Week 07 Tutorial Solutions

1. How is the assignment going?

   Does anyone have hints or advice for other students?

> **Answer:**
>
> Discussed in tutorial.

2. Give MIPS directives to represent the following variables:

   a. `int v0;`
   b. `int v1 = 42;`
   c. `char v2;`
   d. `char v3 = 'a';`
   e. `double v4;`
   f. `int v5[20];`
   g. `int v6[10][5];`
   h. `struct { int x; int y; } v7;`
   i. `struct { int x; int y; } v8[4];`
   j. `struct { int x; int y; } *v9[4];`

   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:**
>
> a. `v0: .space 4`
> b. `v1: .word 42`
> c. `v2: .space 1`
> d. `v3: .byte 'a';`
> e. `v4: .space 8`
> f. `v5: .space 80` (20 * 4-byte ints)
> g. `v6: .space 200` (10 * 5 * 4-byte ints)
> h. `v7: .space 8` (struct is 2 * 4-byte ints)
> i. `v8: .space 32` (4 * 8-byte structs)
> j. `v9: .space 16` (4 * 4-byte pointers)

3. Translate this C program to MIPS assembler.

```c
int max(int a[], int length) {
    int first_element = a[0];
    if (length == 1) {
        return first_element;
    } else {
        // find max value in rest of array
        int max_so_far = max(&a[1], length - 1);
        if (first_element > max_so_far) {
            max_so_far = first_element;
        }
        return max_so_far;
    }
}
```

> **Answer:**
>
> Equivalent MIPS assembler translated directly from C:

```
# Recursive maximum of array function

# Register usage:
#   - `a' is in $a0
#   - `length' is in $a1
#   - `first_element' is in $s0
#   - `max_so_far' is in $t0

# s0 & s1 used for a and first_element because they need
# to keep their value across recursive call

max:
    # prologue
    addi  $sp, $sp, -8
    sw    $ra, 0($sp)
    sw    $s0, 4($sp)

    lw    $s0, ($a0)     # first_element = a[0]

    bne   $a1, 1, end1   # if (length == 1)
    move  $v0, $s0       #   return first_element;
    j     max_return
end1:                    # }

    addi  $a0, $a0, 4    # calculate &a[1]
    addi  $a1, $a1, -1   # length - 1
    jal   max            #
    move  $t0, $v0       # max_so_far = max(&a[1], length - 1);

    ble   $s0, $t0, end2 # if (first_element > max_so_far) {
    move  $t0, $s0
end2:

    move  $v0, $t0       # return max_so_far

max_return:
    # epilogue
    lw    $ra, 0($sp)
    lw    $s0, 4($sp)
    addi  $sp, $sp, 8
    jr    $ra



# some test code which calls max
main:
    addi $sp, $sp, -4  # create stack frame
    sw   $ra, 0($sp)   # save return address

    la   $a0, array
    li   $a1, 10
    jal   max    #

    move  $a0, $v0     # printf ("%d")
    li    $v0, 1
    syscall

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

                       # clean up stack frame
    lw   $ra, 0($sp)   # restore $ra
    addi $sp, $sp, 4   # restore sp

    li  $v0, 0         # return 0
    jr  $ra


.data
```

```
array:
    .word 1 2 3 4 5 6 4 3 2 1
```

Alternative more complex solution using a frame pointer

```
# Recursive maximum of array function

# version with a frame pointer

# Register usage:
#   - `a' is in $s0
#   - `a[0]' is in $s1
#   - `length' is in $s2
#   - `max_so_far' is in $t0

max:
    # prologue
    sw    $fp, -4($sp)
    la    $fp, -4($sp)
    sw    $ra, -8($sp)
    sw    $s0, -12($sp)
    sw    $s1, -16($sp)
    sw    $s2, -20($sp)
    addi  $sp, $sp, -24

    lw    $s1, ($a0)
    move  $s2, $a1

    # base case
max_base_case:
    # if (length == 1) return a[0];
    li    $t0, 1
    bgt   $a1, $t0, max_rec_case
    lw    $v0, ($a0)
    j     max_return

    # recursive case
max_rec_case:
    # int max_so_far = max(&a[1], length-1);
    addi  $a0, $a0, 4
    addi  $a1, $a1, -1
    jal   max
    move  $t0, $v0
    # return (a[0] < max_so_far) ? a[0] : max_so_far;
    ble   $s1, $t0, max_else
    move  $v0, $s1
    j     max_return
max_else:
    move  $v0, $t0
max_return:
    # epilogue
    lw    $s2, -16($fp)
    lw    $s1, -12($fp)
    lw    $s0, -8($fp)
    lw    $ra, -4($fp)
    la    $sp, 4($fp)
    lw    $fp, ($fp)
    jr    $ra


# some test code which calls max
main:
    addi $sp, $sp, -4  # create stack frame
    sw   $ra, 0($sp)   # save return address

    la   $a0, array
    li   $a1, 10
    jal   max    #

    move  $a0, $v0     # printf ("%d")
    li    $v0, 1
    syscall

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

```
                    # clean up stack frame
    lw   $ra, 0($sp)   # restore $ra
    addi $sp, $sp, 4   # restore sp

    li  $v0, 0         # return 0
    jr  $ra


.data
array:
    .word 1 2 3 4 5 6 4 3 2 1
```

4. Translate this C program to MIPS assembler.

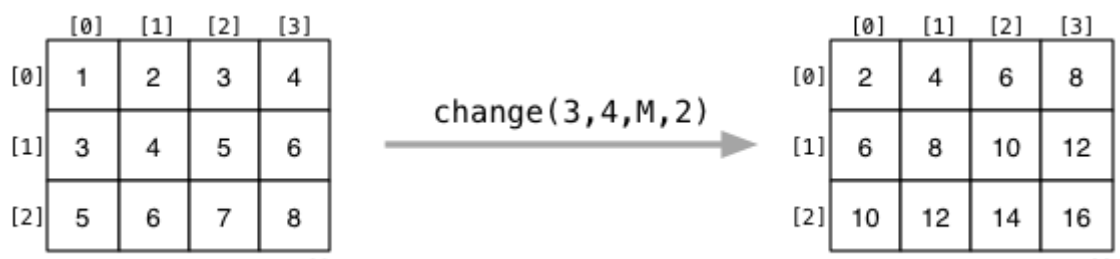
```c
#include <stdio.h>

char flag[6][12] = {
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'}
};

int main(void) {
    for (int row = 0; row < 6; row++) {
        for (int col = 0; col < 12; col++)
            printf ("%c", flag[row][col]);
        printf ("\n");
    }

}
```

Answer:

Equivalent MIPS assembler translated directly from C:

```
# row in $t0, col in $t1
main:
    li   $t0, 0          # for (int row = 0; row < 6; row++) {

loop1:
    bge  $t0, 6, end1
    li   $t1, 0          # for (int col = 0; col < 12; col++) {

loop2:
   beq  $t1, 12, end2    # printf("%c", flag[row][col])
   mul  $t2, $t0, 12     # convert [row][col] to byte offset
   add  $t3, $t2, $t1
   la   $t4, flag
   add  $t5, $t3, $t4
   lb   $a0, ($t5)
   li   $v0, 11
   syscall

   addi $t1, $t1, 1    # col++
   j    loop2

end2:
   li   $a0, '\n'       # printf("%c", '\n')
   li   $v0, 11
   syscall
   addi $t0, $t0, 1    # row++
   j    loop1
end1:

    li  $v0, 0          # return 0
    jr  $ra

.data
flag:
    .byte '#' '#' '#' '#' '#' '.' '.' '#' '#' '#' '#' '#'
    .byte '#' '#' '#' '#' '#' '.' '.' '#' '#' '#' '#' '#'
    .byte '.' '.' '.' '.' '.' '.' '.' '.' '.' '.' '.' '.'
    .byte '.' '.' '.' '.' '.' '.' '.' '.' '.' '.' '.' '.'
    .byte '#' '#' '#' '#' '#' '.' '.' '#' '#' '#' '#' '#'
    .byte '#' '#' '#' '#' '#' '.' '.' '#' '#' '#' '#' '#'
```

5. Consider the following operation that multiplies all of the elements in a matrix by a constant factor:



This operation could be rendered in C99-standard C as

```
void change (int nrows, int ncols, int M[nrows][ncols], int factor)
{
    for (int row = 0; row < nrows; row++) {
        for (int col = 0; col < ncols; col++) {
            M[row][col] = factor * M[row][col];
        }
    }
}
```

Write a function in MIPS assembly equivalent to the above C code. Assume that the arguments are placed in the `$a?` registers in the order given in the function definition. e.g., the function could be called as follows in MIPS:

```
   li   $a0, 3
   li   $a1, 4
   la   $a2, M
   li   $a3, 2
   jal  change
```

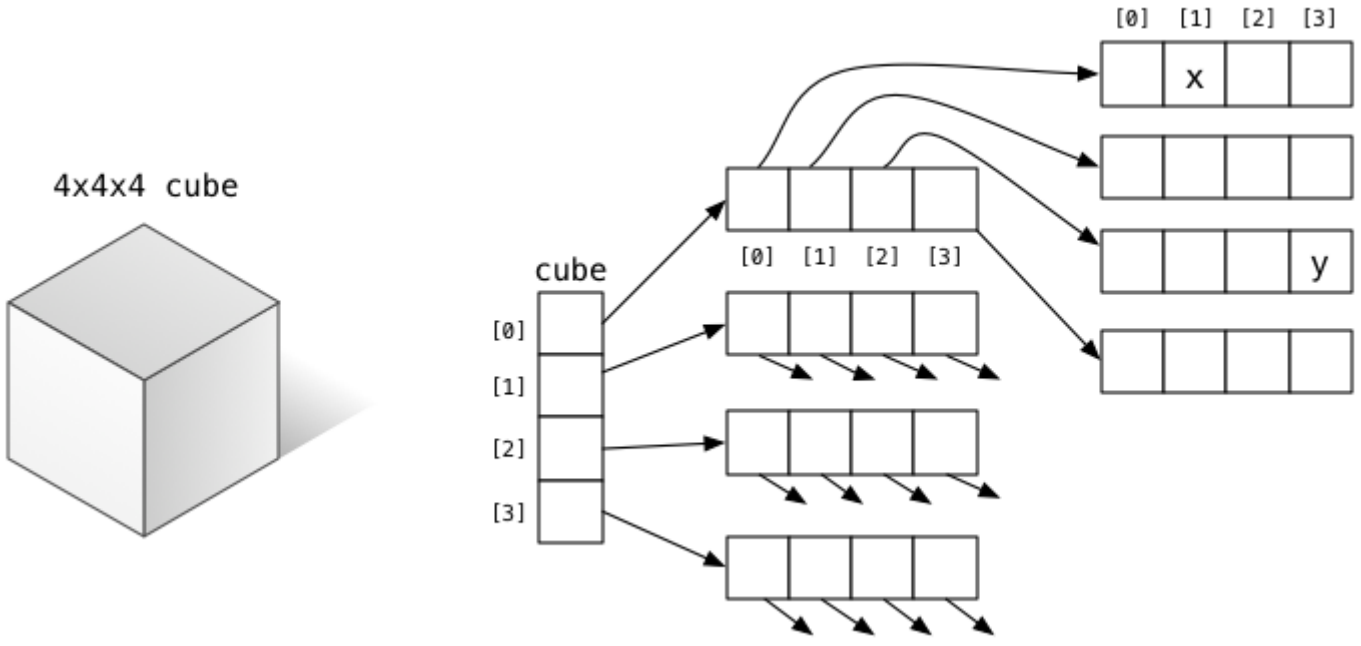
Where `M` is defined as:

```
    .data
M:  .word 1, 2, 3, 4
    .word 3, 4, 5, 6
    .word 5, 6, 7, 8
```

Answer:

```
change:
   # params:    nrows is $a0, ncols is $a1, &M is $a2, factor is $a3
   # registers: row is $s0, col is $s1

   # set up stack frame
   sw   $fp, -4($sp)
   la   $fp, -4($sp)
   sw   $ra, -4($fp)
   sw   $s0, -8($fp)
   sw   $s1, -12($fp)
   addi $sp, $sp, -16

   li   $t4, 4          # sizeof(int)
   li   $s0, 0
loop1:                  # for (int row = 0; row < nrows; row++)
   bge  $s0, $a0, end1
   li   $s1, 0
loop2:                  # for (int col = 0; col < ncols; col++)
   bge  $s1, $a1, end2

   # get &M[row][col]
   mul  $t0, $s0, $a1
   mul  $t0, $t0, $t4   # offset of M[row]  (#bytes)
   mul  $t1, $s1, $t4   # offset within M[row]  (#bytes)
   add  $t0, $t0, $t1   # offset of M[row][col] (#bytes)
   add  $t0, $t0, $a2   # &M[row][col]

   lw   $t1, ($t0)
   mul  $t1, $t1, $a3
   sw   $t1, ($t0)      # M[row][col] = factor * M[row][col]

   addi $s1, $s1, 1
   j    loop2
end2:
   addi $s0, $s0, 1
   j    loop1
end1:

   # clean up stack frame
   lw   $ra, -4($fp)
   lw   $s0, -8($fp)
   lw   $s1, -12($fp)
   la   $sp, 4($fp)
   lw   $fp, ($fp)
```

6. Consider the following 3-d array structure:



The cube could be implemented as an array of pointers, each of which points to a slice of the cube. Each slice of the cube is also an array of pointers, and each of those points to an array of `int` values.

For example, in the diagram above, the cell labelled `x` can be accessed as `cube[0][0][1]`, and the cell labelled `y` can be accessed as `cube[0][2][3]`.

a. Write MIPS assembler directives to define a data structure like this.
b. Write MIPS assembler code to scan this cube, and count the number of elements which are zero.

In other words, implement the following C code in MIPS.

```
int cube[4][4][4];

int nzeroes = 0;
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
        for (int k = 0; k < 4; k++)
            if (cube[i][j][k] == 0)
                nzeroes++;
```

Answer:

```
# possible definition for the data structure
# assumes that SPIM allows forward references to labels
        .data
cube:   .word slice0, slice1, slice2, slice3
slice0: .word row01, row02, row03, row04
slice1: .word row11, row12, row13, row14
slice2: .word row21, row22, row23, row24
slice3: .word row31, row32, row33, row34
row01:  .space 16
row02:  .space 16
row03:  .space 16
row04:  .space 16
# etc. etc.

# assume that ...
#   $s0 represents nzeroes
#   $s1 is i, $s2 is j, $s3 is k
#   $s4 holds the dimension (4)
   li   $s0, 0
   li   $s4, 4
   li   $s1, 0
loop1:
   beq  $s1, $s4, end_loop1
   li   $s2, 0
loop2:
   beq  $s2, $s4, end_loop2
   li   $s3, 0
loop3:
   beq  $s3, $s4, end_loop3
   move $t0, $s1
   mul  $t0, $t0, $s4
   lw   $t0, cube($t0)
   move $t1, $s2
   mul  $t1, $t1, $s4
   add  $t0, $t0, $t1
   move $t1, $s3
   mul  $t1, $s3, $s4
   add  $t0, $t0, $t1
   lw   $t0, ($t0)
   bne  $t0, $0, skip
   addi $s0, $s0, 1
skip:
   addi $s3, $s3, 1
   j    loop3
end_loop3:
   addi $s2, $s2, 1
   j    loop2
end_loop2:
   addi $s1, $s1, 1
   j    loop1
end_loop1:
```

7. For each of the following `struct` definitions, what are the likely offset values for each field, and the total size of the `struct`:

a.
```
struct _coord {
    double x;
    double y;
};
```

b.
```
typedef struct _node Node;
struct _node {
    int value;
    Node *next;
};
```

c.
```
struct _enrolment {
    int stu_id;         // e.g. 5012345
    char course[9]:     // e.g. "COMP1521"
    char term[5];       // e.g. "17s2"
    char grade[3];      // e.g. "HD"
    double mark;        // e.g. 87.3
};
```

d.
```
struct _queue {
    int nitems;     // # items currently in queue
    int head;       // index of oldest item added
    int tail;       // index of most recent item added
    int maxitems;   // size of array
    Item *items;    // malloc'd array of Items
};
```

Both the offsets and sizes should be in units of number of bytes.

Answer:

a. `x` is the first field, so has offset 0. Since `x` is 8 bytes long, `y` will have offset 8. The `struct` holds two 8-byte fields, both of which are correctly aligned, so no padding is required, and so the overall size of the `struct` is 16 bytes.

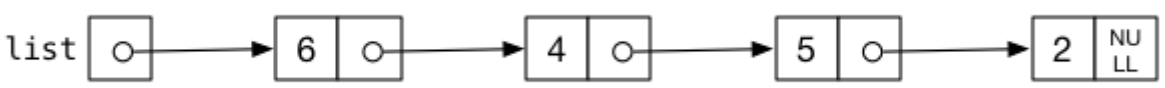
b. `value` is the first field, so has offset 0. Since `value` is 4 bytes long, `next` will have offset 4. Since pointers are also 4 bytes long, and no padding is required, the overall size of the `struct` is 8 bytes.

c. `stu_id` is the first field, so has offset 0. Since `stu_id` is 4 bytes long, `course` has offset 4. `course` is 9 bytes long, but since the following field only needs byte alignment, `term` has offset 13. Similarly, `grade` has offset 18. The `mark` field requires word alignment, and so it needs some padding to move the offset from 21 to 24. Taking into account that `double` values are stored in 8 bytes, the overall size of the `struct` is 32 bytes.

d. `nitems` is the first field, so has offset 0. `head` has offset 4, `tail` has offset 8, `maxitems` has offset 12, and `items` has offset 16. It doesn't matter what the size of an `Item` is, or how any `Items` there are, since they are stored outside the `struct`. Since the pointer is 4 bytes long, and all the values are word-aligned, the overall size of the `struct` is 20 bytes.

8. **Challenge exercise, for those who have seen linked data structures in C.**

Consider a linked list



which could be defined in MIPS as:

```
    .data
list:   .word node1
node1:  .word 6, node2
node2:  .word 4, node3
node3:  .word 5, node4
node4:  .word 2, 0
```

Write a MIPS function that takes a pointer to the first node in the list as its argument, and returns the maximum of all of the values in the list. In other words, write a MIPS version of this C function:

```
typedef struct _node Node;
struct _node { int value; Node *next; };

int max (Node *list)
{
    if (list == NULL) return -1;
    int max = list->value;
    Node *curr = list;
    while (curr != NULL) {
        if (curr->value > max)
            max = curr->value;
        curr = curr->next;
    }
    return max;
}
```

You can assume that only positive data values are stored in the list.

Answer:

Using the standard function prologue and epilogue:

```
max:
   # prologue
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   la   $fp, ($sp)
   addi $sp, $sp, -4
   sw   $ra, ($sp)
   # body
   #  if (list == NULL) return -1;
   #  int max = list->value;
   #  Node *curr = list;
   #  while (curr != NULL) {
   #     if (curr->value > max) max = curr->value;
   #     curr = curr->next;
   #  }
   #  return max;
   bne  $a0, $0, max1
   li   $v0, -1
   j    end_max
max1:
   move $s0, $a0       # use $s0 for curr
   lw   $v0, 0($s0)    # use $v0 for max
loop:
   beq  $s0, $0, end_max
   lw   $t0, 0($s0)    # $t0 = curr->value
   ble  $t0, $v0, max2:
   move $v0, $t0       # max = curr->value
max2:
   lw   $s0, 4($s0)    # curr = curr->next
   j    loop
end_max:
   # epilogue
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra
```

9. If we execute the following small MIPS program:

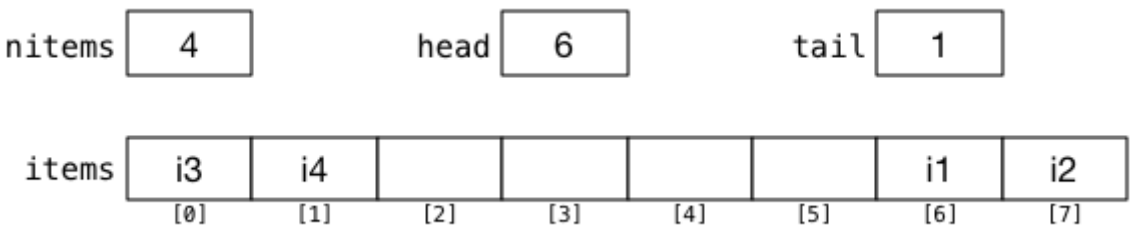
```
   .data
x: .space 4
   .text
   .globl main
main:
   li  $a0, 32768
   li  $v0, 1
   syscall
   sw  $a0, x
   lh  $a0, x
   li  $v0, 1
   syscall
   jr  $ra
```

... we observed that the first `syscall` displays `32768`, but the second `syscall` displays `-32768`. Why does this happen?

Answer:

Initially, the value `32768` is loaded into `$a0`, printed, and then stored in the 4-byte object called `x`. Then a 2-byte quantity is loaded from `x` into `$a0`. The hexadecimal representation of the value stored in `x` is `0x00008000`. When the two bytes are loaded the value `0x8000` is used. Since `lh` deals with signed quantities, and since the first bit of `0x8000` is a 1, it ends up being interpreted as a negative value by "sign extension". This means copying the top bit into all 16 bits of the 32-bit register, giving `0xFFFF8000`, which is the 32-bit two's-complement representation of the number `-32768`.

10. FIFO queues can be implemented using circular arrays. For example:



And the C code to manipulate such a structure could be:

```c
// place item at the tail of the queue
// if queue is full, returns -1; otherwise returns 0
int enqueue (int item)
{
    if (nitems == 8) return -1;
    if (nitems  > 0) tail = (tail + 1) % 8;
    queue[tail] = item;
    nitems++;
    return 0;
}

// remove item from head of queue
// if queue is empty, returns -1; otherwise returns removed value
int dequeue (void)
{
    if (nitems == 0) return -1;
    int res = queue[head];
    if (nitems  > 1) head = (head + 1) % 8;
    nitems--;
    return res;
}
```

Assuming that the items in the queue are `int`s, and the following MIPS data definitions are used:

```
nitems: .word 0
head:   .word 0
tail:   .word 0
items:  .space 32
```

... implement the `enqueue()` and `dequeue()` functions in MIPS.

Use one of the standard function prologues and epilogues from lectures.

Answer:

```
enqueue:
   # prologue
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   la   $fp, ($sp)
   addi $sp, $sp, -4
   sw   $ra, ($sp)
   # body
   #  if (nitems == 8) return -1;
   #  if (nitems > 0) tail = (tail+1) % 8;
   #  queue[tail] = item;
   #  nitems++;
   #  return 0;
   li   $t0, 8
   lw   $t1, nitems
   bne  $t0, $t1, enq1
   li   $v0, -1
   j    end_enq
enq1:
   lw   $t2, tail
   beq  $t1, $0, enq2
   addi $t2, $t2, $1
   rem  $t2, $t2, $t0
   sw   $t2, tail
enq2:
   mul  $t3, $t2, 4
   sw   $a0, queue($t3)
   lw   $t2, nitems
   addi $t2, $t2, 1
   sw   $t2, nitems
   li   $v0, 0
end_enq:
   # epilogue
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra

dequeue:
   # prologue
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   la   $fp, ($sp)
   addi $sp, $sp, -4
   sw   $ra, ($sp)
   # body
   #  if (nitems == 0) return -1;
   #  int res = queue[head];
   #  if (nitems > 1) head = (head+1) % 8;
   #  nitems--;
   #  return res;
   li   $t0, 8
   lw   $t1, nitems
   bne  $t0, $0, deq1
   li   $v0, -1
   j    end_deq
deq1:
   lw   $t2, head           # res = queue[head]
   mul  $t3, $t2, 4
   lw   $v0, queue($t3)
   li   $t3, 1
   beq  $t2, $t3, deq2      # if (nitems == 1) skip
   add  $t2, $t2, 1         # head = (head+1) % 8
   rem  $t2, $t2, $t0
   sw   $t2, head
deq2:
   lw   $t2, nitems         # nitems--
   addi $t2, $t2, -1
   sw   $t2, nitems
end_deq:
```

```
# epilogue
lw   $ra, ($sp)
addi $sp, $sp, 4
lw   $fp, ($sp)
addi $sp, $sp, 4
jr   $ra
```

COMP1521 20T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G