Week 05 Tutorial Questions

1. If the data segment of a particular MIPS program starts at the address `0x10000020`, then what addresses are the following labels associated with, and what value is stored in each 4-byte memory cell?

```
    .data
a:  .word   42
b:  .space  4
c:  .asciiz "abcde"
    .align  2
d:  .byte   1, 2, 3, 4
e:  .word   1, 2, 3, 4
f:  .space  1
```

2. Give MIPS directives to represent the following variables:

   a. `int u;`
   b. `int v = 42;`
   c. `char w;`
   d. `char x = 'a';`
   e. `double y;`
   f. `int z[20];`

   Assume that we are placing the variables in memory, at an appropriately-aligned address, and with a label which is the same as the C variable name.

3. Consider the following memory state:

Address	Data	Definition
0x10010000	aa:	.word 42
0x10010004	bb:	.word 666
0x10010008	cc:	.word 1
0x1001000C		.word 3
0x10010010		.word 5
0x10010014		.word 7

   What address will be calculated, and what value will be loaded into register `$t0`, after each of the following statements (or pairs of statements)?

   a.
```
la   $t0, aa
```
   b.
```
lw   $t0, bb
```
   c.
```
lb   $t0, bb
```
   d.
```
lw   $t0, aa+4
```
   e.
```
la   $t1, cc
lw   $t0, ($t1)
```
   f.
```
la   $t1, cc
lw   $t0, 8($t1)
```
   g.
```
li   $t1, 8
lw   $t0, cc($t1)
```
   h.
```
la   $t1, cc
lw   $t0, 2($t1)
```

4. What is a breakpoint?

   When is it useful in debugging?

5. Translate this C program to MIPS assembler

```c
#include <stdio.h>

int main(void) {
    int i;
    int numbers[10] = {0};

    i = 0;
    while (i < 10) {
        scanf("%d", &numbers[i]);
        i++;
    }
}
```

6. Translate this C program to MIPS assembler

```c
#include <stdio.h>

int main(void) {
    int i;
    int numbers[10] = {0,1,2,3,4,5,6,7,8,9};

    i = 0;
    while (i < 10) {
        printf("%d\n", numbers[i]);
        i++;
    }
}
```

7. Translate this C program to MIPS assembler

```c
int main(void) {
    int i;
    int numbers[10] = {0,1,2,-3,4,-5,6,-7,8,9};

    i = 0;
    while (i < 10) {
        if (numbers[i] < 0) {
            numbers[i] += 42;
        }
        i++;
    }
}
```

8. Translate this C program to MIPS assembler

```c
#include <stdio.h>

int main(void) {
    int i;
    int numbers[10] = {0,1,2,3,4,5,6,7,8,9};

    i = 0;
    while (i < 5) {
        int x = numbers[i];
        int y = numbers[9 - i];
        numbers[i] = y;
        numbers[9 - i] = x;
        i++;
    }
}
```

9. The following loop determines the length of a string, a `'\0'`-terminated character array:

```c
char *string = "....";
char *s = &string[0];
int   length = 0;
while (*s != '\0') {
   length++;  // increment length
   s++;       // move to next char
}
```

Write MIPS assembly to implement this loop.

Assume that the variable `string` is implemented like:

```
   .data
string:
   .asciiz  "...."
```

Assume that the variable `s` is implemented as register `$t0`, and variable `length` is implemented as register `$t1`. And, assume that the character `'\0'` can be represented by a value of zero.

10. Conceptually, the MIPS pseudo-instruction to load an address could be encoded as something like the following:

Since addresses in MIPS are 32-bits long, how can this instruction load an address that references the data area, such as `0x10010020`?

11. Implement the following C code in MIPS assembly instructions, assuming that the variables `x` and `y` are defined as global variables (within the `.data` region of memory):

```
long x;     // assume 8 bytes
int  y;     // assume 4 bytes

scanf("%d", &y);

x = (y + 2000) * (y + 3000);
```

Assume that the product might require more than 32 bits to store.

12. Write MIPS assembly to evaluate the following C expression, leaving the result in register `$v0`.

```
((x*x + y*y) - x*y) * z
```

Write one version that minimises the number of instructions, and another version that minimises the number of registers used (without using temporary memory locations).

Assume that: all variables are in labelled locations in the `.data` segment; the labels are the same as the C variable names; all results fit in a 32-bit register (i.e., no need to explicitly use `Hi` and `Lo`).