

CS220 - Computer System II
Lab 8

Due: 10/19/2016, 11:59pm

1 Introduction

In this lab, you will write **bit-manipulation and testing macros**. A macro is a fragment of code which has been given a name. Typically, macros are implemented using the **#define** directive. Macros are **inlined during compile-time**, and can therefore provide significant performance benefits. Macros, unlike functions are type agnostic. That is, a single macro can be used on multiple types. For example, the below macro can be used to add two ints, floats, longs, etc.

```
1 #define SUM(a, b) ((a) + (b))
```

2 Getting Started

Create a folder Lab8 and files **bits.h** and **bits.c** within Lab8. Create a **main.c** which has a driver program, and add a Makefile which can both build and clean the executable as well as any intermediate files (refer to previous labs/assignments if you still require help).

3 Implementation

For all tasks in this section, place macros and function declarations in **bits.h**. Place functions implementations in **bits.c**. Place the driver program in **main.c**, including **main()** itself.

1. The first task in operating with bits is to generate a vector comprising of the positions of a bit or bits we wish to manipulate. This is called the bit mask. Let us write a simple mask generation macro **BIT(n)** that generates a number with n^{th} bit set to 1 and the rest to 0. In other words, **BIT(n)** will generate 2^n .

```
1 #define BIT(n) (1 << (n))
```

While this is a simple macro, more complex macros are possible (e.g., what if I want a mask with n^{th} and m^{th} bits set to 1?).

2. Now let us implement a macro that, given a value and a mask, will set the bits represented by the mask in the value.

```
1 #define BIT_SET(v, mask)  ( v |= (mask) )
```

Notice that the OR operation will force the bits in *v* corresponding to the mask to 1.

3. Next, let us implement a macro that, given a value and a mask, will clear the bits represented by the mask in the value.

```
1 #define BIT_CLEAR(v, mask)      ( v &= ~(mask) )
```

Notice how the **not operation** generates the reverse mask with all 1's except the masked bits, which are set to 0. Is there another way to implement this macro? (Think **DeMorgan's law!**) Implement using at least one other method and record it in **Lab8.txt**.

4. What if we want to flip bits? That is, if a bit in the mask is 0, we want to set it to 1, and vice versa. An **XOR operation** is ideal for this task. Go ahead and implement it yourself.

```
1 #define BIT_FLIP(v, mask)      /* Your expansion of macro here */
```

5. Now, let us write macros to test **if any of the bits in the mask are set**.

```
1 #define IS_SET_ANY(v, mask)    ( v & (mask) )
```

Notice how if any of the bits positions represented by the mask is set in *v*, the result is non-zero. This can be easily incorporated in an if condition. Using what we know so far, let us write a function that can **print an unsigned int in binary**.

```
1 void print_in_binary(unsigned int x) {  
    /* iterate i from 31 down to 0 (assume 32 bits for unsigned int) */  
3    /* if ith bit in x is set, print 1, else print 0 */  
    /* Finally print a new line */  
5 }
```

Call `print_in_binary` from `main` and test. Here on, use this function to debug your macros. Test each of the macros written so far, and ensure correctness. Similar to `IS_SET_ANY`, write a macro `IS_CLEAR_ANY`.

- Now, let us move on to more complex tasks. A bitfield is a range of bits working as a single number. You usually can't access these ranges directly because memory is accessed in `Dword(4/8)-byte-sized data types`. Each bitfield starts at bit "start" and has a length "len". Consider the following bit-field that consists of 3 values x, y and z. x is 3 bits wide (bits 4-6), y is 4 bits wide (bits 0-4) and z is 5 bits wide (bits 7-11). Suppose the bit field is stored in variable `val`.

```
z z z z z x x x y y y y
```

Suppose you want to get the value of x, which starts at bit 4 and is 3 bits long. You can't get it directly because it is sandwiched at both sides and because it is 4 bits up. However, it can be retrieved as follows:

```
z z z z z x x x y y y y    /* val */
0 0 0 0 z z z z z x x x    /* val >> 4 */
0 0 0 0 0 0 0 0 x x x      /* (val >> 4) & 7 */
```

`Setting the bitfield x to a new value` (say `p`) is basically the inverse of this: left-shift the new value by start position of x and OR it in. However, this doesn't quite work for two reasons: first, OR would combine it with current value of x, which is rarely what we need. Second, if the new value is too large for the field, it will overflow into the next bitfield. To get around these problems, you have to `clear the whole bitfield first` and `mask off the excess bits from the new value before insertion`.

```
/* Step 1, prepare the new value for insertion */
*****p p p    /* p = new value */
0 0 0 0 0 0 0 0 p p p    /* p & 7 */
0 0 0 0 0 p p p 0 0 0 0    /* p = (p & 7) << 4 */

/* Step 2, insert the new value into val */
z z z z z x x x y y y y    /* val */
z z z z z 0 0 0 y y y y    /* val = (val & ~(7<<4)) */
z z z z z p p p y y y y    /* val = val | p */
```

So the general expressions for `getting and setting bitfields` are '`(val>>start)&mask`' and '`val = (val&~mask) | ((p&mask)<<start)`', respectively. I'm giving you the

expressions now because in macro-form, they become quite horrible thanks to the required parentheses. The one thing I've left out is **how to get the mask in the first place**. That is, we want first 'len' number of bits set to 1, remaining 0. Note that $2^n - 1$ will contain all 1's in binary representation (test it using the `print_in_binary` function). We will use this property in the macro.

```
1 #define BIT_MASK(len)      ( BIT(len)-1 )
```

Similarly, macros **BF_MASK to create a bitfield mask** and **BF_PREP to prepare a mask for insertion** can be implemented.

```
1 /* Create a bitfield mask of length len starting at bit start. */
   #define BF_MASK(start, len)      ( BIT_MASK(len)<<(start) )
3
   /* Prepare a bitmask for insertion. */
5 #define BF_PREP(x, start, len)  ( ((x)&BIT_MASK(len)) << (start) )
```

7. Based on the generalization for getting and setting bitfields, and based on the macros above, implement a macro **BF_GET to extract a bitfield of length "len" starting at bit "start" from y**, and a macro **BF_SET to insert a bitfield x of "len" bits starting at "start" position into y**. Don't forget to parenthesize your variables in the expansion!

```
1 #define BF_GET(y, start, len)    /* Your macro expansion here */
   #define BF_SET(y, x, start, len) /* Your macro expansion here */
```

8. You are given that **Virtual address is 32 bits wide**, with the following bitfields (0 is the least significant bit and 31 is the most significant bit):

- (a) Bits 0 through 11 constitute the PAGE OFFSET
- (b) Bits 12, 13 and 14 constitute TLB ID.
- (c) Bits 15 through 23 are unused.
- (d) Bits 24 through 31 constitute TLB Tag.

Implement the following get and set functions. The **get series of functions accept** a virtual address and return the `page_offset`, `tlb_id` or `tlb_tag`. The **set series of functions**

accept a virtual address and a new value for `page_offset`, `tlb_id` or `tlb_tag`, and return the new address with the modified bitfields. Use the macros implemented above.

```
unsigned int get_page_offset(unsigned int address);
2 unsigned int get_tlb_id(unsigned int address);
  unsigned int get_tlb_tag(unsigned int address);
4 unsigned int set_page_offset(unsigned int address, unsigned int
    new_offset);
  unsigned int set_tlb_id(unsigned int address, unsigned int new_tlb_id);
6 unsigned int set_tlb_tag(unsigned int address, unsigned int new_tag);
```

Test each of the get functions with address `0xf712c0d0` as input, and test each of the set functions with address `0xf712c0d0` and values `0x1a3` as page offset, `0` as `tlb_id`, and `0x8400` as `tlb_tag` respectively. Record the outputs in `Lab8.txt`.

4 Submitting the result

Remove binaries and intermediate files from `Lab8`. Create a `tar.gz` of `Lab8` folder with only the source files, header files and text files.

```
$ make clean
2 $ tar -cvzf Lab8\_submission.tar.gz ./Lab8
```

Submit `Lab8_submission.tar.gz` to Blackboard.