Q1-a:

Values will be provided in decimal.

$t0	**$t1**	**$t3**
?	?	?
	27	
0		
		0
	23	
1		
	19	
2		
	15	
3		
	11	
4		
	7	
5		
	3	
6		
		1

The final values of $t0, $t1 and $t3 are 6, 3, and 1 respectively. The register $t0 acts as a loop counter, and is used as a final output as well. The value of $t1 is decremented by 4 (our second input stored in $t2) each time the loop is called. This will happen until $t1 is less than the second input, 4, which will be verified when $t3 becomes 1. $t3 is our checking register for when we run the slt command to check if $t1 is less than $t2. The final output tells the user how many times it had to decrement input 1 by input 2 to make input 1 smaller than input 2.

Q1-b:

If the first input is entered as a negative, and the second is entered as a positive, then we will branch to outputting the result without ever decrementing, since the first value is already less than the second value.

If we put a positive integer for the first input, and a negative for the second, the loop will run indefinitely and will never reach the output function, as input 1 will never be smaller than input 2 (since each decrement by a negative is just an increment).

If we put both values as negative, with example input 1 as -27 and input 2 as -4, the program will branch to output 0 since the first value is already smaller than the second. If we switch the inputs and use input 1 as -4 and input 2 as -27, then we get another infinite loop.

Q1-c:

Modified code:

```
.text
 li $v0, 4                    # Syscall to print a string
 la $a0, msg1
 syscall

 li $v0, 5                    # Syscall to read an integer
 syscall
 add $t1, $zero, $v0          # First input stored here

 li $v0, 4                    # Syscall to print a string
 la $a0, msg2
 syscall

 li $v0, 5                    # Syscall to read an integer
 syscall
 add $t2, $zero, $v0          # Second input stored here
 add $t0, $zero, $zero        # Setting a loop counter to 0

 LOOP:
  slt $t3, $t1, $t2           # Checking if first input < second input
  bne $t3, $zero, DONE        # Branch to DONE if above line is true
  sub $t1, $t1, $t2           # Else first input -= second input
  addi $t0, $t0, 1            # Increment loop counter by 1
  j LOOP                      # Jump to loop (continue the loop)

 DONE:
  li $v0, 4 # Syscall to print a string
  la $a0, msg3
  syscall

  li $v0, 1 # Syscall to print a string
  add $a0, $t0, $zero
  syscall

  li $v0, 4 # Syscall to print a string
  la $a0, remainder
  syscall

  li $v0, 1 # Syscall to print a string
  add $a0, $t1, $zero # This will print the remainder
  syscall

  li $v0, 10 # Syscall to exit
  syscall
```

```
.data
  msg1: .asciiz "\nEnter the first
integer: "
  msg2: .asciiz "Enter the second
integer: "
  msg3: .asciiz "Result: "
  remainder: .asciiz "\nRemainder: "
```

Output if we use 27 and 4 as inputs:

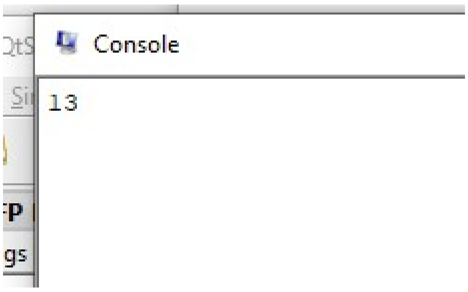
```
Console

Enter the first integer: 27
Enter the second integer: 4
Result: 6
Remainder: 3
```

Q2-a:

Modified code:

```
.text
 la $t0, A_LENGTH
 lw $t0, 0($t0)              # $t0 <- A_LENGTH
 la $t1, A                   # T1 to hold the address of the next array element
 addi $s0, $zero, 0          # Max number, initialised to 0

 NEXT_ARRAY_ELEMENT:
  slt $t3, $zero, $t0        # t3 <- (0 < t0), t3 will be 0 if t0 <= 0
  beq $t3, $zero, DONE

  lw $t2, 0($t1)             # $t2 <- the current array element
  slt $t4, $s0, $t2          # Check if current max is less than the word we just loaded
  bne $t4, $zero, REPLACE

  addiu $t1, $t1, 4          # $t1 += 4 to get address of next element
  addiu $t0, $t0, -1         # Decrementing t0 by 1
  j NEXT_ARRAY_ELEMENT

 REPLACE:
  addi $s0, $t2, 0           # Replace with new minimum
  j NEXT_ARRAY_ELEMENT       # jump to INCREMENT (for loop)

 DONE:
  addi $v0, $zero, 1         # Set syscall to print integer
  add $a0, $s0, $zero
  syscall                    # Prints the integer we just loaded

  li $v0, 10                 # Syscall to exit
  syscall

.data
 A:                          # Our integer array
  .word -1
  .word 4
  .word -16
  .word 0
  .word -2
  .word 5
  .word 13
  .word 2
 A_LENGTH: .word 8           # Length of the array
```

Output from running this code:



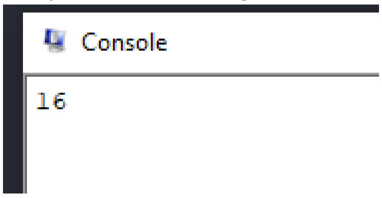
Q2-b:

Modified code:

```
.text
 la $t0, A_LENGTH
 lw $t0, 0($t0)              # $t0 <- A_LENGTH
 la $t1, A                   # T1 to hold the address of the next array element
 addi $s0, $zero, 0          # Max number, initialised to 0

 NEXT_ARRAY_ELEMENT:
  slt $t3, $zero, $t0        # t3 <- (0 < t0), t3 will be 0 if t0 <= 0
  beq $t3, $zero, DONE

  lw $t2, 0($t1)             # $t2 <- the current array element
  slt $s1, $t2, $zero        # Check if word < 0 (negative)
  bne $s1, $zero, ABSOLUTE   # Convert to absolute if true
  j CHECK_MAX

 INCREMENT:
  addiu $t1, $t1, 4          # $t1 += 4 to get address of next element
  addiu $t0, $t0, -1         # Decrementing t0 by 1
  j NEXT_ARRAY_ELEMENT       # Jump back to loop

 ABSOLUTE:
  sub $t2, $zero, $t2        # Convert value to absolute
  j CHECK_MAX

 CHECK_MAX:
  slt $t4, $s0, $t2          # Check if current max is less than the word we just loaded
  bne $t4, $zero, REPLACE
  j INCREMENT

 REPLACE:
  addi $s0, $t2, 0           # Replace with new maximum
  j NEXT_ARRAY_ELEMENT       # jump to INCREMENT (for loop)

 DONE:
  addi $v0, $zero, 1         # Set syscall to print integer
  add $a0, $s0, $zero        # Swap $zero and $s0 to make the value absolute or not
  syscall                    # Prints the integer we just loaded

  li $v0, 10                 # Syscall to exit
  syscall
```

```
.data
 A:                       # Our integer array
  .word -1
  .word 4
  .word -16
  .word 0
  .word -2
  .word 5
  .word 13
  .word 2
 A_LENGTH: .word 8     # Length of the array
```

Output from running this code:



The only difference between the code shown in q2-a and q2-b is the subtraction line we have added under line 13. By doing this subtraction, we make the value of the register absolute, and then we can perform the rest of the program as normal. This will allow us to print the maximum absolute value in the array.

Q2-c-i:

Using the andi command on register $t1 with a value of 7 will store a different value in $t0 based on the bit manipulation (masking) taking place, which we can then compare to zero, allowing us to determine if a number is divisible by 8, since if the value in $t0 is 0, the number that we had originally was divisible by 8, but if we have any value apart from 0 then our number is not a perfect multiple of 8.

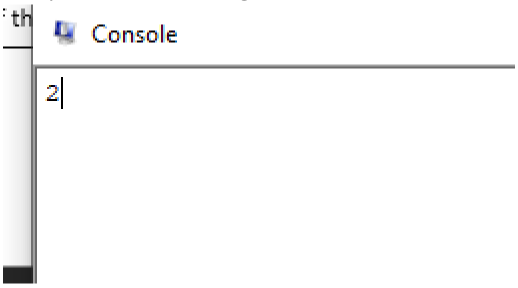
Q2-c-ii:

Modified code:

```
.text
 la $t0, A_LENGTH
 lw $t0, 0($t0)                # $t0 <- A_LENGTH
 la $t1, A                     # T1 to hold the address of the next array element
 addi $s0, $zero, 0            # Number of perfect multiples, initialised to 0

 NEXT_ARRAY_ELEMENT:
  slt $t3, $zero, $t0                      # t3 <- (0 < t0), t3 will be 0 if t0 <= 0
  beq $t3, $zero, DONE

  lw $t2, 0($t1)                           # $t2 <- the current array element
  slt $s1, $t2, $zero                      # Check if word < 0 (negative)
  bne $s1, $zero, ABSOLUTE                 # Convert to absolute if true
  beq $t2, $zero, PERFECT_MULTIPLE         # Special case if word = 0
  j MASK

 PERFECT_MULTIPLE:
  addi $s0, $s0, 1          # s0 += 1
  j INCREMENT               # jump to INCREMENT (for loop)

 INCREMENT:
  addiu $t1, $t1, 4              # $t1 += 4 to get address of next element
  addiu $t0, $t0, -1             # Decrementing t0 by 1
  j NEXT_ARRAY_ELEMENT           # Jump back to loop

 ABSOLUTE:
  sub $t2, $zero, $t2            # Convert value to absolute
  j MASK

 MASK:
  andi $t4, $t2, 0x0007                  # Mask the integer
  beq $t4, $zero, PERFECT_MULTIPLE       # Branch if remainder = 0
  j INCREMENT                            # Jump back to the loop

 DONE:
  addi $v0, $zero, 1                     # Set syscall to print integer
  add $a0, $s0, $zero
  syscall                                # Prints the integer we just loaded

  li $v0, 10 # Syscall to exit
  syscall
```

```
.data
  A:                              # Our integer array
    .word -1
    .word 4
    .word -16
    .word 0
    .word -2
    .word 5
    .word 13
    .word 2
  A_LENGTH: .word 8         # Length of the array
```

Output from running this code:



Q3-a:
Register $rd stores the shifted value of $rt after performing the sll instruction. When performing a sll instruction on $rt with the shift amount h, we are essentially saying that the integer in $rt needs to be multiplied by 2^h, which will perform a left shift on $rt h times, and store the result in $rd. The operation being performed by running the command *sll $rd, $rt, h* is *$rd = $rt * 2^h*.

Q3-b:
The sll instruction is an R-format instruction.

Opcode: 000000
Rs: 00000
Rd: 01001
Rt: 01000
Shift: 00010
Function code: 000000

For sll instructions (and all other R instructions), the opcode is 0. Since it is stored in 6 bits, we have 6 bits of zeros. The rs part of the instruction is not needed, so is always going to be 5 bits of 0. In this instance, we are declaring the destination register, rd, as $t1, which is register 8. The 5-bit binary for 8 is 01000. Our source register, rt, is register 9, which is 01001 in binary. We are shifting our original value by 2, which is 00010 in the 5 bits of the instruction. We also need a function code, which, in the case of the sll instruction, is 000000. So, the final 32 bit instruction for sll $t0, $t1, 2 would be 00000000000010010100000010000000.

Q3-c-iii:
$t1 ← 8 × $t1 (1 instruction)

```
sll $t1, $t1, 3      # Perform a shift of 8
```

Q3-c-iv:
$t1 ← 24 × $t1 (3 instructions)

```
sll $t0, $t1, 3      # Perform a shift of 8
add $t2, $t0, $t0    # Multiply by 2 and store in $t2
add $t1, $t2, $t0    # Add another $t0 to $t2 to make it effectively $t2 x 3, store in $t1
```

Q3-c-v:
$t1 ← 28 × $t1 (3 instructions)

```
sll $t0, $t1, 5      # Multiply by 32
sll $t2, $t1, 2      # Multiply by 4
sub $t1, $t0, $t2    # 32 - 4 = 28
```

Q3-c-vi:
$t1 ← 63 × $t1 (2 instructions)

```
sll $t0, $t1, 6      # Multiply by 64
sub $t1, $t0, $t1    # 64 - 1 = 63
```

Bibliography:

1. Cs.kzoo.edu. 2020. MIPS Instruction Formats. [online] Available at: <http://www.cs.kzoo.edu/cs230/Resources/MIPS/MachineXL/InstructionFormats.html> [Accessed 30 November 2020].

2. Electrical Engineering Stack Exchange. 2020. What Is The Meaning Of "Register.Rd"?. [online] Available at: <https://electronics.stackexchange.com/questions/68788/what-is-the-meaning-of-register-rd> [Accessed 3 December 2020].

3. GitHub. 2020. MIPT-Ilab/Mipt-Mips. [online] Available at: <https://github.com/MIPT-ILab/mipt-mips/wiki/MIPS-pseudo-instructions> [Accessed 27 November 2020].

4. Math.unipd.it. 2020. [online] Available at: <https://www.math.unipd.it/~sperduti/ARCHITETTURE-1/mips32.pdf> [Accessed 1 December 2020].

5. Stack Overflow. 2020. AND Command For MIPS Assembly To See If Number Is Divisible By A Power Of 2?. [online] Available at: <https://stackoverflow.com/questions/29597560/and-command-for-mips-assembly-to-see-if-number-is-divisible-by-a-power-of-2> [Accessed 30 November 2020].

6. Stack Overflow. 2020. Integer Absolute Value In MIPS?. [online] Available at: <https://stackoverflow.com/questions/2312543/integer-absolute-value-in-mips> [Accessed 30 November 2020].

7. Stack Overflow. 2020. MIPS Assembly Return To Call In A Branch Statement. [online] Available at: <https://stackoverflow.com/questions/40559802/mips-assembly-return-to-call-in-a-branch-statement> [Accessed 30 November 2020].

8. Stack Overflow. 2020. What Is The "Relationship" Between Addi And Subi?. [online] Available at: <https://stackoverflow.com/questions/11981660/what-is-the-relationship-between-addi-and-subi> [Accessed 3 December 2020].

9. Stack Overflow. 2020. Why Shift A Bit Using Sll And Such In Mips Assembly?. [online] Available at: <https://stackoverflow.com/questions/32487502/why-shift-a-bit-using-sll-and-such-in-mips-assembly> [Accessed 3 December 2020].

10. Youtube.com. 2020. [online] Available at: <https://www.youtube.com/playlist?list=PL5b07qlmA3P6zUdDf-o97ddfpvPFuNa5A> [Accessed 26 November 2020].