Microprocessors Lab

Dr Asimakis Leros

Lab exercise 2

Objective

- Implementation of loops in assembly
- Arrays and pointers
- Parallel port programming (digital input–output)

Notes

- We will be using the Arduino board. You will have to implement a simple circuit.
- Your lab report must include:
  - short answers (e.g. two lines) to numbered *questions*,
  - your code, results and comments in numbered **programming assignments**.

1 Implementation of loops

Loops are implemented using branch and jump instructions, as are all program flow operations. There are two main ways to construct a loop. The first corresponds to for and while loops, where control of the terminating condition is performed at the beginning of the loop body. As an example, let us see how the following loop is implemented in assembly:

```
for(i = 0; i < 4; i++, n = n/2)
```

The loop shifts right the variable `n` four times; each shift is equivalent to division by two. We are using register r16 as the loop counter `i` and r17 as the variable `n`.

```
   ldi r17, 0x80   ; initialization
   ldi r16, 0      ; i = 0
next:
   cpi r16, 4      ; if i >= 4...
   bcc exit        ; ...exit loop
   lsr r17         ; r17 = r17 / 2
   inc r16         ; i++
   rjmp next       ; goto next
exit:
```

Check the operation of the `LSR` instruction in the AVR assembly manual. Also note that it is much more effective to perform four individual `LSR` instructions instead of a loop.

Any general purpose register can serve as the loop counter. In practise we would rather avoid the index registers (r24–r31) and the first sixteen registers (r0–r15), which do not implement comparison to a constant. This leaves us with r16–r23.

Assignment 1. In the above code fragment, fill in the necessary declarations (`.cseg` etc). Run the program step by step and note, after each comparison (`cpi`), the value of r16, r17, carry and zero flag. How do you explain these values?

Question 1. In place of the instruction `rjmp`, can we use plain `jmp`? What difference will it make?

The second implementation of a loop corresponds to a do–while structure, where the condition for loop termination is tested at the end of the loop body. In this way, the loop for right shift of r17 will be carried out as follows:

```
   ldi r17, 0x80   ; initialization
   ldi r16, 0      ; i = 0
next:
   lsr r17         ; r17 = r17 / 2
   inc r16         ; i++
```

```
   cpi r16, 4       ; if i < 4...
   brcs next        ; ...goto next
```

The second form, testing at the end of the loop, is slightly simpler and shorter (by a `jmp` instruction). On the other hand it is always executed at least once, which is sometimes undesirable.

Question 2. Instead of the `brcs` instruction can we use `brne`? What will be the difference?

Assignment 2. Run the second program. Is there any difference in the final values of r16, r17, carry, zero flag?

Assignment 3. Modify either of the two loops, so that the counter now runs from `i = 4` to `i = 0`. You will have to to make changes to the loop termination condition.

We may sometimes have the initial and final values of the counter stored in register variables. Depending on the loop structure, any bit of the status register can be tested for loop termination.

2 Arrays and pointers

We can access a memory location using any general purpose register and direct addressing (`LDS` and `STS` instructions). In order to access elements of an array, we need one of the X, Y, Z indices and what is called "indirect" addressing (`LD` and `ST` instructions).

The following code adds the 10 first elements of `array`. We use r16 as loop counter, and r18 as the sum variable. Additionally, r17 is used to hold the current array element to be added to the sum; this is necessary, since addition is performed only on registers. Finally, the X index (r27:r26) is used to access the array elements.

```
   eor r16, r16   ; i = 0
   eor r18, r18   ; sum = 0
   ldi r27, high(array)   ; initialize X
   ldi r26, low(array)
next:
   ld r17, X+     ; r17 = array[i]
   add r18, r17
```

```
inc r16         ; i++
cpi r16, 10     ; if i < 10...
brcs next       ; ...goto next
```

We initialize the pointer with the array address, which is known. The convention used is to place the high–order byte in r27 and the low-order byte in r26 (this is referred to as "little-endian" convention). Note that the `ld` instruction, which moves an array element to r17, subsequently increments the pointer.

Assignment 4. Fill in the necessary declarations (`.cseg` etc) in the above code. You have to declare the array in the data segment; however the assembler does not allow you to initialize it with values. You will have to do that manually, using the memory window in the Atmel Studio. Run the program and write down, only during the first and last pass through the loop, the values of r16, r17, r18, X, and flags after the execution of each loop instruction. Give your remarks in your lab report.

Question 3. What is the largest number of elements that can be added in this way (i.e. the maximum size of the array)?

Assignment 5. Suppose that the array elements are unsigned integers. By summing up these elements in r18, the result may exceed 255, which is the largest unsigned int that may fit in one byte; this case will be signified by the carry flag. Modify your program to store the result in a register pair instead of a single register (16 bits instead of 8). You will need an additional 5 or 6 lines of code. Run your modified program with an array of your choice and verify its operation. Note the data used and the result in your lab report in decimal and hex.

Question 4. Are the two bytes used to store the result in the previous assignment enough? Assume a maximum array size as calculated in question 3. Explain your point briefly.

3 Port programming (digital input–output)

The ATMega328 microcontroller has 23 pins available for digital input–output. These are grouped in three ports: port B (pins PB0–PB7), C (PC0–PC6), and D (PD0–PD7). Each pin can be individually defined to be used as either input or output, and can be read from or written to, independently

from other pins. Of course, more than one pin, or all pins that constitute a port, can be accessed concurrently using a single instruction. Access to each port is provided using three 8-bit special registers (per each port). Port B registers are named DDRB, PORTB, and PINB, and similarly for ports C and D. The function of each pin is programmed by a corresponding bit of these three registers. For pin PBn, where n can be 0 to 7, we have:

DDRB The n-th bit of DDRB determins whether the corresponding pin PBn will be used for input or output. A value of 0 signifies input, while 1 signifies output. DDRB is written by a single STS or OUT instruction, so that all 8 pins of port B are programmed concurrently. DDRB can also be read, although there is little use to that.

PORTB This is a buffer that can be written or read. If bit n of DDRB is 1 (output), then, at each clock cycle, the value of the n-th bit of the PORTB buffer is transferred to the corresponding I/O pin. However, if DDRBn = 0 (input), the buffer contents are unaffected (i.e. they are isolated from the corresponding pin). While DDRBn = 0, writing a logical 1 to PORTBn has the effect of activating the pull–up resistor for this pin; writing a logical 0 disables the pull-up resistor. A good practice for unused pins is to define them as inputs with pull-ups enabled.

PINB This is a read-only double buffer, to which the values of corresponding I/O pins are transferred. PINBn can be read regardless of whether DDRBn is 0 or 1. Due to double buffering, there is a small delay of 0.5 to 1.5 clock cycles for a change in pin PBn to appear in PINBn. Writing a logical 1 to the address of PINBn does not affect the buffer itself, but has the effect of reversing the corresponding bit PORTBn; this happens regardless of the value of DDRBn.

It must be noted that all pins on the chip have alternate functions (analog inputs, interrupt signals, serial communication, reset etc). These pin functions are selected and programmed by other control registers. On the Arduino board, pins PB6 and PB7 are used for an external crystal, which provides the 16 Mhz clock signal; pin PC6 is used for the reset signal; and pins PD0 and PD1 are used for the Rx and Tx serial communication signals, through which the board can be programmed through the USB port. This leaves a maximum of 18 pins for digital I/O (20 in case serial communications is not used), even if no alternate functions are used.

A program can access the PINB, DDRB, and PORTB registers in the following ways:

- Using the `IN` and `OUT` instructions; the corresponding addresses of the above registers in the I/O space are 3, 4, and 5.
- Using the `LDS` and `STS` instructions for direct memory addressing. In this manner the I/O registers are xtreatedd as extra memory locations. Their corresponding addresses in the memory (SRAM) space are 23h, 24h, and 25h. The `LD` and `ST` instructions for indirect addressing can also be used.
- If it desired to set or reset a single bit while leaving the rest as are, instructions `SBI` and `CBI` can be used instead of `OUT`.
- Finally, writing a logical 1 at the n-th bit of the PINB register is a convenient way to reverce the value of PORTBn.

Example 1 The following code sets pin PB5 as output and writes a logical 1 to the pin. The rest port B pins are defined as output with pull–up resistance enabled. We use a `NOP` instruction to view the results in the simulator; this is not necessary when we download the program to the Arduino board.

```
start:
   ldi r16, 0b00100000
   out DDRB, r16
   ldi r16, 0b11111111
   out PORTB, r16
   nop
rjmp start
```

Assignment 6. Run the above code in the simulator and watch the I/O register window. Note down which register is affected by each instruction and provide an explanation for this. When does the pin PB5 value change?

Also note, using the disassemble option, the machine language form of the `OUT` instructions. Look in the 328 manual (registers section) for all port register addresses in I/O space.

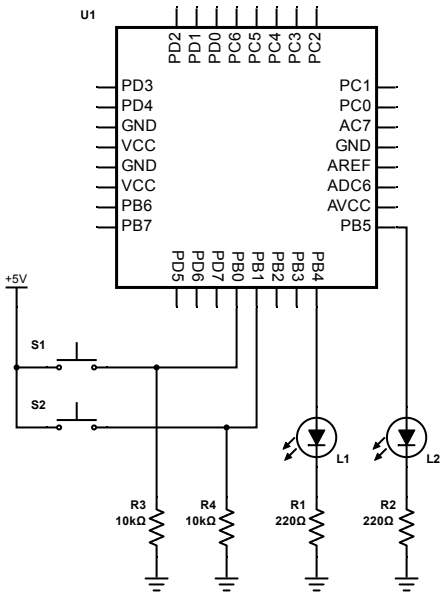

Figure 1: Example 2 circuitry

Example 2 We want to use pins PB4 and PB5 as outputs to some external devices. Pins PB0 and PB1 will be used as control inputs; their values will determine what the output values will be. In this simple example we will use two LEDs as output devices and two push–button or DIP switches as inputs. The rest port B pins will not be used; we shall define them as inputs with enabled pull–up resistors. In this way we avoid a large unwanted current that would damage the chip in case, for example, we inadvertently connect a pin to the power source or ground.

The LEDs can be connected either to the power source with pull–up resistors, or to the ground with pull–down resistors (the latter is shown in Fig. 1). In the former case the LEDs will be lit when a logical 0 is written to the corresponding pin. The switches also need pull–up or pull–down resistors to avoid excessive current. Reasonable values are 200Ω–1kΩ for the LEDs (using smaller resistance values will make the LEDs shine brighter, but also prone to burning out) and around 10kΩ for the switches.

```
start:
   ldi r16, 0b00110000  ; PB5 & PB4 outputs
```

```
   out DDRB, r16
   ldi r16, 0b11111100  ; LEDs on & enable pullups
   out PORTB, r16
```

Suppose that we would like the LEDs to be off initially and that each switch control a corresponding LED (on and off). The simplest thing to do is transfer the value of bit PB0 to PB4, and bit PB1 to PB5. We can manage this by using a temporary register and shifting its contents four times to the left, so that the values of bits 0 and 1 move to bits 4 and 5, respectively.

```
repeat:
   in r16, PINB
   andi r16, 0b00000011 ; Keep only b0 and b1
   lsl r16
   lsl r16
   lsl r16
   lsl r16              ; 000000xy ==> 00xy0000
   out PORTB, r16
   rjmp repeat
```

The above program works in this special case because of the particular mapping of input and output devices to pins of port B. A more general approach is to check each input pin in turn, and perform an action according to the pin value. This is the polling method that we have seen in the lectures.

```
repeat:
   in r16, PINB
   andi r16, 0b00000001 ; Check value of PB0
   brne pb0eq1
   cbi PORTB, 4         ; If PB0 = 0, set PB4 = 0
   rjmp chkpb1
pb0eq1:
   sbi PORTB, 4         ; If PB0 = 1, set PB4 = 1
chkpb1:
   in r16, PINB
   andi r16, 0b00000010 ; Check value of PB1
   brne pb1eq1
```

```
   cbi PORTB, 5          ; If PB1 = 0, set PB5 = 0
   rjmp repeat
pb1eq1:
   sbi PORTB, 5          ; If PB1 = 1, set PB5 = 1
   rjmp repeat
```

Assignment 7. Run the above two programs step-by-step in the simulator of the Atmel Studio and make certain that you understand their function. See the operation of `sbi, cbi` instructions in the AVR Assembly manual.

Assignment 8. Download each program to the Arduino board and verify its operation. in order to download the assembled (object) code you will need the `avrdude` software. At the "external tools" menu of Atmel Studio fill in the command `avrdude` (give the full pathname) and the following parameters (again use full pathname for `avrdude.conf`):

```
-C "...\avrdude.conf" -p atmega328p -c arduino -P COM3
-b 115200 -U flash:w:"$(ProjectDir)Debug\$(TargetName).hex":i
```

Assignment 9. Modify your code to reverse the logic, i.e. have the LEDs on initially and switch them off using the switch. Verify your program on the Arduino board.

Assignment 10. Modify your code, so that each push-button or switch change the state of the corresponding LED from on to off and vice-versa. That is, having input PB0 = 1 will reverse the output PB4. This will require some 5-6 lines of extra code. You will need an extra register to store the current output value. Try your code first in the simulator. Don't be worried if the LEDs occasionally do not seem to respond to the push-buttons; proper operation will require a delay routine for "debouncing".