CS220 - Computer System II
Assignment 10 and 11 (200 points)

Due: 11/27/2017, 11:59pm

1 References

- `http://c-faq.com`
- `https://cdecl.org`
- **Intel 64 and IA-32 architectures software developer's manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, A-Z**

2 Any and All

(20 x 2 = 40 points) Implement the following macros in a header, **macros.h**:

1. Macro TEST_IF_ANY_SET(v, start, end) that tests if *any* of the bits between indices `start` and `end` (inclusive) in vector v is set.
2. Macro TEST_IF_ALL_SET(v, start, end) that tests if *all* of the bits between indices start and end (inclusive) in vector v are set.

This is along the lines of what you did in a previous lab. If you need to implement helper macros, you are free to do so. You are guaranteed that when the macro is tested, the start index will be greater than or equal to the end index.

2.1 Examples

Macro	`v`	`start`	`end`	**Evaluates**
TEST_IF_ANY_SET	0xDEADBEEFDEADBEEF	63	0	1
TEST_IF_ALL_SET	0xDEADBEEFDEADBEEF	63	0	0
TEST_IF_ANY_SET	0xDEADBEEFDEADBEEF	35	32	1
TEST_IF_ALL_SET	0xDEADBEEFDEADBEEF	35	32	1
TEST_IF_ANY_SET	0xDEADBEEFDEADBEEF	7	4	1
TEST_IF_ALL_SET	0xDEADBEEFDEADBEEF	7	4	0

3 Rotate

(80 points) Implement a function `unsigned long rotate(unsigned long val, unsigned long num, unsigned long direction);` using 64-bit Intel x86 assembly in **rotate.S**. The `direction` is 0 or 1 where 0 implies right and 1 implies left. The function implements a rotate right/left operation. It is expected to shift `val` right or left (depending on direction) `num` number of times with a rotate behavior. This means that the bit that is shifted out through left shift (the most-significant bit) is brought back in as the least-significant bit, and the bit that is shifted out during right shift (the least-significant bit) is brought back as the most-significant bit.

3.1 Examples

Val	Num	Direction	Output
0xDEADBEEFDEADBEEF	2	1	0x7AB6FBBF7AB6FBBF
0xDEADBEEFDEADBEEF	2	0	0xF7AB6FBBF7AB6FBB
0xDEADBEEFDEADBEEF	66	1	0x7AB6FBBF7AB6FBBF
0X1	1	0	0x8000000000000000

3.2 Hints

- NOTE: rotate(val, num, direction) is the same as rotate(val, num%64, direction).
- You are guaranteed that direction is either 0 or 1.
- val and num can be any unsigned long number.

4 Backtracing

(80 points) Declare a function prototype `void print_backtrace(int count)` in **bt.h** and implement a function definition in file **bt.c**. This function will print no more than `count` number of return addresses in preceding calling functions or up to main, whichever smaller. The call trace should be similar to what is obtained when we type `bt` on gdb, except the following:

- This function will not print the names of the functions in the trace, just the return addresses in the callee functions.
- This function will not print the instruction pointer's address (#0 in GDB).

You are free to implement helper functions. You are guaranteed that all functions in the program will use frame pointer, and that `main()` will have a single `ret` instruction.

4.1 Example

For example, the following source code in main.c ...

```c
#include <stdio.h>
#include "bt.h"

void baz(){
  print_backtrace(4);
}

void bar(){
  baz();
}

void foo(){
  bar();
}

int main (int argc, char* argv[]){
  foo();
  return 0;
}
```

... produces the disassembly ...

```
...
40053e <baz>:
  ...
  400547: e8 da ff ff ff   call   400526 <print_backtrace>
  40054c: 90   nop
  ...
```

```
40054f <bar>:
  ...
  400558: e8 e1 ff ff ff    call   40053e <baz>
  40055d: 90   nop
  ...
400560 <foo>:
  ...
  400569: e8 e1 ff ff ff    call   40054f <bar>
  40056e: 90   nop
  ...
400571 <main>:
  ...
  400585: e8 d6 ff ff ff    call   400560 <foo>
  40058a: b8 00 00 00 00   mov eax,0x0
  ...
...
```

Your implementation of `print_backtrace` would then produce the following output:

```
#1	0x000000000040054c
#2	0x000000000040055d
#3	0x000000000040056e
#4	0x000000000040058a
```

Note that there is a horizontal tabulation character (`\t`) between the number and the return address, and each return address is right-justified with zeroes such that it is always 18 characters wide (i.e. `0x` followed by 16 hex digits forming the address).

4.2 Hint

A backtrace is nothing but the sequence of return addresses in the preceding stack frames. A couple of points to note:

1. The return address (in the calling function) is stored on the stack.
2. On entry to each function, the old frame pointer (`rbp` register) is pushed on to the stack (its position is right after the return address).

3. In any given function, the return address is always stored at `[rbp+8]` and the base pointer of the calling frame is always stored at `[rbp]`.

4.3 Strategy

1. As a first step, find the bounds of `main` function (you may do this in a separate function). Start from address of `main`, and start reading bytes until you hit the return instruction. Note the address of the return instruction. The start of `main` function and the address of the return instruction form the bounds of `main` function.

2. The goal now is to print `count` number of return addresses in the preceding frames, or until you hit a return address that is inside main function. So:

```
curr_rbp <- get current rbp
while count > 0:
        ret_addr <- get current return address which is in [curr_rbp + 8]
        print ret_addr
        if ret_addr is an address in main:
            return
        curr_rbp <- get rbp for previous frame, which is in [curr_rbp]
        decrement count
```

3. So, how would you get the current base pointer? You have 2 choices. (1) Implement an assembly function that will move `rbp` into `rax`and return, or (2) look at the address of count and infer the address to which `rbp` points.