CEG 3136 – Computer Architecture II
Tutorial 2 – Translating C Code to Assembler Fall 2019

1) Complete the translation of the C function to Assembler.

```
/*------------------------------
Function: addByteArray
Parameters: arrPt – pointer to array
            num – number of elements
                  to sum
Description: Adds the contents of
             an integer array.
Assumption: array contains at
            least one element.
------------------------------*/
int addByteArray(byte *arrPt,
                 byte num)
{
   int sum;
   sum=0;
   do
   {
      sum=sum+*arrPt++;
      num--;
   }while(num != 0);
   return(sum);
}
```

```
;-------------Assembler Code-------------
; Subroutine – addByteArray
; Parameters – arrPt – register X
;              num – register Y
; Results: sum – register D
; Description: Adds contents of an integer
;              array.

addByteArray: PSHX
              PSHY ; preserve Registers
              LDD #0 ; sum=0;









              PULY ; restore registers
              PULX
              RTS  ; return(sum);
```

2) Translate the following short C program into assembler. Use the stack to exchange ALL parameters and the result (assume *int*'s take 2 bytes of storage, and *byte*'s take 1 byte of storage). Ensure that the registers used by the subroutine are not changed after the subroutine has executed. Define the stack usage with the OFFSET directive and labels as offsets into the stack.

```
/*------------------------------
Function: addInts
Description: Adds two 8-bit integers.
------------------------------*/
int addInts(byte val1, byte val2)
{
   int sum;
   sum = val1 + val2;
   return(sum);
}
```

3) The C standard library provides a function to concantenate two strings:

```
char *strcat(char *str1, char *str2)
```

A *string* of characters terminated with a null character is stored in the memory starting at address found in the pointer variable *str1* and a second string at address found in the pointer variable *str2*. Develop a structured assembly subroutine that concantenates the contents pointed to by *str2* to the end of the contents in string *str1*. The function returns the address of string *str1* (that is the address received in the *str1* variable). Assume single byte ASCII characters.

1. First provide a C function that illustrates the algorithm of the subroutine (or functions/subroutines – you may use additional functions/subroutines to provide a solution).
2. Then translate the C functions to assembler code to subroutine(s). Do not forget to comment your code.

Algorithm (C program and description):

Assembler Source Code:

```
; Subroutine: char* strcat(char *str1, char *str2)
; Parameters
;       str1 -

;       str2 -

; Returns
;       str1 -

; Local Variables


; Description: appends the second string to the first string.
```

4) The following C functions were developed as part of the design for the Alarm System Simulator in Lab 1 to check validity of codes being entered by the user (1 digit at a time). Translate the functions to an assembler subroutines. The `isCodeValid` function accesses a global array that contains 4 integer (2 bytes) alarm codes. Note that the ASCII digit is translate to its numeric value.

```
// Declare alarm code array
#define NUMCODES 5
int alarmCodes[NUMCODES];

/*----------------------------
 * Functions: checkCode
 * Parameters: input - input character
 * Returns: TRUE - alarm code detected
 *          FALSE - alarm code not detected
 * Descriptions: Creates alarm code using digits entered until
 *               4 digits are seen. After 4th digit, see if
 *               alarm code is valid using isCodeValid().
 *----------------------------*/

byte checkCode(byte input)
{
   static int mult = 1000; // current multiplier of digit
   static int alarmCode = 0;   // alarm code value
   byte retval = FALSE;

   if(isdigit(input))
   {
      alarmCode = alarmCode + (input-ASCII_CONV_NUM)*mult;
      mult = mult/10;
      if(mult == 0)
      {
         retval = isCodeValid(alarmCode);
         alarmCode = 0;
         mult = 1000;
      }
   }
   else
   {
      alarmCode = 0;
      mult = 1000;
   }

   return(retval);
}
```

```
/*-----------------------------
 * Functions: isCodeValid
 * Parameters: alarmCode - integer alarmCode
 * Returns: TRUE - alarm code valid
 *          FALSE - alarm code not valid
 * Descriptions: Checks to see if alarm code is in the
 *               alarmCodes array.
 *-----------------------------*/
byte isCodeValid(int alarmCode)
{
   int *ptr;  // pointer to alarmCodes
   byte cnt = NUMCODES;
   byte retval = FALSE;
   ptr = alarmCodes;
   do
   {
      if(*ptr++ == alarmCode)
      {
         retval = TRUE;
         break;
      }
      cnt--;
   } while(cnt != 0);
   return(retval);
}
```