<README.md>

 # PES-Assignment-1
Author: Arpit Savarkar

Repository Comments
*_Contains_*
Code for Assignment 1 for PES, ECEN-5813, Fall 2020

Repository for PES-Assignment 1

- bit_operations.h - Header file which contains the function prototypes and enumerators needed for bit_operations.c
- bit_operations.c - The main script for bit manipulation and data representation styles (decimal, binary and hexadecimal) and code for hexdump from a specific location

Involves Six Functions and Unit Tests and helper functions for the following
1) uint_to_binstr(char *str, size_t size, uint32_t num, uint8_t nbits)
- Returns the binary representation of an unsigned integer, as a null-terminated string. On input, str is a pointer to a char array of at least size bytes,numis the value to be converted,and nbitsis the number of bits in the input.If the operation was successful, the function returnsthe number of characters written to str, not including the terminal \0.

2) int_to_binstr(char *str, size_t size, int32_t num, uint8_t nbits)
- Returns the binary representation of a signedinteger, as a null-terminated string. On input, stris a pointer to a char array of at least size bytes,num is the value to be converted,and nbitsis the number of bits in the input.If the operation was successful, the function returns the number of characters written to str, not including the terminal \0. In the case of an error, the function returns a negative value, and stris set to theempty string.

3) uint_to_hexstr(char *str, size_t size, uint32_t num, uint8_t nbits)
- Returns the hexadecimal representation of an unsigned integer, as a null-terminated string. On input, stris a pointer to a char array of at least size bytes, numis the value to be converted, and nbitsis the number of bits to be considered. If the operation was successful, the function returns the number of characters written to str, not including the terminal \0. In the case of an error, the function returns a negative value, and stris set to the empty string.

4) uint32_t twiggle_bit(uint32_t input, int bit, operation_toperation)
- Changes a single bitof the input value, without changing the other bits. Upon invocation, bitis in the range 0 to 31, inclusive. Returns 0xFFFFFFFF in the case of an error

5) uint32_t grab_three_bits(uint32_t input, int start_bit)
- Returns three bits from the input value, shifted down.This function’s outputis best shown graphically.
..... .XXX.... .... TO 00000000 00000000 00000000 00000XXX

6) char *hexdump(char *str, size_t size, const void*loc, size_t nbytes)
- Returns a string representing a “dump” of the nbytesof memorystarting at loc.Bytes areprinted up to 16 bytes per line, separated by newlines

Assignment Comments
This assignment demonstrates C Programming from scratch for data representation conversion and basic bit manipulation, it also demonstrates a code for reading a hexdump from a address specified.

Execution
- To run the Program :
1) make
2) ./bit_operations

- TO Use with Debug Mode :
1)gcc bit_operations.h bit_operations.c -o bit_operations
2) ./bit_operations -d

<Header File – bit_operations.h>

```
#ifndef BIT_OPERATIONS_
#define BIT_OPERATIONS_

#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
/**
*   @brief  Returns a character corresponding to hex table for decimal equivalent
*
*   Given  a   integer  to  a   char  data  set,  this  will  return  a char
*  equivalent in hex
*
*   @param  num  Integer  to  a   data  item
*
*   @return  char.
*/
char convert(int num);

/**
*   @brief  Returns a integer set with a specific bit
*
*   Given  a   integer and a bit location,  this  will  return  a uint32_t
* Set to a specified bit location
*
*   @param  num :  Integer  to  a   data  item
* @param bit : Location of a specific bit on the register corresponding to num
*
*   @return  uint32_t
*/
int set_bit(uint32_t input, int bit);

/**
*   @brief  Returns a integer set with a specific bit
*
*   Given a integer and a bit location, this  will  return  a uint32_t
* cleared to a specified bit location
*
*   @param  num :  Integer  to  a   data  item
* @param bit : Location of a specific bit on the register corresponding to num
*
*   @return  uint32_t
*/
int clear_bit(uint32_t input, int bit) ;
```

```c
/**
 *  @brief  Returns a integer set with a specific bit
 *
 *  Given a integer and a bit location, this  will  return  a uint32_t
* Toggled to a specified bit location
*
 *  @param  num :  Integer  to  a   data  item
* @param bit : Location of a specific bit on the register corresponding to num
 *
 *  @return  uint32_t
 */
int toggle_bit(uint32_t input, int bit);

/**
 *  @brief  Returns a pointer to a string corresponding to binary representation of
* unsigned uint32_t integer
 *
 *  Given a pointer to string instantiated with a specified size, function returns the
* length of the binary equivalent of a number (argument) upto a specified number of
* bits (argument)
*
 *  @param  str :  Pointer to a char  data  set
* @param size : char array Instantiated of at 'size' bytes
* @param num : Integer to be converted to binary
* @param nbits : It is the number of bits of the input
 *
 *  @return  int
 */

int uint_to_binstr(char *str, size_t size, uint32_t num, uint8_t nbits);

/**
 *  @brief  Returns a pointer to a string corresponding to binary representation of
* signed uint32_t integer
 *
 *  Given a pointer to string instantiated with a specified size, function returns the
* length of the binary equivalent of a number (argument) upto a specified number of
* bits (argument)
*
 *  @param  str :  Pointer to a char  data  set
* @param size : char array Instantiated of at 'size' bytes
* @param num : Integer to be converted to binary
* @param nbits : It is the number of bits of the input
 *
 *  @return  int
 */
```

```
int int_to_binstr(char *str, size_t size, int32_t num, uint8_t nbits);


/**
*   @brief  Returns a pointer to a string corresponding to hexadecimal representation of
* unsigned uint32_t integer
*
*   Given a pointer to string instantiated with a specified size, function returns the
* length of the hex equivalent of a number (argument) upto a specified number of
* bits (argument)
*
*   @param  str :  Pointer to a char  data  set
* @param size : char array Instantiated of at 'size' bytes
* @param num : Integer to be converted to binary
* @param nbits : It is the number of bits of the input
*
*   @return  int
*/

int uint_to_hexstr(char *str, size_t size, uint32_t num, uint8_t nbits);

/**
*   @brief  Bit Manipulation to set/clear/toggle a but at a specified bit location
*
*   Changes a single bit of the input value, without changing the other bits. Upon invocation,
bit is in
* the range 0 to 31, inclusive. Returns 0xFFFFFFFF in the case of an error.
*
*   @param  input : Integer data of whose bits are to be manipulated
* @param bit : Specific location upon which bit manipulation is to be carried out
* @param operation : Object to Enum Operation_t
*   @return  uint32_t
*/
typedef enum {
CLEAR,
SET,
TOGGLE
} operation_t;
uint32_t twiggle_bit(uint32_t input, int bit, operation_t operation);

/**
*   @brief  Bit Manipulation to return three bits from the input value, shifted down.
*
*   @param  input :  Pointer to a integer data over which bits values are to be extracted
* @param start_bit : Starting bit location from which next 3 bits would be extracted
*
*   @return  uint32_t
```

```
*/

uint32_t grab_three_bits(uint32_t input, int start_bit);
/**
*   @brief  Hex Dump of a memory location upto a selected number of bytes at a specified memory
* location
*
*   Returns a string pointer representing a “dump” of the nbytes of memory starting at loc. Bytes are
* printed up to 16 bytes per line, separated by newlines. The function returns the pointer str, which
* facilitates daisy-chaining this function into other
* string-manipulation functions such as puts.
*
*   @param  str :  Pointer to a char data set where the hex dump would be stored
* @param size : char array Instantiated of at 'size' bytes
* @param num : Address in memory from where hex dump would be recorded
* @param nbytes : Number of bytes upto which the hex values of the memory would be stored
*
*   @return  Character Pointer
*/

char *hexdump(char *str, size_t size, const void *loc, size_t nbytes);


/**
*   @brief  Helper Function to check or prevent illegal access to memory out of scope
*
*   Returns a success or failure check over number of bytes before they are proceesed
* for decimal to binary and decimal to hex conversion
*
*   @param  str :  Pointer to a char data set where the hex dump would be stored
* @param size : char array Instantiated of at 'size' bytes
* @param num : Address in memory from where hex dump would be recorded
* @param nbits : Number of bits upto which string pointer would be manipulated in memory
* @param base : Base for conversionDecimal -2 , Hexadecimal - 16
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int check_legality(char *str, size_t size, uint32_t num, uint8_t nbits, int base);


/**
*   @brief  Helper Function to convert decimal number to binary representation
*
```

```
*   Returns a pointer to a string which represents the binary representation of the
* decimal number.
*
*   @param  str :  Pointer to a char data set where the hex dump would be stored
* @param size : char array Instantiated of at 'size' bytes
* @param num : Address in memory from where hex dump would be recorded
* @param nbits : Number of bits upto which string pointer would be manipulated in memory
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
void dec_to_bin(char *str, size_t size, uint32_t num, uint8_t nbits);

/**
*   @brief  Test function to test uint_to_binstr() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Check on conversion of negative numbers
* - Check on convertion to binary numbers which require more than specified bytes
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int test_uint_to_binstr(int debug);

/**
*   @brief  Test function to test int_to_binstr() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Check on conversion to binary which require more than specified bytes
* - Segmentation Faults Check
* - Check on convertion to binary numbers which require more than specified bytes
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int test_int_to_binstr(int debug);


/**
*   @brief  Test function to test uint_to_hexstr() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Check on conversion to hexadecimal which require more than specified bytes
* - Segmentation Faults Check
* - Check on convertion to binary numbers which require more than specified bytes
*
```

```
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int test_uint_to_hexstr(int debug);

/**
*   @brief  Test function to test twiggle_bit() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Segmentation Faults Check
* - Check on convertion to binary numbers which require more than specified bytes
* - Check which uses any other setups other than SET, TOGGLE, CLEAR
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int test_twiggle_bit(int debug);

/**
*   @brief  Test function to test grab_three_bits() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Segmentation Faults Check
* - Check on bit manipulation over bits which access more than require more than specified
bytes
* - Check to prevent access to bits which are negative and greater than 30
*
*   @return  Integer ( 1 = Success, 0xFFFFFFFF = Failure )
*/
int test_grab_three_bits(int debug);

/**
*   @brief  Test function to test hexdump() function with test cases
*
*   Returns status as integer "1" if all test cases return successful, else "0"
* Test Cases include
* - Segmentation Faults Check
* - Check to get the hexdump of a specified string upto the specific bytes
*
*   @return  Integer ( 1 = Success, 0 = Failure )
*/
int test_hexdump(int debug);


#endif /* BIT_OPERATIONS_ */
```

<Program file bit_operations.c>

```
/*****************************************************************************
*Copyright (C) 2020 by Arpit Savarkar
*Redistribution, modification or use of this software insource or binary
*forms is permitted as long as the files maintain this copyright. Users are
*permitted to modify this and use it to learn about the field of embedded
*software. Arpit Savarkar and the University of Colorado are not liable for
*any misuse of this material.
*
*****************************************************************************/
/**
* @file bit_operations.c
* @brief An abstraction for bit manipulation operations and
* hexdump from a specific location
*
* This file provides functions and abstractions for bit manipulation
* decimal to binary, decimal to hex, clearing, Setting and toggling
* a bit and printing a hex dump from a specific location
*
* @author Arpit Savarkar
* @date August 27 2020
* @version 1.0
*
*

Sources of Reference :
Online Links :https://stackoverflow.com/questions/7775991/how-to-get-hexdump-of-a-structure-data
Textbooks : Embedded Systems Fundamentals with Arm Cortex-M based MicroControllers

I would like to thank the SA's of the course Rakesh Kumar, Saket Penurkar for their
support to debug the hexdump code.

*/

#include "bit_operations.h"


// ********************** Helper Functions ***********************************

char convert(int num) {
/*
Conversion Table for reference
Decimal: 0 1 2 3 4 5 6 7
Hex 0 1 2 3 4 5 6 7
Decimal: 8 9 10 11 12 13 14 15
```

```
Hex 8 9 A B C D E F
*/
if (num >= 0 && num <= 9)
return (char)(num + '0');
else
return (char)(num - 10 + 'A');
}

int set_bit(uint32_t input, int bit) {
// Returns Input manipulated to set a bit
// "|" is Bitwise OR
return (input | (1U << (bit)));
}

int clear_bit(uint32_t input, int bit) {
// Returns Input manipulated to clear a bit
// "&" is Bitwise AND
return (input & (~(1U << (bit))));
}

int toggle_bit(uint32_t input, int bit) {
// Returns Input manipulated to toggle a bit
// "^" is Bitwise AND
return (input ^ (1U << (bit)));
}

int check_legality(char *str, size_t size, uint32_t num, uint8_t nbits,
int base) {

// Functions to check segmentation fault and access to illegal number
// of bits

int len = 0;

if (size <=0) {
str[0] = '\0';
return -1;
}

// Segmentation Faults Check
if( (nbits/8) > size ) {
str[0] = '\0';
return -1;
}
// Illegal nbits
if (nbits <= 0 ) {
str[0] = '\0';
```

```
return -1;
}

int temp = num;
while (temp>0) { // Returns the modulo as binary of specified base
temp /= base;
len++;
}

if (len == 0) {
str[0] = '\0';
return -1;
}
// Seg Fault Check
if (len > nbits) {
str[0] = '\0';
return -1;
}

}

void dec_to_bin(char *str, size_t size, uint32_t num, uint8_t nbits) {

int base = 2, i =0, len = 0;

// To specify the 0bxxxxxx for the binary
str[0] = '0';
str[1] = 'b';
// Instantiating string with '0's upto nbytes
for (i =2; i < nbits+2; i++) {
str[i] = '0';
}
//Demarkating End of string
str[i] = '\0';

// Need to be stored backwards for correctness
while (num>0) {
str[--i] = convert(num % base); // Returns the modulo as binary for base 2
num /= base;
len++;
}

// To prevent Segmentation faults restricted to nbits
if (len > nbits) {
str[0] = '\0';
}
```

```
}


int uint_to_binstr(char *str, size_t size, uint32_t num, uint8_t nbits) {
int len = 0;

// Illegal setup
if(num<0)
return -1;

// Seg faults and minimum size setup check
if (check_legality(str, size, num, nbits, 2) == -1)
return -1;

// Function to convert the input "num" to Binary
dec_to_bin(str, size, num, nbits);

len = 0;
for (int i =0; str[i]!='\0'; i++)
len++;

return (len);
}


int test_uint_to_binstr(int debug) {
size_t size = 1024;
char str[size];
int ret,i;

if(debug)
printf("\n Test Results for Unsigned Integer to Binary Conversion ");
// 8 bit check

// Valid Number of Bit Check
ret = uint_to_binstr(str, size, UINT8_MAX, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, UINT8_MAX, 8, ret);
if(ret == -1)
return 0;
// InValid Number of Bits as input
ret = uint_to_binstr(str, size, UINT8_MAX+1, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, UINT8_MAX+1, 8, ret);
if(ret != -1)
return 0;
```

```
// InValid String Size - Segmentation/Bus Fault Test
ret = uint_to_binstr(str, 0, UINT8_MAX, 8);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, UINT8_MAX, 8, ret);
if(ret != -1)
return 0;

// 16 Bit Check

// Valid Number of Bit Check
ret = uint_to_binstr(str, size, INT16_MAX, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, INT16_MAX, 16, ret);
if(ret == -1)
return 0;

// InValid Number of Bits as input
ret = uint_to_binstr(str, size, UINT16_MAX+1, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, UINT16_MAX: %d",size, UINT16_MAX+1, 16,
ret);
if(ret != -1)
return 0;

// InValid String Size - Segmentation/Bus Fault Test
ret = uint_to_binstr(str, 0, UINT16_MAX, 16);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, UINT16_MAX, 16, ret);
if(ret != -1)
return 0;

// 32 bit
// Compiler interprets UINT32_MAX as -1 when assigned to uint32_t
// which results in legality check to be as negative number

// InValid Number of Bits as input
ret = uint_to_binstr(str, size, UINT32_MAX, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, UINT32_MAX, 8, ret);
if(ret >= 0)
return 0;

// InValid Number of Bits as input
ret = uint_to_binstr(str, size, UINT32_MAX, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, UINT32_MAX, 16, ret);
if(ret >= 0)
```

```
return 0;

// Valid Input Check
ret = uint_to_binstr(str, size, UINT32_MAX, 32);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, UINT32_MAX, 32, ret);
if(ret >= 0)
return 0;

// InValid String Size - Segmentation/Bus Fault Test
ret = uint_to_binstr(str, 0, UINT32_MAX, 32);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, UINT32_MAX, 32, ret);
if(ret >= 0)
return 0;

return 1;
}


int int_to_binstr(char *str, size_t size, int32_t num, uint8_t nbits) {

// If Unsigned Integer uint_to_binstr() can be used
if (num>0)
return uint_to_binstr(str, size, num, nbits);

// Function to Check Segmentation Faults and Illegal Bit Access
if (check_legality(str, size, num*-1, nbits, 2) == -1)
return -1;

num *= -1;
int i =0, c = 1;

// Decimal to Binary Conversion
dec_to_bin(str, size, num, nbits);

// 1's compliment logic
for(i =2; str[i]!='\0'; i++) {
if(str[i] == '1')
str[i] = '0';

else if(str[i] == '0')
str[i] = '1';
}
int k = i-1;

// 2's compliment logic
```

```
for (i = k; i>=0; i--) {
if(str[i] == '1' && c == 1) {
str[i] = '0';
}
else if(str[i] == '0' && c == 1) {
str[i] = '1';
c = 0;
}
}

int len = 0;
for (int i =0; str[i]!='\0'; i++)
len++;

return (len);
}

int test_int_to_binstr(int debug) {
size_t size = 1024;
char str[size];
uint8_t nbits = 16;
int ret,i;

if(debug)
printf("\n Test Results for signed Integer to Binary Conversion ");
// 8 Bit
// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT8_MIN*2, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT8_MIN*2, 8, ret);
if(ret != -1)
return 0;
// Valid Input Test
ret = int_to_binstr(str, size, INT8_MIN, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT8_MIN, 8, ret);
if(ret == -1)
return 0;

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT8_MAX*2+2, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT8_MAX*2+2, 8, ret);
if(ret != -1)
return 0;

// Valid Input Test
```

```
ret = int_to_binstr(str, size, INT8_MAX, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT8_MAX, 8, ret);
if(ret == -1)
return 0;

// InValid String Size - Segmentation/Bus Fault Test
ret = int_to_binstr(str, 0, INT8_MAX, 8);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, INT8_MAX, 8, ret);
if(ret != -1)
return 0;

// 16 Bit

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT16_MIN*2, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT16_MIN*2, 16, ret);
if(ret != -1)
return 0;
// Valid Input Test
ret = int_to_binstr(str, size, INT16_MIN, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT16_MIN, 16, ret);
if(ret == -1)
return 0;

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT16_MAX*2+2, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT16_MAX*2+2, 16, ret);
if(ret != -1)
return 0;

// Valid Input Test
ret = int_to_binstr(str, size, INT16_MAX, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT16_MAX, 16, ret);
if(ret == -1)
return 0;
// InValid String Size - Segmentation/Bus Fault Test
ret = int_to_binstr(str, 0, INT16_MAX, 8);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, INT16_MAX, 16, ret);
if(ret != -1)
return 0;
```

```
// Compiler interprets UINT32_MAX as -1 when assigned to uint32_t
// which results in legality check to be as negative number

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT32_MIN+1, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MIN+1, 8, ret);
if(ret != -1)
return 0;

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT32_MIN+1, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MIN+1, 16, ret);
if(ret != -1)
return 0;

// Valid Input test
ret = int_to_binstr(str, size, INT32_MIN+1, 32);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MIN+1, 32, ret);
if(ret == 0)
return 0;

// Invalid number of bits as input Test
ret = int_to_binstr(str, size, INT32_MAX-1, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MAX-1, 8, ret);
if(ret != -1)
return 0;

// Invalid Number of bits as input test
ret = int_to_binstr(str, size, INT32_MAX-1, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MAX-1, 16, ret);
if(ret != -1)
return 0;

// Valid input test
ret = int_to_binstr(str, size, INT32_MAX-1, 32);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, INT32_MAX-1, 32, ret);
if(ret == 0)
return 0;
```

```
ret = int_to_binstr(str, 0, INT32_MAX-1, 32);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, INT32_MAX-1, 32, ret);
if(ret != -1)
return 0;

return 1;

}


int uint_to_hexstr(char *str, size_t size, uint32_t num, uint8_t nbits) {

int base = 16, len = 0, i =0;
int k = 2;

// Illegal Num size
if(num < 0)
return -1;

// Illegal Num Bit setup
if (size <= 0 ) {
str[0] = '\0';
return -1;
}

// Illegal Num Bit setup
if (nbits <= 0 ) {
str[0] = '\0';
return -1;
}

// Illegal Length of bit setup
int temp = num;
while (temp>0) { // Returns the modulo as binary for base
temp /= base;
len++;
}

if (len == 0) {
str[0] = '\0';
return -1;
}

if (len > nbits/4) {
str[0] = '\0';
return -1;
```

```
}

str[0] = '0';
str[1] = 'x';

// Initalizing with '0's for required nbits in hex
for (i=0; i < nbits/4; i++) {
str[k++] = '0';
}

// Marking Enf of string
str[k] = '\0';

// Conversion of Decimal to Hex
while(num>0) {
str[--k] = convert(num % base);
num /= base;
len++;
}

// Length Calculation
len = 0;
for(i =0; str[i]!='\0'; i++) {
len++;
}

return len;
}

int test_uint_to_hexstr(int debug) {
size_t size = 1024;
char str[size];
int ret;

if(debug)
printf("\n Test Results for signed Integer to Hex Conversion ");

// 8 bit

// Valid Check Input
ret = uint_to_hexstr(str, size, UINT8_MAX, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, UINT8_MAX, 8, ret);
if(ret == -1)
return 0;
ret = uint_to_hexstr(str, 0, UINT8_MAX, 8);
if(debug)
```

```
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, INT8_MAX, 8, ret);
if(ret != -1)
return 0;


// 16 bit

// Valid Check Input
ret = uint_to_hexstr(str, size, UINT16_MAX, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d", size, UINT16_MAX, 16, ret);
if(ret == -1)
return 0;
ret = uint_to_hexstr(str, 0, UINT16_MAX, 8);
if(debug)
printf("\nString Size: %d, Num: %d, nbits: %d, Length: %d",0, UINT16_MAX, 8, ret);
if(ret != -1)
return 0;


// Compiler interprets UINT32_MAX as -1 when assigned to uint32_t
// which results in legality check to be as negative number

// Invalid Check input
ret = uint_to_hexstr(str, size, UINT32_MAX, 8);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, UINT32_MAX, 8, ret);
if(ret != -1)
return 0;

// Invalid Check input
ret = uint_to_hexstr(str, size, UINT32_MAX, 16);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, UINT32_MAX, 16, ret);
if(ret != -1)
return 0;

// Valid Check Output
ret = uint_to_hexstr(str, size, UINT32_MAX, 32);
if(debug)
printf("\nString Size: %ld, Num: %d, nbits: %d, Length: %d",size, UINT32_MAX, 32, ret);
if(ret != -1)
return 0;

return 1;

}
```

```
uint32_t twiggle_bit(uint32_t input, int bit, operation_t operation) {

// Invalid bit check
if ( bit <0 || bit > 31)
return 0xFFFFFFFF;

// Function call to clear specific bit
if (operation == CLEAR) {
return clear_bit(input, bit);
}
// Function call to set specific bit
else if (operation == SET) {
return set_bit(input, bit);
}
// Function call to toggle specific bit
else if (operation == TOGGLE) {
return toggle_bit(input, bit);
}
else {
// Invalid Operation
return 0xFFFFFFFF;
}

}

int test_twiggle_bit(int debug) {

uint32_t input = 0;
uint32_t output;

if(debug)
printf("\n Test Results for Twiggling particular bits of an input 32 bit number ");

// Validity Check to clear bit 0
output = twiggle_bit(input, 0, CLEAR);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 0, CLEAR,
output);
if(output == 0xFFFFFFFF)
return 0;

// Validity Check to set bit 0
output = twiggle_bit(input, 0, SET);
if(debug)
```

```
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 0, SET,
output);
if(output == 0xFFFFFFFF)
return 0;

// Validity Check to toggle bit 0
output = twiggle_bit(input, 0, TOGGLE);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 0,
TOGGLE, output);
if(output == 0xFFFFFFFF)
return 0;

// bit size restricted 0 - 31
output = twiggle_bit(input, 32, CLEAR);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 32,
CLEAR, output);
if(output != 0xFFFFFFFF)
return 0;

// bit size restricted 0 - 31
output = twiggle_bit(input, 32, SET);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 32, SET,
output);
if(output != 0xFFFFFFFF)
return 0;

// bit size restricted 0 - 31
output = twiggle_bit(input, 32, TOGGLE);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, 32,
TOGGLE, output);
if(output != 0xFFFFFFFF)
return 0;

// Invalid Bit Test
output = twiggle_bit(input, -1, CLEAR);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, -1,
CLEAR, output);
if(output != 0xFFFFFFFF)
return 0;

// Invalid Bit Test
output = twiggle_bit(input, -1, SET);
```

```
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, -1, SET,
output);
if(output != 0xFFFFFFFF)
return 0;

// Invalid Bit Test
output = twiggle_bit(input, -1, TOGGLE);
if(debug)
printf("\nInput Number: %d, Bit manipulated: %d, Operation: %d, Result: %d",input, -1,
TOGGLE, output);
if(output != 0xFFFFFFFF)
return 0;

return 1;


}


uint32_t grab_three_bits(uint32_t input, int start_bit) {

uint32_t output;
int num_elem = 3;
if (start_bit < 0 || start_bit >= 30)
return 0xFFFFFFFF;
// Logic to set the 3 bits from start_bit left to right direction
output = (((1 << num_elem) - 1) & (input >> (start_bit)));
return output;
}

int test_grab_three_bits(int debug) {

uint32_t input = UINT32_MAX-2;
uint32_t output;

if(debug)
printf("\n Test Results for Extracting 3 bits from a particular start_bit ");

// Valid Bit Test
output = grab_three_bits(input, 0);
if(debug)
printf("\nInput Number: %d, Start_bit: %d, Result: %d",input, 0, output);
if(output == 0xFFFFFFFF)
return 0;

// Invalid Bit Test
```

```
output = grab_three_bits(input, 30);
if(debug)
printf("\nInput Number: %d, Start_bit: %d, Result: %d",input, 30, output);
if(output != 0xFFFFFFFF)
return 0;

// Illegal Bit Test
output = grab_three_bits(input, -1);
if(debug)
printf("\nInput Number: %d, Start_bit: %d, Result: %d",input, -1, output);
if(output != 0xFFFFFFFF)
return 0;

return 1;
}


char *hexdump(char *str, size_t size, const void *loc, size_t nbytes) {
// Segmentation Fault Check
if (size <= 0) {
str[0] = '\0';
return str;
}
// Segmentation Fault Check
if (nbytes > size) {
str[0] = '\0';
return str;
}

// Length checks.

if (nbytes <= 0) {
str[0]='\0';
return str;
}

int i, j;
char temp[3]; // Required to restrict compiler from using 2 bytes for special characters
uint8_t rem = 0, num;
unsigned char buff[17]; // String of 17
const unsigned char * pc = (const unsigned char *)loc;
int k = 0;

for (i = 0; i < nbytes; i++) {
// Newline after 16 bytes check with necessary space/offset.

if ((i % 16) == 0) {
```

```
// Preventing newline before "zeroth" line buffer.
if (i != 0) {
str[k++] = '\n';
}

// Output the offset.
str[k++] = '0';
str[k++] = 'x';

// 0x0_"0"_ requires an extra character zero
if(i == 0)
str[k++] = '0';
// Initial Delta from location Decimal to Hex Manipulation
num = i;
do
{
rem = num % 16;
str[k++] = (rem > 9)? rem -10 + 'A' : rem + '0';
num = num/16;
} while (num != 0);
// 2 Spaces between Address and Buffer Values
for (int s =0; s<2; s++)
str[k++] = ' ';
}

// Now the hex code for the specific character.
str[k++] = '0';
str[k++] = '0';

// Hexadecimal equivalent of buffer
num = pc[i];
for(j=0; temp[j]!='\0'; j++)
temp[j] = '0';
j = 0;
while (num != 0)
{
rem = num % 16;
temp[j++] = (rem > 9)? (rem-10) + 'A' : rem + '0' ;
num = num/16;
}

// Manipulation Reversing the hex string to get the right order
j = 0;
for(j = 0; j<=1; j++)
str[--k] = temp[j];
k+=2;
```

```
// Space after the hexdump of memory after every address read
str[k++] = ' ';

if ((pc[i] > 0x20) || (pc[i] < 0x7e))
buff[i % 16] = '.';
else
buff[i % 16] = pc[i];
buff[(i % 16) + 1] = '\n';
}

// Padding out last line if not exactly 16 characters.
while ((i % 16) != 0) {
str[k++] = ' ';
i++;
}
str[k] = '\0';
return str;
}


int test_hexdump(int debug) {

const char *buf= "To achieve great things, two things are needed:\n a plan, and not quite
enough time.";
size_t size = 1024;
char str[size];

if(debug)
printf("\n HexDump from a particular given address \n");

// Valid Input Test
hexdump(str, size, buf, strlen(buf)+1);
if(debug) {
printf("\n Hex dump for string %s \n", buf);
puts(str);
}
if (str[0] == '\0')
return 0;

if(debug)
printf("\n HexDump from a particular given address \n");
// Invalid Input Test
size = 0;
hexdump(str, size, buf, strlen(buf)+1);
if(debug) {
printf("\n Size is %ld \n", size);
puts(str);
```

```
}
if (str[0] != '\0')
return 0;
return 1;

}


int main(int argc, char* argv[]) {
int status[6] = {0};
int debug;

if(argc > 1)
debug = 1;
printf("\n DEBUG Status : %d \n", debug);


status[0] = test_uint_to_binstr(debug);
status[1] = test_int_to_binstr(debug);
status[2] = test_uint_to_hexstr(debug);
status[3] = test_twiggle_bit(debug);
status[4] = test_grab_three_bits(debug);
status[5] = test_hexdump(debug);

for(int i =0; i <6; i++)
printf("\nTest: %d, Result: %d\n", i, status[i]);

return 0;
}
```

<Makefile>

```
# -*- MakeFile -*-

bit_operations: bit_operations.h bit_operations.c
gcc bit_operations.h bit_operations.c -o bit_operations
```