1     ; this is executed, store 6 in r2
          BNE gcd               ; go back to gcd
```

Sixth Iteration:
```
gcd
          CMP  r1, r2;          ; do 6 – 6, C flag set to 1
          SUBGT  r1, r1, r2     ; not executed
          SUBLT  r2, r2, r1     ; not executed
          BNE gcd               ; not executed, end with gcd being 6
```

Solution D):

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).