Lab 2

1. Objectives

In this lab, you will learn more AVR assembly programming by

- implementing positional division,
- evaluating a polynomial,
- implementing C functions, and
- using macros.

2. Preparation

This lab may take a significant amount of time. You need to write most of your programs at home in order to finish Lab 2 in time.

3. Tasks

There are four tasks, and each task is worth 5 marks.

3.1 Task A: Positional Division (5 marks)

Hand-division uses a series of left shifts, magnitude checks and multiple subtractions to get the final answer. Consider the decimal (base-10) number system. For example, 3217/16 can be calculated as:

1. Shift the divisor 16 to the left as many times as possible, until just before it becomes greater than the dividend 3217. This means it is left-shifted by two digits; the shifted divisor is 1600.
2. Subtract a multiple of this shifted divisor ($2 \times 1600 = 3200$) from the dividend, leaving 17 as the partial quotient of 200.
3. In the second iteration, shift the new divisor 1600 right by one digit to become 160. This is greater than the partial remainder 17, so do not subtract anything.
4. In the third iteration, shift the new divisor 160 right by one digit to become 16.
5. Subtract a multiple of this shifted divisor ($1 \times 16 = 16$) from the new dividend (the previous partial remainder) 17, leaving 1 as the new partial remainder. Add the multiple 1 to the previous partial quotient of 200, giving 201.
6. Finally, stop the iteration here, as no more right shifts are possible. The old partial quotient of 201 becomes the actual quotient (result); the old partial remainder becomes the actual remainder.

The same idea can be applied to the division in the binary number system. The program in Figure 1 is an implementation of this positional division algorithm in C. Assume that the register pair r16:r15 store the dividend, r18:r17 store the divisor and r20:r19 store the quotient. Write an AVR assembly program to implement the C program pos-div.c.

```
int posdiv(unsigned int dividend, unsigned int divisor)
{
  unsigned int quotient;
  unsigned int bit_position = 1;
  quotient = 0;
  while ((dividend > divisor) && !(divisor & 0x8000))
   {
    divisor = divisor << 1;
    bit_position = bit_position << 1;
   }
  while (bit_position > 0)
   {
    if (dividend >= divisor)
      {
       dividend = dividend - divisor;
       quotient = quotient + bit_position;
      }
    divisor = divisor >> 1;
    bit_position = bit_position >> 1;
   }
  return quotient;
}
```

Figure 1: pos-div.c

3.2 Task B: Evaluating A Polynomial Function (5 marks)

The C program polynomial.c shown in Figure 2 evaluates a polynomial function $F(x)=a_n x^n + a_{n-1} x^{n-1} + ... + a_1 * x + a_0$ by using Horner's method, where $a_0, a_1, ..., a_n$, and x are one-byte signed integers, and n is a one-byte unsigned integer, and *result* is a two-byte signed integer.

Write an AVR program to implement the C program. Your program must satisfy the following requirements:

- All the constants are stored in the program (FLASH) memory.
- All the global variables are stored in the data memory (SRAM). So you need to initialise the array poly[] in your program by copying each coefficient from the program memory to its corresponding location in the data memory and do the similar thing for x and n.
- You must use at least one macro in your program.
- Use the register r20 to indicate whether an overflow occurs during the execution of your program. If there is an overflow, set r20 to 1. Otherwise, set r20 to 0.

```
// Evaluate a polynomial using Horner's method
// Let us evaluate 100x5 – 60x4 + 120x3 – 100x2 +50 x -70  for x = 5
signed char poly[6] = {100, -60, 120, -100, 50, -70};
signed char x = 5;
unsigned char i, n=5;
short int result;

int main()
   {
      result=poly[0];
      for (int i=1; i<=n; i++)
           result = result*x + poly[i];
      return 0;
   }
```

Figure 2: polynomial.c

When demonstrating your program, you need to show two scenarios: one with an overflow and one without overflow.
Hint: you need to work out an algorithm for multiplying a two-byte signed number by a one-byte signed number.

3.3 Task C: Hanoi Tower Problem (5 marks)

The following is an excerpt from the book Concrete Mathematics - A Foundation for Computer Science (2nd Edition) by Graham, Knuth and Patashnik: A neat little puzzle called the Tower of Hanoi, invented by the French mathematician Edouard Lucas in 1883. We are given a tower of eight unequally sized disks, initially stacked in decreasing size on one of three pegs. The objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never moving a larger one onto a smaller. Lucas furnished his toy with a romantic legend about a much larger Tower of Brahma, which supposedly has 64 disks of pure gold resting on three diamond needles. At the beginning of time, he said, God placed these golden disks on the first needle and ordained that a group of priests should transfer them to the third, according to the rules above. When they finish, the Tower will crumble and the world will end.

Now the question arises: How many moves are necessary and sufficient to perform the task?
The C program shown in Figure 3 solves this problem, where n is the number of disks and A, B and C denote the three pegs. After the program is completed, the value of counter is the total number of moves.

Assume that the number of disks is at most 15. Write an AVR assembly program using the C code shown in Figure 3. Your program must satisfy the following requirements:

- The initial values of all variables are stored in the program (FLASH) memory.
- All the global variables are stored in SRAM.
- All the local variables and parameters of a function are stored in the stack frame of the function.

- No overflow occurs during the execution of your program.

How many bytes of SRAM memory are needed for the stack to run this program correctly?

```
#include <stdio.h>
int counter = 0;

void move(int n, int A, int C, int B)
 {
  if (n==1)
     counter++;
  else
    {
     move(n-1, A, B, C);
     move(1, A, C, B);
     move(n-1, B, C, A);
    }
 }

int main()
  {
   int n=8, A=1, B=3, C=2;
   move(n, A, B, C);
   return 0;
  }
```

Figure 3: tower-Hanoi.c

3.4 Task D (5 marks)

In this part, you will write an AVR assembly program to perform Hoare's Quicksort algorithm. A C program that implements the Quicksort algorithm is provided in Figure 4. Write an AVR assembly program to implement the C program. Your AVR assembly program must satisfy the following requirements:

1. The array *test* is stored in the SRAM.
2. The initial values of the array *test* are stored in the program memory as constants.
3. The size of each element of the array *test* is TWO bytes.
4. All local variables are stored in the stack.

```
int partition(int *array, int p, int r)
  {
     int pivot, i, j, tmp;

     pivot = array[p];
     i = p;
     j = r + 1;

    while (1)
      {
         do  i++;
         while ( array[i] <= pivot &&i<=r );
         do  j--;
         while (array[j] > pivot );
         if (i >=j ) break;
         /* swap array[i] and array[j] */
         tmp = array[i];
         array[i] = array[j];
         array[j] = tmp;
      }
    /* swap array[p] and array[j] */
     tmp=array[p];
     array[p]=array[j];
     array[j]=tmp;
     return j;
  }

void quicksort (int *array, int p, int r)
 {
  int q;
  if ( p < r )
     {
      q = partition(array, p, r);
      quicksort(array, p, q-1);
      quicksort(array, q+1, r);
     }
 }

int test[10] = {100, 200, -70, -20, 50, 30, 60, -40,100, 30};
int main()
 {
  quicksort(test, 0, 9);
  return 0;
 }
```

Figure 4: Quicksort

4. Deadline

The deadline of this lab is Week 5.