# Lab 9: Linking, I/O, and Caches
## Due Date: Monday 4/24/2017 11:59PM

This lab covers material on linking, loading, I/O, and caches (lectures 20–23). There are 100 points total.

## Written Problems (*100 points*)

1. Briefly explain the role of a linker.

2. For the following code, draw a symbol table that the assembler would construct and fill it in:

```
            .ORIG x3000
            LD R0, zero
            LD R1, number
    loop    BRz end
            ADD R0, R0, 1
            ADD R1, R1, -1
            BRnzp loop
    end     LEA R0, endstr
            PUTS
            HALT

    zero    .FILL x0
    number  .FILL x1000
    endstr  .STRINGZ "I'm Done"
```

3. Create a program on fourier named `test.c` and fill it with the following:

```
        #include <stdlib.h>

        extern int foo();

        int main () {
            foo();
```

```
            return 0;
        }
```

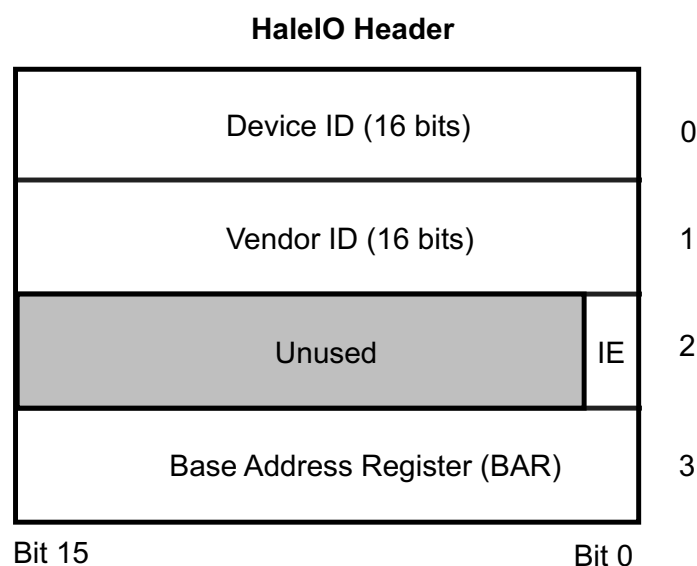Then run the following command:

```
$> gcc -o test test.c
```

What is the output of compiling?

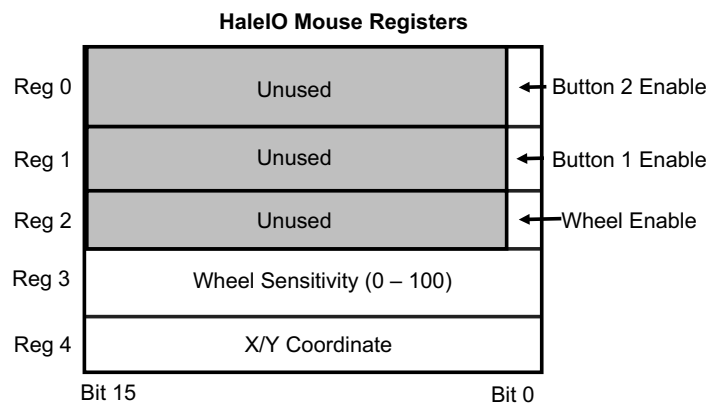Now run the following:

```
$> gcc -c test.c -o test.o
```

Now what is the output? Why is it different?

4. I decide to build a new I/O subsystem for LC-3. I'm going to design an I/O *bus* that can connect up to 256 different devices, each housed in a *slot*. I call it the *HaleIO Bus*. In this scheme, every device must expose a standard header that describes its capabilities and its accessible registers. This is called the *HaleIO configuration space*. The configuration space is simply an array of 16-bit values (shown below).

**HaleIO Header**

| Device ID (16 bits) | | 0 |
| Vendor ID (16 bits) | | 1 |
| Unused | IE | 2 |
| Base Address Register (BAR) | | 3 |

Bit 15                                                          Bit 0

Every device must specify its ID, its Vendor ID, an interrupt enable bit, and a base address register (BAR) in its config space. The interrupt enable bit (`IE`) allows the device to generate interrupts to the CPU. The purpose of the BAR is to tell software where the device's memory mapped registers are in memory. In other words, it contains an MMIO address. This is the *base* address at which software can read and write the device's registers. For example, if I have a device that has 2 registers and with a BAR value of `xF778`, that means I can access the device's first register at `xF778` and its second register at `xF779`.

To allow software to read and write the config space of any device, I expose three new MMIO registers on the LC-3. The first is the device number register, DEVNUM at address xFCF8. Valid device numbers range from 0-255 (these correspond to slots on the bus). The next is the offset register, DEVOFF at address xFCF9. The final register is the data register, DEVDATA at xFCFA. To read the configuration space of a device, software must put the device number in DEVNUM and specify which config space offset it wants to read in DEVOFF (in that order). The value at that offset will then be loaded by the hadrware into DEVDATA. To write a config space offset, I store a value into DEVDATA first, then specify the device number and offset the same way. The HaleIO bus specification states that reads to the config space of a slot that has no device attached will return all 0s.

**HaleIO Mouse Registers**

| | | |
|---|---|---|
| Reg 0 | Unused | ← Button 2 Enable |
| Reg 1 | Unused | ← Button 1 Enable |
| Reg 2 | Unused | ← Wheel Enable |
| Reg 3 | Wheel Sensitivity (0 – 100) | |
| Reg 4 | X/Y Coordinate | |

Bit 15                                    Bit 0

I designed my first HaleIO-compliant device, a mouse. The mouse device ID is xF00D. You don't know which HaleIO slot (dev number) it's in. Write an LC-3 program that probes the HaleIO bus for the mouse. If it doesn't find it, your program will print an error using PUTS. If it does find it, you should use the mouse's MMIO registers (shown above) to enable all the buttons and the wheel, and set the wheel sensitivity to 55.

5. I decide to add a cache to the LC-3. It will be a direct-mapped, 1KB cache and will have a cache line size of 32 bytes.

   a) How many words will be stored in a single cache line?

   b) How many entries (cache lines) are in the cache?

   c) Assuming the standard LC-3 address width (16-bits), how many bits of the address must we use for the line offset (block offset), the cache index, and the tag, respectively?

   d) For the following piece of C code, what will the cache miss rate be with this cache? Recall that miss rate is the number of cache misses divided by the total number of memory references.

```
int i;
int a[512]; // ints are 16 bits on LC-3
for (i = 0; i < 512; i++) {
    int x;
    x = a[i];
}
```

e) I need to write a super fast linear algebra package for manipulating matrices, e.g. for engineering simulations. In C, we can represent matrices as multi-dimensional arrays. The C language says that such arrays will be laid out in memory in *row-major* order. This means, e.g. for the following matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

in C I can access $a_{23}$ with a[2][3] and my matrix will be laid out in memory like so:

| address | value |
|---------|-------|
| 0 | $a_{11}$ |
| 1 | $a_{12}$ |
| 2 | $a_{13}$ |
| 3 | $a_{21}$ |
| 4 | $a_{22}$ |
| 5 | $a_{23}$ |

I'm dealing with $512 \times 16$ matrices and I'm trying to decide how I'll access them. The first option:

```
int i,j;
int a[512][16];
for (i = 0; i < 512; i++) {
    for (j = 0; j < 16; j++) {
        int x;
        x = a[i][j];
    }
}
```

The second option:

```
int i,j;
int a[512][16];
for (i = 0; i < 16; i++) {
    for (j = 0; j < 512; j++) {
        int x;
        x = a[j][i]; // note the difference
    }
}
```

Which option is better and why?

f) Memory speeds have historicaly not at all kept up with CPU speeds. What effect will that have on which choice you make above? (Remember the cache lives on the CPU).

4

# Hand-in Instructions

**Make sure to put your name on your submission. Submissions without names will be given zero points! For code, this means put a comment at the top of your code file(s) with your name on it.**

**Physical** : If you're submitting a written copy, hand it to one of the TAs or to the instructor. You can also leave it in the instructor's mailbox in the CS department office, but make sure to get it time stamped when you do (see the "Submitting Work" section of the syllabus).

**Digital** : If you would like to submit an electronic copy, note that I will only accept PDF files (no Word docs please). Again, see the "Submitting Work" section of the syllabus. Please do not take a poorly lit picture of your assignment. Your grade will suffer commensurately with our inability to read your work. Once you have a PDF, you should submit it on `fourier`. You should name your file `yourid-lab9.pdf` where `yourid` is the thing in front of the @hawk.iit.edu in your e-mail address.

You can first get your PDF (for example, for me it might be called `kh123-lab9.pdf`) onto `fourier` like so:

```
[me@mylocalmachine]$ scp kh123-lab9.pdf kh123@fourier.cs.iit.edu:
```

Then you can login to `fourier` via ssh and submit it:

```
[kh123@fourier]$ cp kh123-lab9.pdf /home/khale/HANDIN/lab9
```

# Late handins

If you're turning in your assignment late digitally, you'll need to e-mail me your PDF file directly.