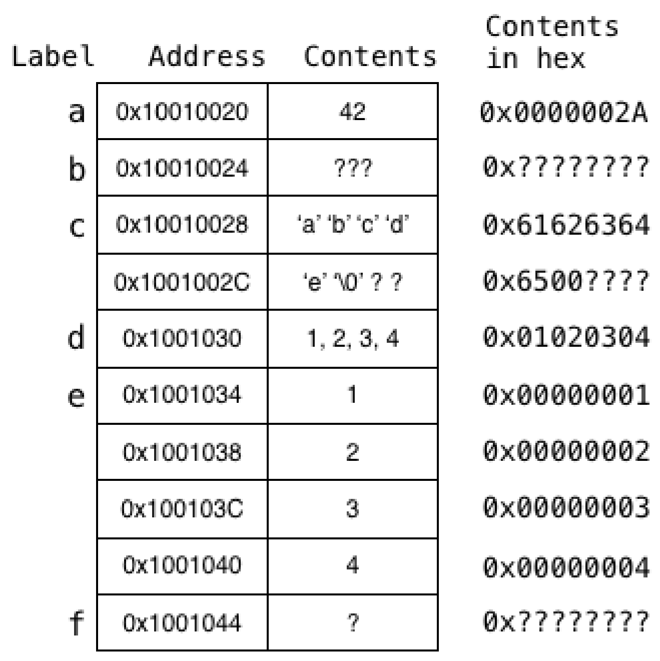
Week 05 Tutorial Answers

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
```

Answer:

Memory map showing labels, associated addresses, and contents of memory cells:



Label	Address	Contents	Contents in hex
a	0x10010020	42	0x0000002A
b	0x10010024	???	0x????????
c	0x10010028	'a' 'b' 'c' 'd'	0x61626364
	0x1001002C	'e' '\0' ? ?	0x6500????
d	0x1001030	1, 2, 3, 4	0x01020304
e	0x1001034	1	0x00000001
	0x1001038	2	0x00000002
	0x100103C	3	0x00000003
	0x1001040	4	0x00000004
f	0x1001044	?	0x????????

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.

Answer:

Note that `.space` *n* allocates an uninitialised *n*-byte region of memory, while `.word` *v* allocates four bytes (one *word*) of space, initialised with the value *v*.

   a. `u: .space 4`
   b. `v: .word 42`
   c. `w: .space 1`
   d. `x: .byte 'a';`
   e. `y: .space 8`
   f. `z: .space 80` (20 * 4-byte ints)

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)
```

Answer:

a.
```
la   $t0, aa
```
Loads the address associated with the label `aa` into `$t0`. Since `aa` is located at `0x10010000`, the value loaded into `$t0` is `0x10010000`.

b.
```
lw   $t0, bb
```
Load the contents of the 4-byte memory cell associated with the label `bb` into the register `$t0`. Since `bb` is located at `0x10010004`, the address used for this instruction is `0x10010004`. The contents of that address is `666`, and this is the value loaded into register `$t0`.

c.
```
lb   $t0, bb
```
Load the contents of the 1-byte memory cell associated with the label `bb` into the register `$t0`. The address is clear, and the same as in the previous question, `0x10010004`. What's less clear is which byte of the contents (666) will be loaded.

The first thing to do is to work out what 666 looks like as a 32-bit quantity: `0x0000029a`. We need to decide whether the byte containing `0x00` or the byte containing `0x9a` is located at `0x10010004`. The answer to this depends on whether the machine is *big-endian* or *little-endian* (i.e., what order we take the bytes from a 4-byte memory word, when we access them byte-at-a-time). On CSE, the byte containing `0x9a` is loaded.

The `lb` instruction has another interesting property: it *sign-extends* the value to 32 bits. This means that it will propagate the high-order bit in the byte all the way to the high-order bit in the 32-bit word, so `$t0` ends up with the value `0xffffff9a`, since the high-order bit of `0x9a` is a one.

d.
```
lw   $t0, aa+4
```
This loads a 32-bit value from the memory location that is 4 bytes beyond the memory location associated with the label `aa`. Since `aa` is associated with `0x10010000`, the address is determined as `0x10010004`, and the value 666 is loaded into `$t0`.

e.
```
la   $t1, cc
lw   $t0, ($t1)
```
The first instruction loads `cc`'s address into register `$t1` (i.e., `0x10010008`). The second instruction then uses the value in `$t1` as an address, and loads the 32-bit quantity from that address. Thus the address is `0x10010008`, and the value loaded into `$t0` is `0x00000001` (i.e. 1).

The two instructions are equivalent to `lw $t0, cc`.

f.
```
la   $t1, cc
lw   $t0, 8($t1)
```
The first instruction loads `cc`'s address into register `$t1` (i.e., `0x10010008`). The second instruction then takes the value in `$t1`, adds 8 to it, and uses the result as the address. So, the address used by the second instruction is

8+0x10010008 = 0x10010010. Thus, the value 5 is loaded into $t0.

g.
```
li   $t1, 8
lw   $t0, cc($t1)
```

The first instruction loads the constant 8 into register `$t1`. The second instruction then takes the address associated with `cc`, adds the contents of `$t1` to it, and uses the result as the address. So, the address used by the second instruction is `0x10010008+8` = `0x10010010` (the same as in the previous question). Thus, the value 5 is loaded into `$t0`.

h.
```
la   $t1, cc
lw   $t0, 2($t1)
```

The first instruction loads `cc`'s address into register `$t1` (i.e. `0x10010008`). The second instruction then takes the value in `$t1`, adds 2 to it, and uses the result as the address. So, the address used by the second instruction is `2+0x10010008` = `0x1001000A`.

However: because this is a `lw` instruction, the address must be 4-byte aligned. Thus, executing this instruction will result in a memory alignment error.

4. What is a breakpoint?

   When is it useful in debugging?

Answer:

Discussed in tute.

5. Translate this C program to MIPS assembler

```
#include <stdio.h>

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

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

Answer:

```
# read 10 numbers into an array
# i in register $t0
main:

    li   $t0, 0         # i = 0
loop0:
    bge  $t0, 10, end0  # while (i < 10) {

    li   $v0, 5         #   scanf("%d", &numbers[i]);
    syscall             #

    mul  $t1, $t0, 4    #   calculate &numbers[i]
    la   $t2, numbers   #
    add  $t3, $t1, $t2  #
    sw   $v0, ($t3)     #   store entered number in array

    addi $t0, $t0, 1    #   i++;
    j    loop0          # }
end0:

    jr   $ra            # return

.data

numbers:
    .word 0 0 0 0 0 0 0 0 0 0  # int numbers[10] = {0};
```

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++;
    }
}
```

Answer:

```
# i in register $t0
main:
    li   $t0, 0         # i = 0
loop1:
    bge  $t0, 10, end1  # while (i < 10) {

    mul  $t1, $t0, 4    #   calculate &numbers[i]
    la   $t2, numbers   #
    add  $t3, $t1, $t2  #
    lw   $a0, ($t3)     #   load numbers[i] into $a0
    li   $v0, 1         #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'      #   printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t0, $t0, 1    #   i++
    j    loop1          # }
end1:

    jr   $ra            # return

.data

numbers:
    .word 0 1 2 3 4 5 6 7 8 9  # int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

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++;
    }
}
```

Answer:

```
# i in register $t0
main:

    li   $t0, 1          # i = 1
loop2:
    bge  $t0, 10, end2   # while (i < 10) {

    mul  $t1, $t0, 4     #
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #   $t3 = &numbers[i]
    lw   $t5, ($t3)      #   $t5 = numbers[i]


    bge  $t5, 0, skip2   #   if (numbers([i]) < 0) {
    addi $t5, $t5, 42    #     numbers[i] += 42
    sw   $t5, ($t3)      #
                         #   }
skip2:
    addi $t0, $t0, 1     #   i++;
    j    loop2           # }
end2:


    jr   $ra            # return

.data

numbers:
    .word 0 1 2 -3 4 -5 6 -7 8 9  # int numbers[10] = {0,1,2,-3,4,-5,6,-7,8,9};
```

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++;
    }
}
```

Answer:

```
# i in register $t0, x in $t4, y in $t8
main:

    # assume there is code here to assign values to the array

    li   $t0, 0          # i = 0
loop2:
    bge  $t0, 5, end2    # while (i < 5) {

    mul  $t1, $t0, 4     #
    la   $t2, numbers    #
    add  $t3, $t1, $t2   #   $t3 = &numbers[i]
    lw   $t4, ($t3)      #   x = numbers[i]

    li   $t5, 9          #   $t5 = 9 - i
    sub  $t5, $t5, $t0   #
    mul  $t6, $t5, 4     #
    add  $t7, $t6, $t2   #   $t7 = &numbers[9 - i]
    lw   $t8, ($t7)      #   y = numbers[9 - i]


    sw   $t8, ($t3)      #   numbers[i] = y
    sw   $t4, ($t7)      #   numbers[9 - i] = x

    addi $t0, $t0, 1     #   i++;
    j    loop2           # }
end2:


    jr   $ra            # return

.data

numbers:
     .word 0 1 2 3 4 5 6 7 8 9  # int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

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

```
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.

Answer:

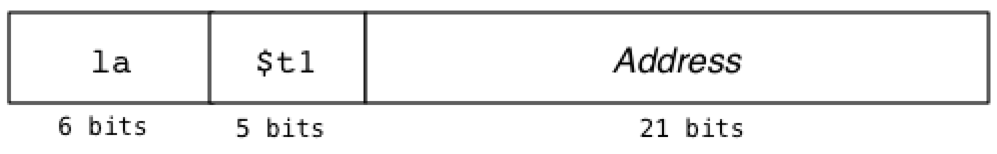
```
   .data
string:
   .asciiz "...."

   .text
   la   $t0, string     # s = &string[0];
   li   $t1, 0
while:
   lb   $t2, ($t0)      # if (*s == 0) goto end_loop
   beq  $t2, $0, end_loop

   addi $t1, $t1, 1     # length++
   addi $t0, $t0, 1     # s++
   j    while           # goto while

   # $t1 contains the length of the string
```

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`?

Answer:

The `la` pseudo-instruction is implemented as two real MIPS instructions:

```
lui  $t1, BaseAddr      # top 16 bits of address
ori  $t1, $t1, Offset   # bottom 16 bits of address
```

... so the address is actually loaded in two sections: the top 16 bits of the register are loaded with the top 16 bits of the base address for the region being referenced; then the bottom 16 bits are loaded with an offset into the region. The assembler splits the address into its two 16-bit components, and places the top 16 bits in the `lui` (*load upper immediate*) instruction, and the bottom 16 bits in the `ori` (*bitwise-or immediate*) instruction.

For example, imagine that we want to load the address of an object located at `0x10010020` in a data region that starts at address `0x10010000`. The first instruction would load `0x1001` into the top 16 bits of `$t1`, giving it a value of `0x10010000`. The second instruction would bitwise-OR this with the offset value i.e. `0x0020`. The final address would thus be produced via

```
Address = 0x10010000 | 0x0020 = 0x10010020
```

The same technique is used for the `li` (*load immediate*) instruction, which loads a 32-bit constant value into a specified register.

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.

Answer:

```
   .data
x: .space 8
y: .space 4
   .text

   li   $v0, 5
   syscall
   sw   $v0, y

   lw   $t0, y
   addi $t0, $t0, 2000
   lw   $t1, y
   addi $t1, $t1, 3000
   mult $t0, $t1      # (Hi,Lo) = $t0 * $t1
   mfhi $t0
   sw   $t0, x        # top 32 bits of product
   mflo $t0
   sw   $t0, x+4      # bottom 32 bits of product
```

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`).

Answer:

A version that aims to minimise instructions (using more registers):

```
lw   $t0, x
lw   $t1, y
mul  $t2, $t0, $t0
mul  $t3, $t1, $t1
add  $t4, $t2, $t3
mul  $t5, $t0, $t1
sub  $t5, $t4, $t5
lw   $t4, z
mul  $v0, $t5, $t4
```

A version that aims to minimise registers used (using more instructions):

```
lw   $t0, x
lw   $t1, y
mul  $t0, $t0, $t0
mul  $t1, $t1, $t1
add  $t0, $t0, $t1
lw   $t1, x
lw   $t2, y
mul  $t1, $t1, $t2
sub  $t0, $t0, $t1
lw   $t1, z
mul  $v0, $t0, $t1
```

An even better solution from John Luo (minimising instructions and registers):

```
lw  $t0, x           # t0 = x
lw  $v0, y           # v0 = y
mul $t1, $t0, $t0    # t1 = x * x
mul $t0, $t0, $v0    # t0 = x * y
mul $v0, $v0, $v0    # v0 = y * y
add $v0, $t1, $v0    # v0 = (x * x) + (y * y)
sub $v0, $v0, $t0    # v0 = ((x * x) + (y * y)) - (x * y)
lw  $t0, z           # t0 = z
mul $v0, $v0, $t0    # v0 = (((x * x) + (y * y)) - (x * y)) * z
```

And another excellent solution from Jacob Mikkelsen (also minimising instructions and registers):

```
# Rewrite as (x(x - y) + y^2) * z
lw  $t0, x           # x
lw  $t1, y           # y
sub $v0, $t0, $t1    # x - y
mul $v0, $v0, $t0    # x(x - y) = x^2 - xy
mul $t1, $t1, $t1    # y^2
add $v0, $v0, $t1    # x^2 + y^2 - xy
lw  $t0, z           # z
mul $v0, $v0, $t0    # (x^2 + y^2 - xy) * z
```