

Inf2C Computer Systems

Tutorial 2, Week 5

Solutions

Boris Grot, Stratis Viglas, Aris Efthymiou, Salman Khan

1. **Memory copy function.** Write a function in MIPS assembly that will perform a copy of a block of given words from one memory location to another. The function input parameters are the initial (lowest) source address, the initial target address and the number of words to copy.

Answer *The possibility of overlapping blocks makes this harder than it might first appear. It's best to write a simple version first, ignoring this corner-case, and then modify this simple version to cover the overlapping blocks case.*

Here is a sample for this first simple version. We assume \$a0 is the source address, \$a1 is the destination address, and \$a2 is the length in words.

```
        sll      $a2, $a2, 2      # multiply by 4 to convert
                                   # length to bytes
        add      $t1, $a0, $a2    # address of the end of the block
loop:   slt      $t0, $a0, $t1    # t0 is 1 if there is still copying
                                   # to be done
        beq      $t0, $zero, done
        lw       $t2, 0($a0)      # read from source
        sw       $t2, 0($a1)      # write to destination
        addi     $a0, $a0, 4       # increment source address
        addi     $a1, $a1, 4       # increment destination address
        j        loop
done:
```

Here, the addresses are compared rather than incrementing the index variable and comparing it with the length. This is faster, but feel free to use the other method.

This is what is needed to solve the overlapping problem: If the source address is lower than the target address, you can safely copy words from

the highest address down to the lowest. If the source address is higher than the destination address, then it's safe to copy from the lowest address up to the highest.

```

sll    $a2, $a2, 2    # convert length to bytes
slt    $t0, $a0, $a1  # $t0 is 1 if source lower
                        # than destination
beq     $t0, $zero, src_high  # source lower, so
                                # copy from high
                                # address down
add     $t1, $a0, $a2  # point to last word +1
addi    $t1, $t1, -4   # t1 is last word of source
add     $t2, $a1, $a2
addi    $t2, $t2, -4   # t2 is last word of destination
addi    $t3, $a0, -4   # address to stop looping
li      $t4, -4        # will use t4 to increment.
                        # Negative since we're moving down
j       loop
src_high:
add     $t1, $a0, $zero
add     $t2, $a1, $zero
add     $t3, $a0, $a2  # address to stop looping
li      $t4, 4         # positive increment in this case
loop:
beq     $t3, $t1, end
lw      $t5, 0($t1)
sw      $t5, 0($t2)
add     $t1, $t1, $t4
add     $t2, $t2, $t4
j       loop
end:

```

Again, you can start with a solution where there are two separate loops, one for each direction of memory traversal and improve it by observing they can be joined into one.

2. **Memory copy function refinement.** Suppose we want to change the granularity of the memory copy function from words to bytes. How can the above program be converted to do this efficiently? Note that a load or store word (4 bytes) takes one cycle, as does loading or storing a byte.

Answer The simple solution here is use the code above with the only change that we use the load/store byte instructions instead of word instructions and the pointers are incremented by 1 instead of 4.

However, this would be very slow if large chunks of memory are to be copied. It is worth investigating if we can use word load/store as much as possible and only copy bytes at the two edges of the arrays if needed.

Unfortunately, this only works when the source and destination addresses are either both aligned or misaligned in the same way.

3. C and fun with pointers.

- (a) What will the following piece of C code do and why?

```
int *a = 10; *a = 100;
```

Answer *It will crash. The reason is that the statement `int *a = 10;` assigns a value to the pointer, making it point to address `0x0000000A` in memory. However, this memory has not been allocated and is not accessible by the user's program. Then, statement `*a = 100;` tries to change that piece of memory and give it a value of 100, which results in the program crashing.*

- (b) If there is something wrong with the previous C code, can you fix it?

Answer *Basically make the pointer point to something on the heap, either through `malloc()`, or by using an additional integer variable and assigning its address to `a`, i.e., either:*

```
int *a = (int *) malloc(sizeof(int));
```

or

```
int x; int *a = &x;
```

A comment on the return type of `malloc`. It is the special type of generic pointers `void`. C allows casting between this type and any other pointer type. Indeed, the explicit cast is optional – C will insert the appropriate type conversion if it is left out. That is, the following two statements are equivalent:*

```
int *a = (int *) malloc(sizeof(int));
```

```
int *a = malloc(sizeof(int));
```

- (c) Given the following declaration:

```
int **array;
```

allocate a triangular array of `n` rows. That is, row 0 should have 1 column, row 1 should have 2 columns and so on. Each cell of a column should have an initial value equal to the current row, i.e., `array[0][0] == 0`, `array[1][0] == array[1][1] == 1` and so on.

Answer *Here's a sample implementation:*

```
int n;
```

```
int **array;
```

```
n = ...;
```

```
int i, j;
```

```
array = (int **) malloc(n*sizeof(int *));
```

```
// iterate over rows
```

```
for (i = 0; i < n; i++) {
```

```
    array[i] = (int *) malloc((i+1)*sizeof(int));
```

```

    for (j = 0; j <= i; j++)
        array[i][j] = i; // or *((array+i)+j)
}

```

The first `malloc()` call allocates as many pointers to integers as there are rows in the array, the second one allocates as many integers as there are columns in each row.

- (d) How would you de-allocate the array you allocated in the previous question?

Answer *Memory is de-allocated in inverse allocation order:*

```

for (i = 0; i < n; i++)
    free(array[i]); // de-allocate each row
free(array);       // de-allocate the table

```