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.

   The linker resolves references to symbols defined in external files

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"
```

Symbol Name	*Address*
loop	*x3002*
end	*x3006*
zero	*x3009*
number	*x300A*
endstr	*x300B*

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?

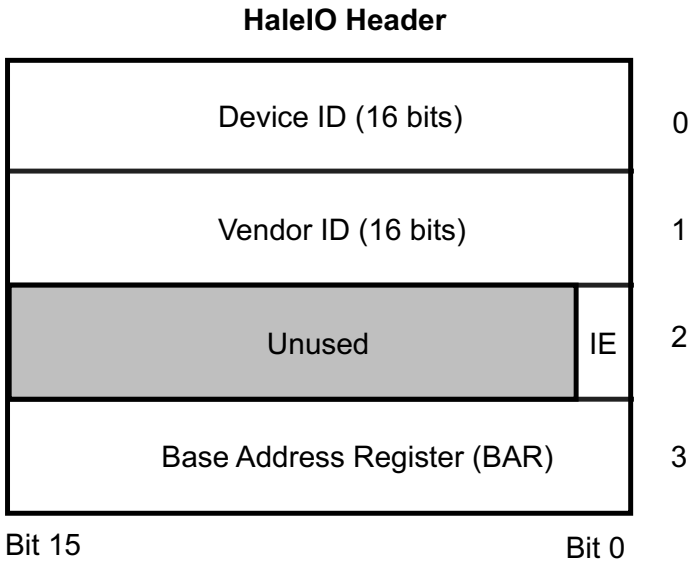
Now run the following:

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

Now what is the output? Why is it different?

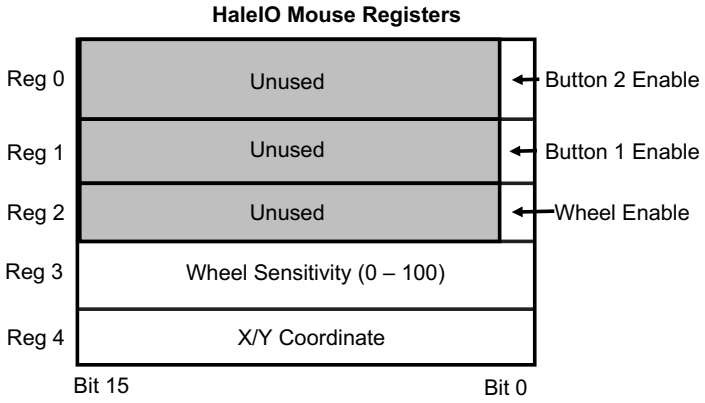
The first command produces an error because the linker cannot find foo. The second command succeeds because it doesn't actually invoke the linker! It just produces an object file with unresolved references (this is not an executable file).

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).



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.



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.

```
            LD R0, MAX_DEVS
            AND R2, R2, 0
            LD R1, MOUSE_ID
try_again   BRn nope
            ST R0, DEVNUM
            ST R2, DEVOFF
            LD R3, DEVDATA
            LD R4, MOUSE_ID_COMP
            ADD R5, R3, R4   ; if it's the right ID, should b
            BRz found
            ADD R0, R0, -1
            BRnzp try_again

nope        LEA R0, NOT_FOUND
            PUTS
            HALT

found       LD R2, BAR_OFFSET   ; config space offset 3 (BAR)
            ST R0, DEVNUM
            ST R2, DEVOFF
            LD R3, DEVDATA ; get the BAR value into R3
            ST R3, MOUSE_MMIO ; we now have the mouse MMIO ad
            LEA R4, 1
            STR R4, R3, 0 ; enable button 1
            STR R4, R3, 1 ; enable button 2
            STR R4, R3, 2 ; enable the wheel
            LD R4, WHEEL_SENS
            STR R4, R3, 3 ; set wheel sensitivity to 55
            LD R2, INT_OFFSET ; config space offset 2
            LEA R4, 1 ; int enable bit
            ST R4, DEVDATA
            ST R0, DEVNUM
            ST R2, DEVOFF ; this enables interrupts
            HALT

DEV_MAX .FILL 255

DEVNUM  .FILL xFCF8
DEVOFF  .FILL xFCF9
DEVDATA .FILL xFCFA
```

```
MOUSE_ID       .FILL xF00D
MOUSE_ID_COMP  .FILL x0FF3 ; xFFFF - MOUSE_ID
MOUSE_MMIO     .BLKW 1

WHEEL_SENS     .FILL x37

; cfg space offsets
VEN_OFFSET     .FILL x1
INT_OFFSET     .FILL x2
BAR_OFFSET     .FILL x3

NOT_FOUND .STRINGZ "ERROR: device not found"
.END
```

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?

   16

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

   32

   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?

   4 for offset within line, 5 for cache index, and 7 for the tag

   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];
}
```

   This matrix fits perfectly in the cache. We'll only miss once every 16 memory references. That means we'll miss in the cache 32 times here. That gives us a miss rate of $\frac{32}{512}$ or 6%. Not bad.

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 512×16 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?

The first option is more cache friendly. If we stride by the column index, as in the second option, we will miss in the cache much more frequently. If this matrix was small enough to fit in our cache, it wouldn't matter, since even if we loaded rows into the cache and then didn't use them subsequently, they'd still be in the cache when we got back around to that row. Since this matrix (and most matrices in practice) is

big enough, this doesn't happen since early rows will get evicted by later rows. So when we wrap around to previous rows, they're no longer in the cache!

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).

While I can't think of any code that ages like wine, picking the wrong scheme here could put you in a position where your code gets relatively worse *over time. As CPU speeds outpace memory speeds, the relative cost of an access to main memory increases. If I double my CPU's clock frequency and keep the memory system clocked the same, I have to wait on a memory access for twice as many cycles. Therefore cache misses are more troublesome, making your cache unfriendly code a slowly rotting heap of bits.*

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.