Assignment 5. AVR

Exercise 1. (basic) Write a voter program for the ATmega328P. Inputs are PD2, PD1 and PD0; output is PB5. The output should activate whenever two or more input pins are set to “1”.

Exercise 2. (basic) Write a program for the ATmega328P that activates pin PB5 when an input number in PORTD is greater than 27.

Exercise 3. Write a program for the ATmega328P to control a water pump that fills up a water tank.

a) (basic) The level of the tank is detected through pins PD3 to PD0 in PORTD. The pump is controlled by PB5 in PORTB. The pump should be activated when the level in the tank is below 3, and turned off when it reaches level 12 (or higher). The level is encoded in natural binary and the output is active-high.

b) (medium) There would be any benefit in encoding the level in gray code? Modify the program to use the input level in gray code.

Exercise 4. (basic-medium) Write an assembly subroutine that finds the minimum and maximum values in a list of unsigned bytes in data memory.

- The subroutine takes two arguments: list address (in r25:r24) and the length of the list (in r22).
- Return the maximum value in r24 and the maximum value in r22.
- The subroutine should adhere to the C calling convention.
- Write a test program in assembly.

Exercise 5. (medium) Write a subroutine in assembly that calculates the seven segment hexadecimal code of 4-bit natural binary number. The argument is passed to the subroutine in register r24 and the result is returned in register r24 as well. The subroutine should meet the C calling convention. Clue: you may use a conversion table in program memory.

Write a test program to test the subroutine. For example, a program that counts from 0 to 15 and call the subroutine for 7-segment conversion. Or, better than that, a program that reads a 4-bit number from PORTB and write the 7-segment code to PORTD, so it can be tested in a real board.

Exercise 6. (medium) Write an assembly subroutine that adds a list of numbers in data memory.

a) The numbers are bytes. Arguments to the subroutine are: list address (r25:r24), number of bytes to add (r22).

b) The numbers are bytes. Arguments to the subroutine are: list address (r25:r24), first element (r22), last element (r20).

c) The numbers are 16-bit words. Arguments to the subroutine: list address (r25:r24), number o words to add (r22).

In all cases:

- Adhere to the C calling convention.
- Return the result as a 16-bit word in the correct registers according to the C calling convention.
- Write a test program in assembly.
- Write a test program in C language (optional).

Exercise 7. Write a program for the ATmega328P on an Arduino UNO board that implements a digital die.

a) (medium) The input is a push button connected to PB0. The input pin is zero when the button

is pressed and it should active the pull-up resistor in the input port so that an external pull-up is no necessary. The output is a seven segment display connected to PORTD. When the button is pressed, the output goes through numbers 1 to 6 very fast and in a cyclic way. The numbers stop changing when button is released, creating randomness.

Organize the code in various sections. E.g.: initialization of ports and registers, main program/loop (check button press, rotate the dice, etc.), 7-segment code generation (you may use the subroutine done in a previous exercise), etc.

Specify how to connect the 7-segment display and button to an Arduino UNO board.

b) (medium-advanced, depending on solution) Create a new version that emulates the throwing of two dice of 1 to 6 points. You should still use just one 7-segment display and the same connections than before. Describe your solution and see if it will yield the same probability than the throwing of two real dice.

Exercise 8. We want to design a simple electronic game that activates 8 LEDs in sequence from LED0 to LED7 in a cyclic way. The game has two push buttons. When button “start” is pressed, the sequence starts. When button “stop” is pressed the sequence stops at the currently active LED. The objective of the game is to stop the sequence at the highest possible LED number. These are the points earned by the player in that round. Then, the “start” button is pressed and the next player tries. The game is won by the player that collects more points after 10 rounds.

a) (medium-advanced) Implement the game in assembly code for the Arduino UNO board. Use PD0 to PD7 as LED output pins, PB0 as the “start” button and PB1 as the “stop” button. Make the game to repeat the sequence once every second approximately. You can use the “delay_ms” subroutine in the avr-bare repository or write your own delay subroutine.

b) (advanced) Improve the game by making it possible to adjust the difficulty level (how fast the sequence repeats). Use PB2 and PB3 to select the difficulty level. Define how the values in PB2 and PB3 affects the difficulty level.

Exercise 9. Write a program that blinks a LED with a frequency $f = 1 + div$, where *div* is 3 bit number. *div* is selected with PINB[2:0]. Output for LED connection is PORTB5. Use the delay_ms subroutine of example 5 to control the time.

a) (medium) Write the main program in AVR assembly.

b) (advanced) Write the main program in AVR C (use example 10 as reference).

Exercise 10. Modify the blink program in the unit's example using Timer1 so that:

a) (basic) The LED blinks only when PB0 is set.

b) (medium) The LED blinks for 5 seconds only, every time PB0 is set.

c) Comment possible applications of the program in (b).

Exercise 11. (advanced) Checksum algorithms are extensively used to detect errors in stored or transmitted data. One of the most simple of these algorithms is the parity byte[1] which computes the XOR operation of all the bytes in the data and appends the result to it. To check that there are no errors, it is enough to compute the XOR of the received data (including the checksum) and if the result is different from zero, an error have occurred. This method can detect any single bit error.

a) Write a subroutine *parity_gen* that generates the parity byte of a list of bytes in memory. The subroutine takes the address of the list and the number of bytes in the list (less than 256) as parameters and returns the value of the checksum.

b) Write a subroutine *parity_check* that takes two arguments: a list of bytes and a parity checksum and returns “1” in case the checksum is correct and “0” otherwise.

1 https://en.wikipedia.org/wiki/Checksum

c) Write a test program to test *parity_gen* and *parity_check*.

In all the cases:

- Adhere to the C calling conventions.
- Specify which registers you will use to hold the arguments to the subroutines and the return values.

Exercise 12. (advanced) Write a subroutine that sorts a list of unsigned bytes in data memory from lower to higher.

Input:

- r25:r24 – address of the list.
- r22 – size of the list (number of elements).

Implement the sorting algorithm that you prefer or use the *qsort* function from the C library. Write a program to test the subroutine.

Exercise 13. Standard C and other languages strings are assumed to end in a null character (zero byte).

a) (basic-medium) Write an assembly subroutine (*strlen_asm*) that counts the number of characters of a string in memory. The routine takes the address of the string as argument and returns a 16-bit number with the length in bytes excluding the null byte. The subroutine should be callable from C programs.

b) (medium) Write assembly and C programs to test the subroutine. Run the programs in a real microcontroller or in a debugger using a simulator.

c) (medium-advanced) Disassemble the *avr-libc* version of *strlen* and compare it to *strlen_asm*. Does *strlen* do the same as *strlen_asm*? Does *strlen* do it in the same way? Is it similar somehow?

   NOTE: in GNU/Linux systems with the `avr-libc` package installed you can find the *avr-libc* library object code at file `/usr/lib/avr/lib/libc.a`.

d) (medium) Find the source code for the *avr-libc*[2] version of *strlen*. Is it written in C language or assembly? Why do your think the developers opted for one solution or the other? Are all the functions of the standard C library in *avr-libc* written in C/assembly?

Leyend

- (basic): the exercise uses basic concepts and the difficulty is low.
- (medium): the exercise may use more advanced concepts and the difficulty is a bit higher: the solution may not be obvious at first glance.
- (advanced): the exercise uses some of the most advanced topics studied in the unit and some details of the solution may need a thorough thinking.

2 https://github.com/avrdudes/avr-libc