Number Systems

1. (a)  1001 0111
   (b)  0011 1111
   (c)  0001 1001 1110 0011 1010 0111 1111 0011. Note that it is easy to convert hexadecimal numbers to binary numbers by converting one nibble (4 bits) at a time.

2. (a) 0xD53F
   (b) 0x79E9
   (c) 0x97

3. (a) The easiest way to compute a negative number is to take the corresponding positive number and then to take the two’s complete of this number. The procedure is as follows (see also Lecture 3): The first step is to convert the non negative decimal number 59\textsubscript{10} into binary form. In this case, a 12-bit number. That is, 59\textsubscript{10} is 0000 0011 1011 in binary form. This is followed by inverting all the bits, resulting in 1111 1100 0100. Finally, we add 1 to this number 1111 1100 0101. The most significant bit is 1, meaning that the sign bit is set. Thus, the sign extended 16-bit value is 1111 1111 1100 0101, or 0xFFC5 in hexadecimal form. If we instead zero extend 1111 1110 0101, we only add zeros to the most significant bits, that is, the zero extended value is 0x0FC5.

   (b) The sign extended value is 0xFE73 and the zero extended value is 0x0E73.

   (c) The sign extended and zero extended variants are the same: 0x027E

4. The most significant byte (MSB) is 0x32 and the least significant byte (LSB) is 0x21. A byte is number consisting of 8 bits.

   The meaning of the term word depends on the processor. If the processor operates on 16-bit registers, it is a 16-bit processor and the word size is 16 bits. For MIPS32, the word size is 32 bits, that is, 4 bytes. 64-bit processors operate on 64 bits wide words.

   A nibble is a 4-bit number. A byte consists of two nibbles. A nibble correspond to one character when a number is written in hexadecimal form.
Assembly Languages

5. A possible solution is given in the code listing below:

```
data
.align 2

# Test buffers
src: .ascii "0123456789abcdef"
.space 16
dst1: .space 32
dst2: .space 32

.text
main:
    # TEST 1
    # Copy from an aligned position to an aligned position, 
    # using an aligned number of bytes.
    la $a0, src
    la $a1, dst1
    li $a2, 14  # copy 14 bytes
    jal memcpy

    # TEST 2
    # Copy from as well as to unaligned positions.
    la $a0, src
    addi $a0, $a0, 5  # make the destination address unaligned
    la $a1, dst2
    li $a2, 7       # copy 7 bytes
    jal memcpy

stop:   j stop

# Simple memory copy function. One byte at a time is 
# copied.
# Input: $a0 = source address
#        $a1 = destination address
#        $a2 = number of bytes to copy

memcpy:
    loop:
        beq $a2, $zero, done  # branch if done
        lb $t0, 0($a0)       # load byte
        sb $t0, 0($a1)       # store byte
        addi $a0, $a0, 1    # increment src pointer
        addi $a1, $a1, 1    # increment dst pointer
        addi $a2, $a2, -1   # decrement counter
        j loop

done:
    jr $ra
```

The main function contain two test cases. The first test copies data, where both the source and the destination addresses are aligned. The second test copies data to and from unaligned addresses.
6. This exercise has many possible solutions. The following solution is optimized for the case when both the source and destination addresses are aligned. It is general and can handle the same input as the previous solution, but is faster in some cases. Can you find a way to design an even faster implementation?

```assembly
# Simple memory copy function. One byte at a time is copied.
# Input: $a0 = source address
# $a1 = destination address
# $a2 = number of bytes to copy
memcpy:
    # check if aligned
    andi $t0,$a0,3
    andi $t1,$a1,3
    or $t0,$t0,$t1
    bne $t0,$zero,byteloop

    # copy word by word
    wordloop:
        slti $t0,$a2,4 # check if fewer than 4 bytes
        bne $t0,$zero,byteloop
        lw $t0,0($a0) # load word
        sw $t0,0($a1) # store word
        addi $a0,$a0,4 # increment src pointer
        addi $a1,$a1,4 # increment dst pointer
        addi $a2,$a2,-4 # decrement counter
        j wordloop

    # standard, byte by byte of the rest
    byteloop:
        beq $a2,$zero,done # branch if done
        lb $t0,0($a0) # load byte
        sb $t0,0($a1) # store byte
        addi $a0,$a0,1 # increment src pointer
        addi $a1,$a1,1 # increment dst pointer
        addi $a2,$a2,-1 # decrement counter
        j byteloop

done:
    jr $ra
```
7. (a) The program consists of three functions. The main function is the start of the program and calls function makelist. The second argument of the call has value 8, which is calculated by dividing the list length in bytes (sizeof(factlist)) which is 32 bytes, by the length of an integer word (indicated by sizeof(int)). The makelist function itself calls function fact the number of times that is indicated in the argument length. The result is stored in the array factlist. Finally, function fact computes the factorial number \( n! \) of \( n \).

(b) # Complete code listing for the factlist program
# You may copy & paste this solution into the MARS simulator.
# Macros used for saving register on the stack
.macro PUSH (%reg)
    addi $sp, $sp, -4
    sw %reg, 0($sp)
.end_macro
.macro POP (%reg)
    lw %reg, 0($sp)
    addi $sp, $sp, 4
.end_macro

# start of section
.data
.align 2
factlist: .space 32 # 8 words, each 4 bytes long

.text

# Start of test program
start:
    jal main
stop:     j stop

# Factorial function n!
# Input: $a0 = value n
# Output: $v0 = the new value n!
fact:    addi $v0, $0, 1 # r = 1

factloop:
    ble $a0, $0, donefact # n <= 0
    mul $v0, $v0, $a0 # r = r * n
    addi $a0, $a0, -1 # n--
    j factloop

donefact:
    jr $ra
# Creates a list of factorial numbers
# Input: $a0 = start address
# $a1 = length in bytes
makelist:
    PUSH ($ra)
PUSH ($s0)  # for start
PUSH ($s1)  # for length
PUSH ($s2)  # for i
PUSH ($s3)  # for factlist address
move $s0,$a0  # save start
move $s1,$a1  # save length
addi $s2,$0,0  # i = 0
la $s3,factlist  # factlist address
makeloop:
    slt $t0,$s2,$s1  # i < length
    beq $t0,$0,makeend  # jump if end of while
move $a0,$s0  # setup argument
jal fact  # call fact
sll $t0,$s2,2  # get correct word address
add $t1,$s3,$t0  # adds address & i counter
sw $v0,0($t1)  # store the result
addi $s0,$s0,1  # start++
addi $s2,$s2,1  # i++
j makeloop
makeend:
    POP ($s3)
POP ($s2)
POP ($s1)
POP ($s0)
POP ($ra)
jr $ra

# Main function
main:  PUSH ($ra)
addi $a0,$0,3
addi $a1,$0,8
jal makelist
POP ($ra)
jr $ra

(c) A .global directive needs to be added. In this case:
   .global makelist

(d) The easiest way to update the program is to start with adding nop (no operation) instructions after each jump or branch. Then, after that, we can optimize the program and change the order of instructions. Please see one possible solution below. Note also that we removed the POP and PUSH macro calls, and instead replaced it with just one addi instruction, followed by store and load words.
# Factorial function n!
# Input: $a0 = value n
# Output: $v0 = the new value n!

fact: addi $v0,$0,1  # r = 1

factloop:
  ble $a0,$0,donefact  # n <= 0
  nop
  mul $v0,$v0,$a0  # r = r * n
  addi $a0,$a0,-1  # n--
  j factloop
  nop

donefact:
  jr $ra
  nop

# Creates a list of factorial numbers
# Input: $a0 = start address
# $a1 = length in bytes

makelist:
  addi $sp,$sp,-20  # removes one instruction for each push
  sw $ra,16($sp)
  sw $s0,12($sp)
  sw $s1,8($sp)
  sw $s2,4($sp)
  sw $s3,0($sp)

  move $s0,$a0  # save start
  move $s1,$a1  # save length
  addi $s2,$0,0  # i = 0

  la $s3,factlist  # factlist address

makeloop:
  slt $t0,$s2,$s1  # i < length
  beq $t0,$0,makeend  # jump if end of while
  move $a0,$s0  # setup argument  # No NOP is needed
  jal fact
  nop
  sll $t0,$s2,2  # get correct word address
  add $t1,$s3,$t0  # adds address & i counter
  sw $v0,0($t1)  # store the result
  addi $s0,$s0,1  # start++
  j makeloop
  addi $s2,$s2,1  # i++  # By changing the order,
                   # we save one instruction.

makeend:
  lw $s3,0($sp)
  lw $s2,4($sp)
  lw $s1,8($sp)
  lw $s0,12($sp)
  lw $ra,16($sp)
  addi $sp,$sp,20
  jr $ra
  nop
Machine Languages

8. (a) It is a pseudo instruction, meaning that the assembler replaces the pseudo instruction with basic instructions. In this case, the assembler may replace the `li` instruction with

```
lui $at, 0x1234
ori $t1, $at, 0x89ab
```

(b) Address       Byte value
0xffff0004 0x12
0xffff0005 0x34
0xffff0006 0x89
0xffff0007 0x12

(c) Address       Byte value
0xffff0004 0xab
0xffff0005 0x89
0xffff0006 0x34
0xffff0007 0x12

9. (a) 0x02538820
0x214afffc
0xa1b1ffb2
0x1540fff

(b) By looking at the MIPS reference sheet, we can see that `addi` is an I-type instruction. We can express the encoding easily in C by using the bitwise shift left operator `<<` to move bits to the left, and the to combine the results using the bitwise OR operator `|`. Starting from left to the right, we first encode the the op-code. For `addi`, the op-code is 8. Since the opcode field is located between bits 31 and 26, we shift the opcode value 26 bits to the left, resulting in `(8 << 26)`. Next, we would like to encode the `rs` field. The register in this example is $t2, which has register code 10. Hence, we can encode the `rs` field as `(10 << 21)` since the `rs` field is located between bits 25 and 21. In the same way, we get `(10 << 16)` for the `rt` field. Finally, we know that we should encode a negative number $-4$ in the immediate field. Because this field is located between bits 15 and 0, we can use the immediate value almost directly. The only thing we have to do is to mask the 16 lowest bits, which can be done as follows: `(0xffff & -4)`. Finally, we combine all these subexpressions with the OR-operator, and we get the C expression:

```
(8 << 26) | (10 << 21) | (10 << 16) | (0xffff & -4)
```
We can test this expression, by creating the following small C program.

```c
#include <stdio.h>
int main()
{
    int c = (8 << 26) | (10 << 21) | (10 << 16) | (0xffffff & -4);
    printf("0x%08X\n", c);
}
```

If we execute this program, we get the result 0x214AFFFC, which corresponds to the encoding we did by hand in the previous exercise.