1. (45 points)

```assembly
addi $v0, $zero, -1  # initialize to avoid counting zero word

loop:
lw  $v1, 0($a0)      # read next word from source
addi $v0, $v0, 1    # increment count words copied
sw   $v1, 0($a1)     # write to destination
addi $a0, $a0, 4    # advance pointer to next source
addi $a1, $a1, 4    # advance pointer to next destination
bne  $v1, $zero, loop  # loop if word copied != zero

Bug 1: count ($v0) is initialized to zero, not -1 to avoid counting zero word
Bug 2: count ($v0) is not incremented.
Bug 3: loops if word copied is equal to zero rather than not equal.
```

### b: assume a[] elements are 32-bit words

```assembly
main:
add  $s0, $0, $0  # initialize j to 0
addi $s1, $0, 20 # initialize loop limit to 20
add  $t0, $a1, $0 # initialize base addr

loop:
  beq  $s0, $s1, end  # if j = 20, goto end
  lw   $t1, 0($t0)    # get a[j]
  lw   $t2, 4($t0)    # get a[j+1]
  add  $t1, $t1, $t2  # a[j] = a[j+1] + a[j]
  sw   $t1, 8($t0)    # store a[j] to a[j+2]
  addi $t0, $t0, 4    # next element
  addi $s0, $s0, 1    # increment j
  j    loop

end:   nop
```

2. (10 points)

The data below show the values stored in memory:

(i) 0xAE0B0008

(ii) 0x8D080040

(a) (2 points) Is it possible to say whether these values represent instructions or data?

No. A value in memory can be either data or instruction depending on how it is treated by datapath.
(b) (6 points) Suppose that this data are MIPS instructions. What instructions do they represent? In other words show assembly code for each of them.

(c) (2 points) What type (I-type, J-type, or R-type) instruction do the hexadecimal entries above represent?

(i) \( \text{0xAE0B0008} \)

\[
\begin{align*}
= & \quad (101011) \ (10000) \ (01011) \ (0000 \ 0000 \ 0000 \ 1000) \\
= & \quad (\text{Opcode}) \quad (\text{Rs}) \quad (\text{Rt}) \quad (\text{I}) \\
= & \quad \text{sw} \ \text{Rt} \ \text{offset}_{10}(\text{Rs}) \ # \ \text{offset} \ \text{is} \ \text{the} \ \text{decimal} \ \text{equivalent} \ \text{of} \ \text{the} \ 16\text{bit} \ \text{Immediate} \ (\text{I}). \\
= & \quad \text{sw} \ \$11 \ 8(\$16) \\
= & \quad \text{sw} \ \$t3 \ 8(\$s0) \ -- \ \text{Type - Immediate}
\end{align*}
\]

(ii) \( \text{0x8D080040} \)

\[
\begin{align*}
= & \quad (100011) \ (01000) \ (01000) \ (0000 \ 0000 \ 0100 \ 0000) \\
= & \quad (\text{Opcode}) \quad (\text{Rs}) \quad (\text{Rt}) \quad (\text{I}) \\
= & \quad \text{lw} \ \text{Rt} \ \text{offset}_{10}(\text{Rs}) \ # \ \text{offset} \ \text{is} \ \text{the} \ \text{decimal} \ \text{equivalent} \ \text{of} \ \text{the} \ 16\text{bit} \ \text{Immediate} \ (\text{I}). \\
= & \quad \text{lw} \ \$8 \ \text{64}(\$8) \\
= & \quad \text{lw} \ \$t0 \ \text{64}(\$t0) \ -- \ \text{Type – Immediate}
\end{align*}
\]

3. (20 points)

(a) (10 points) Convert the following MIPS instructions into corresponding hexadecimal representation.

(b) (5 points) What type (I-type, J-type, or R-type) instruction are shown above in (i)-(v)?

(c) (5 points) What addressing modes (immediate, register, base, PC-relative or pseudo direct) are used in (i)-(v)?

*Using the green card, form a 32 bit binary number for each instruction and convert to hex.*

(i) add \$t0, \$t1, \$0

\( \text{0x 01204020} \) R-Type; Register addressing

(ii) lw \$t1, -4(\$s3)
0x 8e69fff  I-Type; Base addressing

(iii) slti $s1, $v0, -1

0x2851ffff I-Type; Immediate addressing

(iv) jr $ra

0x03e00008 R-Type; Register addressing

(v) srl $a0, $t0, 10

0x00082282 R-Type; Immediate addressing

4. (25 points) Assume that the code is placed at location 0x0040 0000 in memory. Show the machine code (in hex) for this loop

Code for implementation without delayed branch

```
LOOP: lw $t0, 0($a0)  0x0040 0000:  0x8c880000
   lw $t1, 0($a1)  0x0040 0004:  0x8ca90000
   beq $t0, $0, EXIT  0x0040 0008:  0x11000004
   addi $a0, $a0, 4  0x0040 000C:  0x20840000
   addi $a1, $a1, 4  0x0040 0010:  0x20a50004
   bne $a0, $a2, LOOP  0x0040 0014:  0x1486ffff
EXIT:  0x0040 0018:
```

Code for implementation with delayed branch

```
LOOP: lw $t0, 0($a0)  0x0040 0000:  0x8c880000
   lw $t1, 0($a1)  0x0040 0004:  0x8ca90000
   beq $t0, $0, EXIT  0x0040 0008:  0x11000003
   addi $a0, $a0, 4  0x0040 000C:  0x20840004
   addi $a1, $a1, 4  0x0040 0010:  0x20a50004
   bne $a0, $a2, LOOP  0x0040 0014:  0x1486fffa
EXIT:  0x0040 0018:
```
How to solve this:

Except branching instructions (like bne, beq, j, etc) other instructions does not depend on memory location (address) values. Using the green card the machine code for non-branching instructions can be easily computed as in problem 2.

Assuming first that there is implementation without delayed branch here is how to calculate for beq:

```
beq $t0, $0, EXIT
```

= opcode(beq) $8 $0 (instruction offset) # instruction offset is simply the distance count of the label from the current position.

= (000100) (01000) (00000) (410)

= (000100) (01000) (00000) (0000 0000 0000 0100)

= 0X11000004

Similarly, for bne $a0, $a2, LOOP

= opcode(bne) $a0 $a2 (instruction offset) # Label ‘LOOP’ is -510 instruction count away from the current position.

= (opcode(bne)) ($a0) ($a2) (1111 1111 1111 1101) (-5 in sign extended form)

= 0x1486fff

For implementation with delayed the distance count is always from next instruction so that it us 3 for the first branch and -6 for the second one.

5. (45 points) The sequence $F_n$ of Fibonacci numbers is defined using recurrence relation $F_n = F_{n-1} + F_{n-2}$ for $n > 2$ and assuming $F_1 = 1$ and $F_2 = 1$. In c language one can use the following recursive function to obtain any number in this sequence:

```c
int fib_iter(int a, int b, int n) {
    if (n==2)
return a;
else
return fib_iter(a+b, a, n-1);
}

(a) (25 points) Write a MIPS code for such recursive function. Assume that \( a, b, \) and \( n \) are passed with \( \text{\$a0}, \text{\$a1} \) and \( \text{\$a2} \), respectively, while the return value is in \( \text{\$v0} \).

CODE:

```mips
.data
.globl output_str
output_str: .asciiz "The output of fibonacci series for N($a0) is\n"
.text
.globl main

main:
    #main has to be a global label
    addi $sp, $sp, -4   #Move the stack pointer
    sw  $ra, 0($sp)    #save the return address
    li  $a1, 1   # (a) arg 1 to function fibonacci(int a, int b, int N)
    li  $a2, 1   # (b) arg 2 to function fibonacci(int a, int b, int N)
    li  $a0, 5   # (N) arg 3 to function fibonacci(int a, int b, int N)
    jal fibonacci
    add $t1, $v0, $0
    li  $v0, 4   #print string (system call 4)
    la  $a0, output_str   # takes the address of string as an
    syscall  # argument
    li  $v0, 1   #print integer (system call 1)
    add  $a0, $t1, $0   # takes the address of string as an
    syscall  # argument

    lw  $ra, 0($sp)   #Usual stuff at the end of the main
    addi $sp, $sp, 4   #restore the return address
    jr  $ra    #return to the main program

fibonacci:
    # Prologue: set up stack pointers for fibonacci
    # Need two local variables to hold the results of the two
    # recursive calls to Fibonacci
```
addi $sp, $sp, -8  # allocate stack space
sw $a1, 0($sp)    # save $a1 on the stack
sw $ra,4($sp)    # save return address

N2:
# if N == 2, return a (by moving 'a' into $v0)
li $t0, 2            
beq $a0, $t0, N3    
add $v0, $a1, $0    #OUTPUT in $v0
j fibend

N3:
addi $a0, $a0, -1    # compute N - 1
add $a1, $a1, $a2    # compute b=a
lw $t1, 0($sp)       # compute a=a+b
add $a2, $t1, $0
jal fibonacci
fibend:
# Epilogue: restore stack and frame pointers and return
lw $ra,4($sp)       # restore return address
addiu $sp, $sp, 8   # restore caller's stack pointer
jr $ra    # return

########END OF FUNCTION############

(b) (5 points) Show the context of the stack for the case n = 4, i.e. when MIPS function is called with a = 1, b = 1 and n = 4. Assume that the value of $sp is 0x7FFF FFFC before the function call, the function is called from the instruction at the location 0x0040 0F00 and the first instruction of the function itself is at the location 0x0040 F000.

STACK FRAME

<table>
<thead>
<tr>
<th>N=sa0=2</th>
<th>$a1 = 3 (return $a1)</th>
<th>$sp = 0x7ffe4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B=sa2=2</td>
<td>$ra = 0x0040F030</td>
<td>--------------</td>
</tr>
<tr>
<td>N=sa0=3</td>
<td>$a1 = 2</td>
<td>$sp = 0x00fffec</td>
</tr>
<tr>
<td>B=sa2=1</td>
<td>$ra = 0x0040F030</td>
<td>After 2nd call to fn (Section 3)</td>
</tr>
<tr>
<td>N=sa0=4</td>
<td>$a1=1</td>
<td>$sp = 0x00fff4</td>
</tr>
<tr>
<td>B=sa2=1</td>
<td>$ra = 0x00400f04</td>
<td>After 1st call to fn (Section 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$sp = 0x7fff fff4</td>
</tr>
</tbody>
</table>

After 1st call to fn (Section 1)

After 2nd call to fn (Section 3)

After 3rd call to fn

$sp = 0x7ffe4
Before entering the function, the values of (N,a,b) are (4,1,1). These values are stored in the registers ($a0, $a1, $a2). At this time the value of $sp is 0x7FFF FFFC. When the function 'fibonacci' is called by main (see section 1 in the code) by the instruction "JAL fibonacci" from the location 0x0040 0F00, $ra value is updated to the next instruction's address (instruction after JAL fibonacci) automatically. So, $ra value becomes 0x0040 0F04 after function call.

Function definition starts from 0x0040 F000.

After the 1st call (see section 2 in the code):

1. $ra and $a1 are stored into the stack. $sp value decrements by 8 to 0x7FFF FFF4.

2. 
   N = $a0 = n-1 = 3 
   a = $a1 = a+b = $a1 + $a2 = 2 
   b = $a2 = a = $a1 = 1

After the 2nd function call from within function call 1 (see section 3 in the code):

1. $ra value changes to the instruction location after "JAL fibonacci" (2nd call) which is 0x0040 F000 + 4*(1210). '12' is the value obtained by counting the number of instructions from the start of the function definition to where JAL fibonacci is called 2nd time. So $ra becomes 0x0040 F0030. $sp decrements by 8 to 0x7FFF FFEC.

2. 
   N = $a0 = n-1 = 2 
   a = $a1 = a+b = $a1 + $a2 = 3 
   b = $a2 = a = $a1 = 2

After 3rd call from within function call 2:

1. $ra remains the same, because the return instruction after executing the function is the same. (this is only true if the same function is called because the function definition is the same).

2. $sp becomes 0x7FFF FFE4.
3. Function returns a = $a1 = 3.

NOTE:

The values of $sp is not unique. It depends on the number of values stored in the stack.
The value of $ra$ is also not unique. It depends on the number of instructions in the code.