CS203 Lab 6 – Text Calculator

Assigned: Monday October 6th, 2025.
Due: Tuesday October 21st, 2025 by 11:59 pm.

The Lab Assignment.

Create a text-based calculator using memory units called registers to store your results and instructions you enter to perform work upon these memory units.

Running the Program.

When the program is run, no command-line arguments are needed. But once running, the program will continually loop, waiting for the next instruction to act upon. An example program run is provided below, you can assume there are five registers: R0, R1, R2, R3, and AC. The 'R' registers are general purpose registers of sixteen-bits each. The 'AC' register is the accumulator, also 16-bits, where all math and bit-wise results are stored by default. In the example, the instruction 'display' is used to display the current state of the calculator, showing the currently stored binary value in both hexadecimal and decimal representation. An overflow bit is also shown, which is whether the last math operation ('add' and 'sub') produced an overflow result. The display and bit-wise instructions will not manipulate the overflow bit.

```
./calc-umax-100
> display

    bin--------------   hex---   dec---  overflow: 0
R0: 00000000 00000000   0000     0
R1: 00000000 00000000   0000     0
R2: 00000000 00000000   0000     0
R3: 00000000 00000000   0000     0
AC: 00000000 00000000   0000     0

> load R1 -34
> load R2 0x3AB
> load R3 0b1001101

> display

    bin--------------   hex---   dec---  overflow: 0
R0: 00000000 00000000   0000     0
R1: 11111111 11011110   FFDE     -34
R2: 00000011 10101011   03AB     939
R3: 00000000 01001101   004D     77
AC: 00000000 00000000   0000     0

> add R0 R2
> display

    bin--------------   hex---   dec---  overflow: 0
R0: 00000000 00000000   0000     0
R1: 11111111 11011110   FFDE     -34
R2: 00000011 10101011   03AB     939
R3: 00000000 01001101   004D     77
AC: 00000011 10101011   03AB     939

> mov AC R0
```

```
> display

    bin--------------   hex---   dec---  overflow: 0
R0: 00000011 10101011   03AB     939
R1: 11111111 11011110   FFDE     -34
R2: 00000011 10101011   03AB     939
R3: 00000000 01001101   004D     77
AC: 00000011 10101011   03AB     939

> sub R0 R3
> display

    bin--------------   hex---   dec---  overflow: 0
R0: 00000011 10101011   03AB     939
R1: 11111111 11011110   FFDE     -34
R2: 00000011 10101011   03AB     939
R3: 00000000 01001101   004D     77
AC: 00000011 01011110   035E     862

> load R0 0x56AD
> load R1 0x56AD
> add R0 R1
> display

    bin--------------   hex---   dec---  overflow: 1
R0: 01010110 10101101   56AD     22189
R1: 01010110 10101101   56AD     22189
R2: 00000011 10101011   03AB     939
R3: 00000000 01001101   004D     77
AC: 10101101 01011010   AD5A     -21158
```

The above example uses a subset of the instructions provided, shown on the next page, with the overflow only being set after the last 'add' instruction and just before the display instruction. The reason the overflow is set is because the decimal value 22189 is added to 22189 for a result of 44378, but a sixteen-bit two's complement register can only index up to $2^{16-1} - 1$ or 32,767.

All registers will be represented as string of 16 char, with each char location containing either a zero or a one, and all operations will be performed by manipulating the character strings. (An array of char can also be used.) You cannot use any string manipulation functions from the system-library, instead the program must use the basic operations as we have done up to this point.

Instructions to implement.

You will need to implement the following instructions, similar to SSAM:

```
display
halt
clear
mova rA                         R[rA]    <= R[AC]
mov  rA rB                      R[rA]    <= R[rB]
load reg   immediate_val        R[reg]   <= immediate_val
add  reg-a reg-b                R[AC]    <= R[reg_a]  + R[reg_b]
sub  reg-a reg-b                R[AC]    <= R[reg_a]  - R[reg_b]
and  reg-a reg-b                R[AC]    <= R[reg_a]  & R[reg_b]
or   reg-a reg-b                R[AC]    <= R[reg_a]  | R[reg_b]
xor  reg-a reg-b                R[AC]    <= R[reg_a]  ^ R[reg_b]
neg  reg                        R[reg]   <= -R[reg]
shl  reg   pos-immediate-val    R[reg]   <= R[reg]   << pos-immediate-val
shr  reg   pos-immediate-val    R[reg]   <= R[reg]   >> pos-immediate-val
```

The instruction semantics are provided using Register Transfer Notation (RTN), which is a short hand for defining how assembly instructions behave. For example, instruction 'add' will perform a two's complement addition of register reg-a and reg-b, called the operands, storing the result in the accumulator (AC). Each 'reg' operator in the instruction list can be any register: R0, R1, R2, or R3. Registers can be added to themselves, so "add R0 R0" will have the effect of doubling the R0 value and storing the result in register AC.

- All registers will store encoded data as strings of zeros and ones. Data types like short cannot be used and the program cannot store the encoded data as a hexadecimal or decimal value. Instead, you should either use a pointer variable or an array of char. All calculations will be performed by reading these strings directly.
- The 'display' instruction will always display the current registers in the example shown above.
- The 'halt' instruction will cause the program to terminate.
- The 'clear' instruction will resent the registers and overflow bit to zero.
- The 'mova' instruction will move the AC to one of the general-purpose registers.
    - 'reg-a' can only be: R0, R1, R2, or R3.
- The 'mov' instruction will move one register to another with the following restrictions.
    - 'reg-a' can only be: R0, R1, R2, or R3.
    - 'reg-b' can only be: R0, R1, R2, or R3.
- The instructions 'add' and 'sub' will perform their work using two's complement notation and need to determine if an overflow was produced, and setting a char variable to indicate the result.
- The instructions 'and', 'or', and 'xor' will not perform any carry values and will not change the value of the overflow-bit.
- The 'neg' instruction will negate the identified register, where it will be converted to two's complement by a bit flip and an addition of one. This negation will be on the register data and the result will not be stored to the accumulator (AC).
- The 'shl' and 'shr' instructions (shift-left and shift-right, respectively) will move the bits to the left or right by the provided 'pos-immediate-val' magnitude.
    - The 'shl' instruction will pad the lower bit position with zeros.
    - The 'shr' instructions will pad the upper-bit position with the value that was previously stored. So, if the high-bit was set to 1 and the register is shifted right by three, then three 1's will be added to the vacated locations.
- The 'immediate-val' operand identifies a numeric value in one of three formats:
    - decimal – given as a decimal value starting with either a digit or a minus sign (indicating a negative value).
    - hexadecimal – identified with the two starting characters "0x".

    - binary – identified with the two starting characters “0b”.
- The ‘pos-immediate-val’ operand is the same as ‘immediate-val’, but will never be negative. Thus, if a -3 is given, then that will be an error, or if ‘0b1101’ is given then it should be interpreted as the positive value 13.

Parsing an Input String.

The book gives some basic information for retrieving data from the stdin using getchar and scanf. There are other ways of doing this, one is a stdio.h function called fgets. Below is example code for using this function.

```
#include <stdio.h>

#define MAXLINE 1024

int main() {

  char data[MAXLINE];

  // Data    contains a pointer to a buffer
  // MAXLINE lists the size of the buffer
  // stdin   is the input stream
  fgets(data, MAXLINE, stdin);

  printf("%s\n", data);

  return 0;
}
```

The above code sets a maximum buffer size, with the variable `data` being the buffer. By passing the maximum buffer size, fgets will never read more than the size of the buffer making the function "safe". For this assignment, making a large buffer means you should not overflow the buffer. Using this example, you can use fgets to pull in instructions as a string that you can the manipulate to capture input instruction parts. For example, assume the string "`add R0 R1`" is entered into your calculator. You can parse this character string to create three different sub-strings with the following code.

```
#include <stdio.h>

#define MAXLINE 1024

int main() {

  char data[MAXLINE];

  char *tmp  = NULL;
  char *cmd  = NULL;
  char *rega = NULL;
  char *regb = NULL;

  fgets(data, MAXLINE, stdin);

  tmp = data;
  cmd = data;

  while(*tmp && *tmp != ' ') tmp++;

  if(*tmp == ' ') { // operand found
    *tmp = '\0';    // terminate the sub-string
    tmp++;          // move to the next character
    rega = tmp;     // store the next substring as the register operand
  } else {
    printf("Input terminated prematurely.");
  }

  while(*tmp && *tmp != ' ') tmp++;

  if(*tmp == ' ') { // operand found
    *tmp = '\0';    // terminate the current substring
    tmp++;          // move to the next substring
    regb = tmp;     // store the next substring as the next register operand
  } else {
```

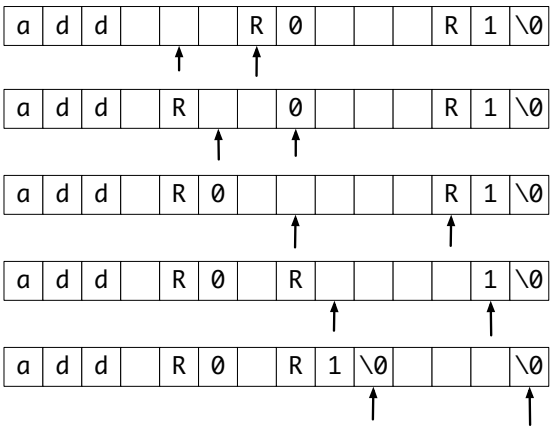
```
    printf("Input terminated prematurely.");
  }

  printf("cmd  : %s\n", cmd);
  printf("reg-a: %s\n", rega);
  printf("reg-b: %s\n", regb);
  return 0;
}
```

The program will produce the following, given the appropriate input.

```
% ./scanf_ex
add R0 R1
cmd  : add
reg-a: R0
reg-b: R1
```

Once you have the string, you can determine what each of the sub-strings are with this technique. One tricky problem is if there are multiple spaces, requiring you to shift the extra spaces out of the string. An example is provided here where two pointers are used walk through the string shifting characters. The left pointer identifies the destination and the right is the source and works by sliding these pointers across the space. The process is complete when the right pointer has reached the null terminator.



The general algorithm for parsing is the following:

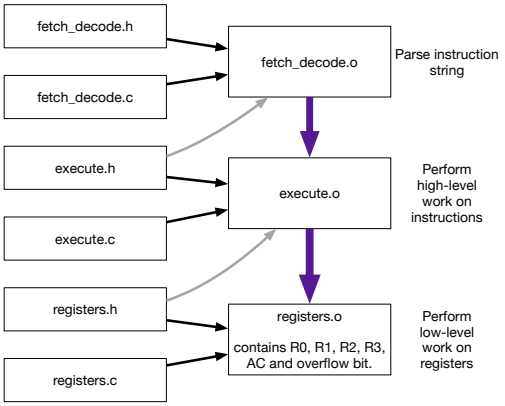
1. Read in a string using the function 'fgets'.
2. Remove multiple spaces using the two pointers walking through the string to remove the excess spaces.
3. Store the address of the first character in a pointer variable for the instruction.
    a. Iterate through the string looking for the space and null terminate the string.
4. Once the instruction is determined, process the remaining operands accordingly based on the format of the instruction. This is done in the same way that the instruction is isolated.
5. Once all the parts of the instruction have been determined, pass the strings to the appropriate function located in the "execute.c" source file.
    a. This instruction can then manipulate the registers in the "register.c" source file.

If an error is detected in the instruction string, print a message indicating the problem, then stop decoding the current instruction and continue the request loop for future instructions.

Project Requirements.

For this lab, a partial project will not be provided and the requirements are listed here.

- There will be three paired source and header files. These are stratified to allow you to implement and test the lowest level functionality first, and then build up to higher level functionality.



  - **fetch_decode** : will contain all functions for requesting input from the user and decoding what actions will need to happen. These functions will then drive the functions in the “execute.c” source file, which provides functions for executing the instructions.

  - **execute** : will perform all work on the stored registers, but must do so through low-level functions provided by the “registers.c” source file. Thus file “execute.c” will implement the high-level instructions, but the low-level functions provided by “registers.c” will do the actual work on the strings.

  - **registers** : file contains all the registers as global values that are not revealed in the associated header file. This is how data hiding is implemented in C. The only functions that can use this global information are in the registers.c file. Thus, these strings are only accessible to this source file, and the raw data can only be manipulated by functions provided by the register files. However, the functions in registers.c are identified in the header file so that execute.c can access this functionality.

- As indicated previously, all registers will be implemented as strings of characters. This can be done by either mallocing a character string or using a statically defined array of characters. In either case, you might still create a MAXREGSIZE constant with a “#define” macro in the “registers.h” file.

- You are not allowed to use any string handling functions beyond what you create. Instead, all string manipulation will be provided by functions that you create for simple manipulations of the strings.

- Additionally, the math operations will also be implemented as functions that take pointers to the various strings being manipulated.

- To implement the different math instructions, the lecture notes on signed and unsigned numbers will be helpful.

Lab Submission.

Once done, zip the lab assignment and submit it to the Moodle.