HW 3: Doubly Linked Lists & Heapsort

CSE/IT 122

NMT Department of Computer Science and Engineering

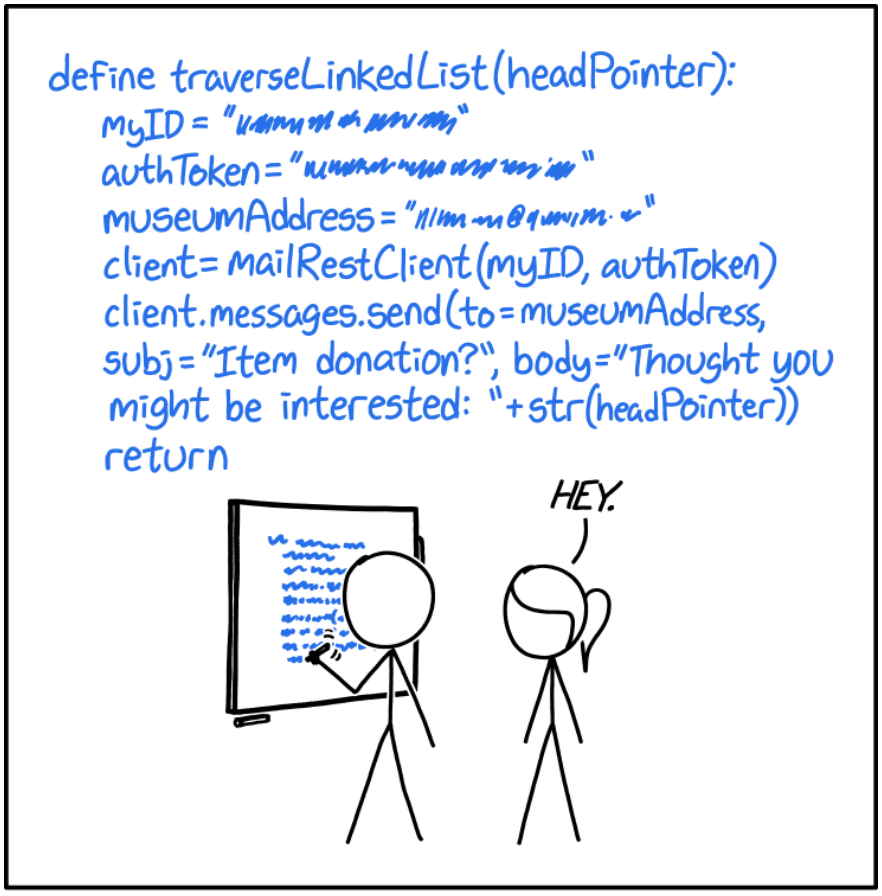

Figure 1: https://xkcd.com/2483/

Contents

1 Introduction

This homework is a continuation of last semester's lab on linked lists, but this time you will implement a binary calculator using doubly linked lists.

You will ask the user for the word size, a value from 1 to 64, whether the word is an unsigned or signed value, and enter a binary expression. Once you parse the expression, you will evaluate the expression, display the result of the calculation and set various flags as a result of the calculation just like your processor does.

Expressions are of the form *a op b* where *a* and *b* are binary bit strings less than or equal to the word size and *op* is one of the following operators: ADD (+) and SUBTRACT (-).

You are simulating binary expressions on a two's complement machine. You will use three registers (r1, r2, and r3) and a user-defined word-size to evaluate the expressions. You will model registers with a doubly linked list. For all registers the most significant bit (MSB) is at the head of the list and the least significant bit (LSB) is at the tail of the list. The MSB on a two's complement machine is the sign bit.

In addition to building a binary calculator using a doubly linked list, you will also be writing a heapsort in this homework.

2 The Register

Registers consist of *n*-bits, where *n* is the word-size. You are going to model a bit with the following node, or self-referential structure:

```
1 struct bit_t {
2     unsigned char n; /* store either 0 or 1 not '0' or '1' */
3
4     struct bit_t *prev;
5     struct bit_t *next;
6 };
```

n-bit_t structures are strung together via a doubly linked list to create a register of word-size *n*.

Since we just going to model binary expressions, we will use three registers `r1, r2` and `r3`. `r1` will hold *a*; `r2` will hold *b*; and `r3` will hold the result of *a op b*.

3 The CPU

We are going to model a CPU using a structure that contains pointers to the head and tail of each of the registers, and integers for the flags and word_size. Flags are set as a result of an operation.

```
struct cpu_t {
        int word_size;
        int unsign; //0 -- signed, 1 for unsigned
        //flags
        int overflow;
        int carry;
        int sign;
        int parity;
        int zero;
        struct bit_t *r1_head;
        struct bit_t *r1_tail;
        struct bit_t *r2_head;
        struct bit_t *r2_tail;
        struct bit_t *r3_head;
        struct bit_t *r3_tail;
};
```

The cpu_t structure creates a sentinel node for the three registers. The meaning of each member of the structure is defined below.

3.1 Word Size

Registers consist of n-bits and for our purposes here we are calling n the word size. Your code must allow varying word sizes of $1 \leq n \leq 64$.

FYI: Intel considers a word to be 16 bits. A 32-bit processor then has registers that are double words and a 64-bit processor has registers that are quad words.

Note: in all examples below the word-size is 4-bits.

3.2 Unsigned vs. Signed

This is for the *interpretation* of the decimal representation of the bit values. A CPU doesn't care if binary numbers are signed or unsigned. The CPU carries out the same operation for both. However, if `unsigned == 1` then for the decimal representation of the number the most significant bit (MSB; the head of the list) is considered positive and if `unsigned == 0`, i.e. signed and which is the default case, the MSB is considered a negative value on a two's complement machine.

For example, unsigned $1111_2 = 15_{10}$, while for signed $1111_2 = -1_{10}$.

It is important to understand that whether or not `unsign` is set or not has no bearing on how the CPU evaluates expressions. Nor does it have any impact on how flags are set. It only affects the interpretation of the number.

3.3 Overflow Flag

The CPU doesn't care if overflow occurred or not but you as a programmer can certainly check to see if the overflow flag (OF) is set. This flag is used to determine if an error occurred in the arithmetic. Overflow applies to errors with signed numbers.

You set the overflow flag with the following rules:

1. If the sum of two numbers with both sign bits set to 0 yields a number with the sign bit set to 1, the overflow flag is set to 1.

   Two positive integers should yield a positive integer not a negative integer. For example, 0100 + 0100 = 1000, which in turn sets the overflow flag. Even if the data is considered unsigned the overflow flag would still be set. You just ignore it.

2. If the sum of two numbers with both sign bits set to 1 yields a number with the sign bit set to 0, the overflow flag is set to 1.

   Adding two negative integers should yield a negative number, but if the answer is positive or zero, an error occurred. For example, 1000 + 1000 = 0000, then the overflow flag would be set to one (1). Again, the overflow flag is set even if the integers are considered unsigned.

3. In all other cases the overflow flag is set to 0.

3.4 Carry Flag

The carry flag (CF) is used to indicate if during the addition of two numbers a carry over occurred in the MSB. For example, 1111 + 0001 = 0000 and the carry flag is set to 1 (true). 0111 + 0001 = 1000 and the carry flag is set to 0 (false).

If you are using unsigned values – remember it is up to you to provide that interpretation not the machines – then you check the carry flag to see if an error occurred in the arithmetic or not.

For both OF and CF, the CPU sets their values but the interpretation of the flags is entirely up to you. The machine just performs binary operations and sets flags; it doesn't care what the flags are set to or if the data is considered unsigned or signed.

3.5 Sign Flag

The sign flag (SF) is set if the result of the operation sets the MSB to 1. Whether the integer is signed or unsigned has no bearing on the setting this flag.

3.6 Parity Flag

The parity flag (PF) is set to 1 if the number of 1's in the result register (r3) is even otherwise it is set to 0.

For example, 1100 + 0000 = 1100 sets the PF = 1; while 1010 + 0100 = 1110 sets the PF = 0.

3.7 Zero Flag

The zero flag (ZF) is set if all bits in the result register (r3) are set to 0.

1000 + 1000 = 0000 and ZF = 1; while 1111 + 1111 = 1110, and ZF = 0.

3.8 Head and Tail Pointers

The head and tail pointers point to the head and tail of each register. The tail points to the least significant bit (the bit in the 1's column), and the head points to the Most Significant Bit (MSB).

4 Operators

These are the operators you need to implement. The setting of flags follows how Intel does it.

4.1 Addition

You implement addition bit by bit and from right to left as you need the carry from the lower order bits. You are modeling the behavior of a full adder so you need bit inputs (A and B) and a carry (C_{in}) and the result of the addition results in a sum (S) and a carry out (C_{out}). The truth table for addition is:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Addition sets the OF, CF, SF, PF, and ZF.

4.2 Subtraction

Carry out subtraction like a machine. On a two's complement machine you find the complement of the number (bit flip and add 1) to convert the subtraction to an addition. For example 1111 - 1111 = 1111 + 0001 = 0000.

Subtraction sets the OF, CF, SF, PF, and ZF.

Sample Output for Subtraction. Note the switch to displaying a plus (+) sign and finding and displaying the complement of the number.

```
$ ./lab10
```

```
enter word size: 4
unsigned values [y/N]:
enter binary expression: 1111 - 1111
1111
+
0001
----
0000
flags
OF: 0
CF: 1
SF: 0
PF: 0
ZF: 1
decimal: 0
do you want to continue [Y/n]?: n
Goodbye
```

5 The logic of the program

The outline of the program's logic is:

1. Grab the word size, whether the integer is signed (default) or unsigned, and the binary expression.
2. Error check the input (see below).
3. As users do not need to enter the leading zeros, you need to zero pad the input so the length of the string matches the length of the register. That is, you need to add the leading zeros to the strings that the user didn't. For example, if the user entered 111 and the word_size is 8, the string should be 00000111.

4. Create all registers of length word_size (doubly linked lists) and copy the strings into registers r1 and r2.
5. Carry out the expression, store in register r3, and set the appropriate flags.
6. Pretty print the result (see below).
7. Either repeat or exit the program. In either case, free all memory you allocated for the registers and if you repeat make sure to set the CPU structure to appropriate default values.

6 Error Checking

You need to check for errors in the input otherwise bad things will happen. You need to error check the following:

- the word size is between $1 \leq word\_size \leq 64$
- the bit strings contain only the characters '0' and '1'
- the bit strings are less than or equal to the word size
- the operator is a valid operator: +, -

If an error occurs, tell the user what the error is and ask for new input.

Sample Error Output:

```
$ ./lab1


enter word size: -1
error in word size, must be between 1 and 64
enter word size: 0
error in word size, must be between 1 and 64
enter word size: 68
error in word size, must be between 1 and 64
enter word size: 4
unsigned values [y/N]:
enter binary expression: 1234 + 111
error in input -- something other than a 1 or 0 entered
error in first operand. retry
enter word size: 4
unsigned values [y/N]:
enter binary expression: 111111 + 1111
error in input -- length is greater than the word size
error in first operand. retry
```

```
enter word size: 4
unsigned values [y/N]:
enter binary expression: 1111 > 1111
error in operator. retry
enter word size: 4
unsigned values [y/N]:
enter binary expression: 11 + 11111111
error in input -- length is greater than the word size
error in second operand. retry
enter word size:
```

7 Pretty Print

Follow the sample below for your input and output. Note how the number of dashes changes with word size and we are printing out zero padded strings.

```
$ ./lab1


enter word size: 1
unsigned [y/N]: y
enter binary expression: 1 + 1
1
+
1
-
0
flags
carry: 1
sign: 0
parity: 0
zero: 1
unsigned decimal: 0
do you want to continue [Y/n]?:
enter word size: 64
unsigned [y/N]:
enter binary expression: 11111 + 111111111111111111111111
0000000000000000000000000000000000000000000000000000000000011111
+
0000000000000000000000000000000000000000111111111111111111111111
------------------------------------------------------------------
0000000000000000000000000000000000000001000000000000000000011110
flags
carry: 0
sign: 0
```

```
parity: 0
zero: 0
decimal: 67108894
do you want to continue [Y/n]?: n
Goodbye
```

8 Programming Requirements

Make sure you follow these requirements:

Exercise 1 (double.c, double.h).

- Use the structures `bit_t` and `cpu_t` as is.
- Use `strtok` to parse the expression.
- Where appropriate use default items for input. The user should be able just to hit "enter" to accept the default case, which is indicated by the capital letter. In fact, any other input except for the non-default case, results in the default case.
- Error processing is to be coded using switch statements and #defines.
- Use `strncat` and `strncpy` to zero pad the string.
- Registers use and store the numbers 0 and 1, not the characters '0' and '1'.
- You perform subtraction via addition. That is you find the complement before carrying out the addition.
- Make sure you print out the decimal equivalent of the binary number. This is where you will use the value of `unsign` to correctly print out the decimal equivalent of the binary number.
- If a user wants to continue, use a `goto`, or other looping mechanism, to repeat the loop.
- Check that you free all memory correctly with valgrind.
- Use a makefile to compile, test, and run your programs.

Exercise 2 (heap.c, grand.c, random.c, random.h).

- Write an integer heapsort program. The program `grand` generates random integers one per line. The `heap` program takes an integer data file (one integer per line) and sorts it using a heapsort with a max heap. The files `grand.c, random.c, random.h` and `heap.c` are provided to help get you started.

- `grand.c` is a complete program, you just have to compile and run it to create integer data sets that you feed the `heap` program to sort. `heap.c` is incomplete. `heap.c` currently reads in a file and dynamically allocates an array which holds the data of the file. Your task is to write the heapsort. You are not allowed to modify the structure `heap_t` as it contains everything you need. Use its members in your code. Your first task is to build a heap and the second step is to delete the elements. *NB* implement the heap insertions and deletions as discussed in class.
- You are creating a max heap. You are required to build and sort the heap using only the existing storage `heap->data`. That is, you build the heap and sort it *in place*. **Important: do not allocate another array to build or to sort your heap as that is waste of storage space**.

Sample Input

```
$ ./grand -n 5 -m 3 -f data
$ ./heap data
```

Sample Output

```
0
1
1
1
2
```

Make sure you free all memory that is allocated. Check with valgrind.

FYI: the heap program is simple version of the UNIX utility `sort`.

Submitting

You should submit your code by placing all files listed below into an archive file (.zip, .gz, .tar.gz) and uploading that archive file to Canvas before the due date. The files that should be included in your archive file are listed below in the **List of Files to Submit**.

List of Files to Submit

Exercises start on page 8.