1. The thumb-2 instruction set support both the thumb instruction set (instructions of 16 bits long) and the arm instruction set (instruction of 32 bit long) with optimisation to use the most appropriate form, thus it allows both code density (32 bits) and efficient memory usage (16 bits ). In brief it does on its own what the thumb and arm instruction set does.
2. The PC hold the address of the next instruction to be executed, buit given that with the thumb instruction set all instruction are half-word aligned (2 bytes), all addresses will be multiple of two, therefore the least significant bit of the address pointed to by the PC is always 0. If instead the arm instruction set is used, the addresses will be multiple of 4 since arm instruction are 32 bits long, once again the two least significant bit is 0. That means whether we used arm or thumb instruction the least significant bit is always set to 0. Hence we can use it to specify the instruction we are using.

2.

a)

MOVS R3, #0 1

loop: ADDS R3, R3, #3 1

MOVS R4, #0xFF 1

CMP R3, R4 1

BEQ somewhere\_else 1 or 3 mostly 1 (not taken)

B loop 1 or 3 mostly 3 (taken)

0xff = 255 and 255/3 = 85 so will go through the loop 85 times before taking the somewhere else. But each time, we spent (1 +1 +1 + 1 +3) = 7 clock cycle and for the last one we will not execute the line B loop since the branch BEQ somewhere\_else is taken. Also there is one cycle to initialise R3 to 0. Hence we have:

1 + (1 +1 +1 + 1 +3)\*84 +( 1 +1 +1 + 3) = 595 clock cycles

b)

We are using a processor of 8MHz i.e. 8000000 cycles per seconds, therefore 595 cycles will take:

595/8000000 = 74.375 micro seconds

3.

a)

1 147 483 647 + 1 234 567 890

Let convert this values to hexadecimal so we can deal with them in terms of bits

1 147 483 647 = 0x446535ff and

1 234 567 890 = 0x499602d2

Adding these two numbers gives 0x8dfb38d1 (or 2382051537 in decimal) which is a negative number since its most significant bit is a 1 (0x8 = 0b1000). Therefore the final result is :

0x80000000 - 0xdfb38d1 = - (0x7204c72f) = -1912915759

b)

c) 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE which is negative (N set) and more than the maximum signed value ( V set) but but Z not set since the result is not a zero, C not set since the are signed..

0xAABBCCDD \*0x7FFFFFFF = 555de66dd5443323 which is a positive number (0x5=0101) while the initial numbers were of opposite sign (0xA = 1010 and 0x7 = 0111), hence the V flag will be set due to the overflow and the

1. No because setting the Z flags mean the result of Rn – Rm = 0, while N is only set if Rn and Rm were different.
2. Only the N flag is set, the Z flag is cleared (if it was set) and the others are unchanged.
3. Both the N and Z flag are set, others are left unchanged
4. Both the N and Z flag are set, others are left unchanged
5. N flag is set while the z flag is cleared. (other will be cleared if they were set before this operation)

4.

a)

0x08001fff has 1 as its least significant bit, so it is not divisible by 2(therefore not divisible by 4 either). Yet we try to use it as pointer to access a word (4 bytes). Not being divisible by 4 implies that it is not word align, hence it is not a valid instruction.

1. LDR R0, foobar load the register R0 with the content (data) of foobar while LDR R0, =foobar load R0 with the memory address (generally in flash) of the label foobar, not its content.

5.

CMP R0,R1

BCS foo

B bar