Algirdas BAŠKYS

MICROCONTROLLERS

Project No
VP1-2.2-ŠMM-07-K-01-047

The Essential Renewal of
Undergraduates Study Programs
of VGTU Electronics Faculty

Vilnius "Technika" 2012

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Algirdas BAŠKYS

MICROCONTROLLERS

A Laboratory Manual

Vilnius "Technika" 2012

A. Baškys. Microcontrollers: A Laboratory Manual. Vilnius: Technika, 2012. 121 p. [3,93 author's sheets, 2012 06 04].

The knowledge about the features and programming of 8 bit mid-range PIC microcontrollers are presented. The architecture, memories structure, ports and instruction set of PIC16F84A microcontroller are analyzed. The laboratory works tasks dedicated to PIC16F84A microcontroller programming are given. The edition is dedicated to bachelor students studying Computer Engineering and Electronics Engineering study programmes. Science area: Technological sciences (T000), science field: Electronics and Electrical Engineering (01T).

The publication has been recommended by the Study Committee of VGTU Electronics Faculty.

Reviewed by:
Assoc. Prof Dr Gediminas Gražulevičius, VGTU Department of Computer Engineering,
Dr Nerijus Paulauskas, VGTU Department of Computer Engineering

This publication has been produced with the financial assistance of Europe Social Fund and VGTU (Project No VP1-2.2-ŠMM-07-K-01-047). The book is a part of the project "The Essential Renewal of Undergraduates Study Programs of VGTU Electronics Faculty".

This is an educational methodology book, No 1339-S, issued by VGTU Press TECHNIKA *http://leidykla.vgtu.lt*

Language editor *Dalia Blažinskaitė*
Typesetter *Laura Petrauskienė*

eISBN 978-609-457-156-5
doi:10.3846/1339-S

CONTENTS

1. THE CONCEPT AND CLASSIFICATION OF MICROCONTROLLERS

A microcontroller (MC) contains the main computer components: processor, program and data memories, input/output interfaces. Therefore, it can be named single-chip computer. The term "Microcontroller" tells that this device is developed to control objects and processes. Because of this, the chip of MC contains various additional components as timers, A/D and D/A converters, voltage references, PWM generators, serial UART and USB interfaces etc. Constant improvement of MC parameters and low price allows penetrating the MCs into the various fields of human activity. We can find the microcontrollers in most of the devices that control, measure, calculate, or display information. As an example, the modern automobile can include up to 50 MCs. To interface with the environment, the additional components as various logical voltage level matching circuits, sensors, displays, connectors, switches, LEDs and so on should be used with the MC. Such a system, which includes MC or several MCs and additional components often, is named Microcomputer. Microcomputer in contrast to personal computer is very specialized developed for concrete purpose, e.g. control automobile engine or brakes or the hard disk drive of personal computer. Since such microcomputers are embedded in other machinery, usually they are called embedded systems or embedded computers. The variety of the embedded computers is extremely high, therefore, there are lot of laboratories and firms that develop the embedded computers. The time comes when most of electronic devices will be based on the MCs, i.e. the engineers that develop or provide the service of electronics should have not only good knowledge of electronics hardware design but good knowledge of creating of MC programs, which usually are called firmware, as well.

MCs are classified by architecture, instruction set, MC ideology and producer.

1. Classification of MCs by architecture. There are two MC architectures Fon Neiman and Harvard. The MC developed using Fon Neiman architecture has common memory for storage of data and programs and, as a consequence, the common bus for transferring of instructions addresses and data.

Harvard architecture differs from Fon Neiman architecture. It has separate memory units for program and data storage and separate busses for transferring of instructions addresses and data. Harvard architecture allows us to reach higher data transfer speed. The single instruction can be executed during one machine cycle using the MC based on the Harvard architecture. Most MC families are created using Harvard architecture.

2. Classification of MCs by instruction set. The MCs are divided into two groups by instruction set. There are MCs that belong to the reduced instruction set computer (RISC) group and MC that belong to the complex instruction set computer (CISC) group. Majority of MC are based on the RISC ideology. CISC ideology is mostly used in microprocessors.

3. Classification of MCs by ideology. According to ideology MC are distributed into the families. The most popular families of 8-bit MCs are:

1. 8051 family (*Intel* ideology);
2. 68HC05 family (*Motorola* ideology);
3. AVR family (*Atmel* ideology);
4. PIC family (*Microchip* ideology).

8051 family. The MCs of this family are developed using Harvard architecture. They belong to RISC MCs. The 8051 family MCs are manufactured by the such a firms as: Intel, Atmel, Dallas Semiconductor, Philips, Siemens, ISIS (*Integrated Silicon Solutions*).

68HC05 family. This MC family is known under the HC05, HC08 and HC11 titles. The producer of these MCs is Motorola. They are developed using Fon Neiman architecture. The MCs of this family belong to CISC type.

AVR family. The MC of this family are developed using Harvard architecture. They belong to RISC type. The main advantage of AVR MCs is high speed. These MCs are able to execute one instruction during one clock cycle. The producer of AVR MCs is Atmel.

PIC family. The MCs of this family belong to Harvard architecture. They are RISC type. The producer of PIC MCs is Microchip. This firm, which official was named Arizona Microchip Technology, was found in 1988 years. The 8-bit PIC MCs are the most popular and have the biggest market among the 8-bit MCs.

2. THE CLASIFFICATION AND CHARACTERISTIC OF 8-BIT PIC MICROCONTROLLERS

The laboratory works of MCs course are dedicated to teach the students creating of microcontroller programs using assembler. The variety of MCs is wide, thus it is important to choose the suitable family of MCs, which could be used as the basis for studies. Taking advantage of our experience and experience of other universities [1], the 8-bit Mid-range architecture family of PIC (Peripheral Interface Controller) MCs developed and produced by Microchip company [2] was chosen. It is one of the most popular and easy-to-use MC families. Therefore, the MCs of this family are good for the studies. They are characterized by rather simple but well developed architecture and a small but powerful instruction set. The family has a wide range of representatives with various features and has a large Internet based community.

The 8-bit MCs produced by Microchip are divided into three groups:

1. Baseline 8-bit architecture PIC MCs. These MCs have 12 bit 33 instruction set, (384-3,5 K) x 12 bits program memory, 6-44 pins. Operation speed is up to 5 MIPS (Millions Instruction Per Second). Some MCs include ADCs and comparators for processing of the analogue signals. Baseline 8-bit architecture includes PIC10FXXX, PIC12FXXX, PIC16F5XX and PIC16C5X MCs.

2. Mid-Range 8-bit architecture PIC MCs. The representatives of this MC group have 14 bits 35 instruction set, (896-14 K) x 14 bits program memory, 14-68 pins. Operation speed is up to 5 MIPS. Many of MCs include ADCs and comparators. There are MC which have I2C, SPI, USB and USART interfaces. This group includes PIC16FXXX, PIC16CXXX and PIC16CRXXX MCs.

3. High Performance 8-bit architecture MCs. These MCs have 16 bits 77 instruction set, (8-128)K x 16 bits program memory, 18-100 pins. Operation speed is up to 16 MIPS (Millions Instruction Per Second). Most of MCs include ADCs, comparators, voltage references, operational amplifiers for processing of the analogue signals. Some MC have DACs for output of analogue signal. Most of them support I2C, SPI, USB and USART interface standards. This category includes PIC18FXXX and PIC18FXXJXX MCs.

There are about 70 types of 8-bit mid-range PIC MCs. They can be divided into three groups by type of the program memory: MCs with Flash, MCs with ROM program memory and OPT (*One-Time-Programmable*) MCs. The Instruction set of these MC has only 35 instructions. These MCs have 2–7 ports with number of input/output pins in range from 12 to 53. Many of MC have 8–12 bit ADC. Some of MCs have 1–2 voltage comparators. The maximal clock frequency is 20 MHz, however, there are MC with clock frequency up to 40 MHz. Some of MCs include internal clock resonator.

The 8-bit mid-range PIC MCs architecture, data memory, ports, Reset organization, principles of timer/counter, instruction set and assembly directives are analyzed in this course studding MC PIC16F84A, which is one of the most popular devices of this MC group.

3. MICROCONTROLLER PIC16F84A

3.1. Characteristics

The main characteristics of MC PIC16F84A are as follows:

- 18 pins PDIP case, designed for through hole mounting, mini SOIC or SSOP case, which are designed for surface mounting. Also the MC chips can be delivered;
- 2 input/output ports (A and B), which include 13 input/output pins.
- 35 instruction set, the execution of most instructions takes one machine cycle, which corresponds to 4 clock frequency periods.
- 1024 x 14 bit Flash program memory;
- 68 x 8 bit static RAM and 68 x 8 EEPROM data memories;
- 13 bit eight level hardware stack;
- 8 bit timer/counter with 8 bit programmable prescaler;
- 10 thousands writing/erasing cycles for the program memory;
- 10 millions of data writing/erasing cycles for the EEPROM data memory;
- Program code protection feature;
- Watch dog;
- Power saving mode (SLEEP mode);
- Serial programming interface (*In-Circuit Serial Programming – ICSP*) allows programming of the MC without removing the device form PCB.

3.2. The architecture

The block diagram of PIC16F84A MC is presented in Fig. 3.1. MC is developed using Harvard architecture. We can see that it has separate program memory and data memory. There are two data memories. One of them is static RAM (SRAM), which consists of file registers. There are two types of file register: Special Function Registers (SFR) and General Purpose Registers (GPR). The file reg-

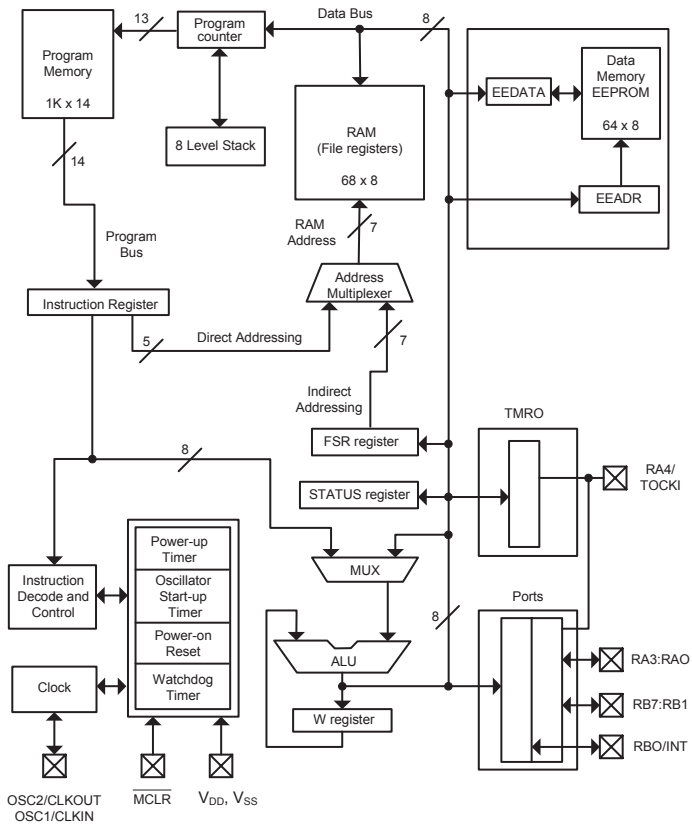


Fig. 3.1. Block diagram of PIC16F84A microcontroller

isters are the 8 bit memory cells that provide the link between the MC hardware and software. The SFRs control the operating of MC functional blocks, while GPR are used to store the values of variables and constants. In this type of memory the number of writing/erasing cycles for the SRAM memory is unlimited. It operates at the same speed as ALU of MC. However, the data stored in the registers of SRAM is lost if the power supply of MC is switched of. Therefore, the EEPROM memory, which belongs to non-volatile electronic memory, must be used for storage of data as well. However, the EEPROM memory is relatively slow and has limited number of writing/erasing cycles. Because of this, it should be employed only to store the data that must be saved when power supply of MC is switched off.

The width of data bus is 8 and program bus 14 bit (line). Program counter generates the address of instruction in the program memory, which has to be executed. The address bus is of 13-bit, although PIC16F84A program memory volume is of 1024 x 14 bits, it allows to address directly up to 8 K x 14 bit memory volume.

MC has 8 x 13 bit stack. The stack can store up to 8 program memory addresses, which can be called return addresses that are needed in the situations when the execution of subroutine or interrupt program is over and the MC should continue the execution of the main program again.

The purpose of Timer/Counter is to measure the time and to calculate the events. It operates in Timer or Counter mode and calculates the number of received pulses. The pulses can be transferred from the clock of MC (Timer mode, frequency of pulses is equal to 1/4 of MC clock frequency) or trough I/O pin RA4/TOCK1 of MC (Counter mode). Every pulse increases the contents of 8-bit TMR0 register by one. Additionally, the 8-bit prescaler can be used with the Timer/Counter. The prescaler is programmable and can divide the frequency of pulses by ratio 1:2, up to 1:256.

There are four PIC16F84A MC operating modes determined by the type of resonator:

- LP – crystal resonator, clock frequency 32–200 KHz;
- XT– crystal resonator, clock frequency 0.455–4 MHz;
- HS – crystal resonator, clock frequency 8–10 MHz;
- RC – RC circuit is used as resonator.

The RC resonator option saves the cost while the LP crystal option saves power. Configuration bits are used to select the various options. Crystal or ceramic resonators are connected to MC through OSC1/CLCIN and OSC2/CLCOUT pins. The external clock source instead of internal MC oscillator can be used. The signal of the external clock has to be connected to OSC1/CLCIN pin.

Power-on Reset block generates reset impulse, when supply voltage V_{DD} reaches (1.2–1.7) V after it has been turned on. Power-up timer holds PIC16F84A MC in the Reset mode for 72 ms after the power is switched on. If during that time transition processes of supply voltages are over and potential at MC terminals are in the steady state, it is not necessary to use any external RC circuits to increase the duration of the Reset mode. Because of this, this block frequently allows withdraw external RC Reset circuits. In this case it is enough to connect MCLR pin directly, or by resistor to V_{DD} (positive supply source pole).

Oscillator Start-up timer holds PIC16F84A in Reset mode till clock frequency becomes steady.

PIC16F84A MC has Watchdog, which is intended to observe the execution of the MC program. If the execution process fails, it generates the Reset signal for the MC to set the data stored in the SFR to initial values and to return the operating of MC to the beginning of the program. Watchdog timer has its own RC resonator. The Watchdog can be activated or not during the MC configuration.

PIC16F84A has 8-bit arithmetic-logic unit (ALU), which executes arithmetical and logic operations and controls MC operating. Its status is represented by the STATUS register contents. Very important role in PIC16F84A MC performs register W. It is used for data transfer to other registers.

Input/Output ports PortA and PortB include pins RA0-RA4 and RB0-RB7, which are designed to receive or send the digital signals from (to) other devices.

3.3. The program memory

The program memory of PIC16F84A MC is represented in Fig. 3.2.

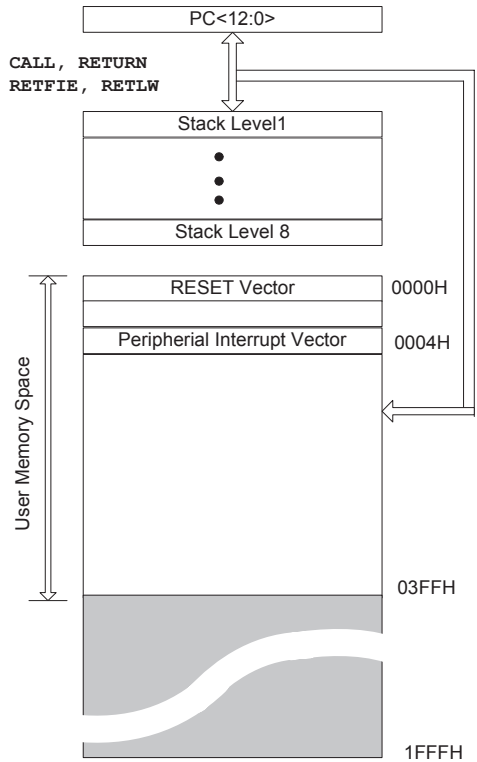


Fig. 3.2. Program memory

The purpose of the program memory is to store the MC program. There are located appropriate number of 14 bit Flesh memory cells. The PIC16F84A MC has 1024 such a memory cells, because of this the 1024 x14 bit words can be stored. The program memory includes two special memory cells, which are located under the addresses 0000H (it is called Reset Vector) and 0004H (it is called Interrupt Vector). The MC is developed in such a way that after the Reset the microcontroller starts the execution of program from the Reset Vector and after the interrupt event – from Interrupt Vector.

3.4. Data memory

Static RAM (Fig. 3.3) is divided into two parts. In the first part (the top part in Fig. 3.3) are stored Special Function Registers (SFR), in the second one – General Purpose Registers (GPR) (the bottom part in Fig. 3.3).

Static RAM has two memory banks: Bank 0 and Bank 1. The 12 addresses of every Bank are reserved for Special Function Registers, remaining are left for General Purpose Registers. PIC16F84A has 68 General Purpose Registers and all of them are located in Bank 0 under addresses 0Ch to 4Fh. If there is a call on General Purpose Registers, which are in the 1-st Bank (addresses 8Ch-CFh), which physically does not exist in PIC16F84A, this call is redirected to appropriate registers in Bank 0. For example, if there is a call on 0Ch or 8Ch addresses, the access is made to the same 0Ch register located in 0 Bank.

Special Function Registers perform important role in MC. They are used to control MC operation and act as the bridge between MC software and hardware. The information about Special Function Registers is given in Tables 3.1 and 3.2. The detailed information about some Special Function Registers is given bellow.

STATUS register is mostly used register. It includes the data memory bank selection bit and the flag bits that reflect the status of

Arithmetical logical Unit (ALU). The purpose of STATUS register bits is as follows:

RP0 (5th bit) – SRAM data memory bank selection bit. If this bit is set to 0, the MC has access to contents of Bank 0, if it is set to "1" – to Bank 1. After Reset this bit is set to 0, i.e. by default MC has access to Bank 0.

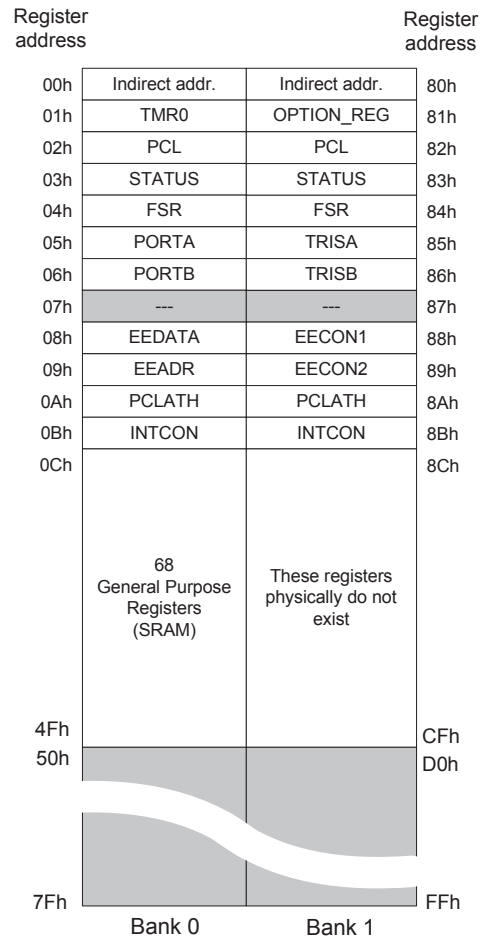


Fig. 3.3. SRAM data memory

T0 (4th bit) and PD (3rd bit) show us what event triggered the Reset. The exact functions of T0 and PD are: T0 is WDT (Watchdog timer) trigger bit. It is set to “1” if the power supply is switched on or instructions CLRWDT, SLEEP where executed. This bit is set to “0” when Watchdog timer register overflows. PD bit is set to “1” when the power supply is switched or when the instruction CLR-WDT is executed. PD bit is set to “0” if the instruction “SLEEP” is executed. The exact functions of bits Z, DC and C that indicates the status of ALU are:

Z – Zero bit indicates (It is set to “1”) whether the result during the execution of arithmetic or logic instruction was zero. If Z bit remains “0” – the result of execution of arithmetic or logic instruction is not zero.

DC indicates (It is set to “1”) whether the less significant nibble of register overflows during the execution of arithmetic instruction and a carry-out from the 4th low order bit of the result occurred.

C indicates (It is set to “1”) whether the register overflows during the execution of arithmetic instruction and a carry-out from the most significant bit of the result occurred.

Bits RP1 (6th bit) and IRP (7th bit) are not used in PIC16F84A microcontroller.

OPTION_REG register. The purpose of register bits is as follows:

RBPU (7th bit) enables or disables Pull-up resistors between port B output pins and supply Vss. If bit is set to “1” – PORTB pull-ups are disabled, if it is set to “0” – PORTB pull-ups are enabled.

INTEDG (6th bit) – Interrupt Edge Select bit. “1” – Interrupt on rising edge of RB0/INT pin, “0” – Interrupt on falling edge of RB0/INT pin.

TOCS (5th bit) – Timer/Counter (TMR0) Clock Source Select bit. “1” – Transition on RA4/T0CK1 pin, “0” – Internal instruction cycle clock.

TOSE (4th bit) – Timer/Counter (TMR0) Source Edge Select bit. “1” – Increment on high-to-low transition on RA4/T0CKI pin, “0” – Increment on low-to-high transition on RA4/T0CKI pin.

PSA (3rd bit) – Prescaler Assignment bit. “1” – prescaler is assigned to the Watchdog (WDT), “0” – Prescaler is assigned to Timer/Counter (TMR0)TMR0.

PS0-PS2 (0-2 bits) – Prescaler Rate Select bits.

INTCON register. It contains enable bits for all interrupt sources. Purpose of INTCON register bits:

GIE (7th bit) – Global Interrupt Enable bit. “1” – enables all interrupts, “0” – disables all interrupts;

EEIE (6th bit) – Write to EEPROM data memory complete Interrupt Enable bit. “1” – enables the writing complete interrupt, “0” – disables the write complete interrupt;

T0IE (5th bit) – Timer/Counter register TMR0 Overflow Interrupt Enable bit. “1” – enables the TMR0 interrupt, “0” – disables the TMR0 interrupt;

INTE (4th bit) – RB0/INT Interrupt Enable bit. “1” – enables the RB0/INT interrupt, “0” – disables the RB0/INT interrupt;

RBIE (3rd bit) – RB Port Change Interrupt Enable bit. “1” – enables the RB port change interrupt,“0” – disables the RB port change interrupt.

T0IF (2nd bit) – Timer/Counter register TMR0 Overflow Interrupt Flag bit. “1” – TMR0 has overflowed (must be cleared in software), “0” – TMR0 did not overflow;

INTF (1st bit) – RB0/INT Interrupt Flag bit. “1” – The RB0/INT interrupt occurred, “0” – The RB0/INT interrupt did not occur;

RBIF (0 bit) – RB Port Change Interrupt Flag bit. “1” – when at least on the one of the RB7-RB4 pins the potential has changed from the value that corresponds to “0” to value that corresponds to “1” or vice versa (must be cleared in software), “0” – none of the RB7-RB4 pins have changed state.

PCL register. The program counter (PC) specifies the address of the instruction to fetch for execution. The 8 lowest bytes are called the PCL register. This register is readable and writable. The high byte is called the PCH register and is not directly readable or writable. All updates to the PCH register go through the PCLATH register.

INDF and SFR registers. Both of them are used for indirect addressing. INDF physically does not exist. Accessing this register actually means that user is accessing the virtual address which is stored in the SFR register.

Table 3.1. Special Function Registers located in Bank1

Address	Name	Bit number							
		7	6	5	4	3	2	1	0
00h	INDF	For the indirect addressing using FSR register							
01h	TMRO	8-bit Real-Time Clock/Counter							
02h	PCL	8 lower bits of Program Counter							
03h	STATUS	IRP	RP1	RP0	TO	PD	Z	D	C
04h	FSR	Indirect data memory address pointer							
05h	PORTA	-	-	-	R4/ TOCK	RA3	RA2	RA1	RA0
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
07h	-	Unimplemented location							
08h	EEDATA	EEPROM Data Register							
09h	EEADR	EEPROM Address Register							
0Ah	PCLATH	-	-	-	Write buffer for upper 5 bits of the PC				
0Bh	INTCON	GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF

Table 3.2. Special Function Registers located in Bank1

Address	Name	Bit number							
		7	6	5	4	3	2	1	0
80h	INDF	For the indirect addressing using FSR register							
81h	OPTION	RBPU	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0
82h	PCL	8 lower bits of Program Counter							
83h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
84h	FSR	Indirect Data Memory Address Pointer 0							
85h	TRISA	-	-	-	PORTA Data Direction Register				
86h	TRISB	PORTB Data Direction Register							
87h	-	Unimplemented location, read as '0'							
88h	EECON1	-	-	-	EEIF	WRERR	WREN	WR	RD
89h	EECON2	EEPROM Control Register 2 (not a physical register)							
8Ah	PCLATH	-	-	-	Write buffer for upper 5 bits of the PC				
8Bh	INTCON	GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF

EEDATA register is used for storing data that will be stored to or read from EEPROM memory cell.

EEADR register is used for storing the address of EEPROM memory cell, in which the data will be written or from which will be read.

EECON1 register is used to control the operating of EEPROM data memory.

TRISA, TRISB registers are used to set the pins of ports A and B as inputs or outputs. "1" – sets the appropriate pin to act as input and "0" – sets the appropriate pin to act as output.

PORTA, PORTB registers are used to store data that are sent to corresponding ports when appropriate port pins act as outputs or to store the data that are red from ports when port pins operate as inputs.

3.5. Input/Output ports

There are four different circuits that serve the port pins. The circuit diagram that is used for the pins RA0–RA3 of port A is shown in Fig. 3.4. It has three latches: data latch where is stored a bit, which is sent to pin when it works as output; TRIS latch where is stored bit, which sets pin to input or output mode; latch where is stored the pin signal, when pin is set to input mode. When pin is set as an output, the output voltage is formed using push-pull CMOS transistor stage, which can supply up to 25mA current. When pin is set to input mode, TRIS latch closes the push-pull stage CMOS transistors, so pin operating as input is characterized by the high impedance. The MC operates with the TTL voltage levels, i.e. the "0" corresponds 0 to 0.8 V and "1" corresponds 2.4 to 5 V.

The Input/Output pin RA4/TOCK includes the additional function – it is used for the receiving of the external pulses when Timer/Counter operates in the Counter mode. The block diagram of the circuit that serves the RA4 pin is given in the Fig. 3.5. It differs from the

circuit that is used for the pins RA0–RA3 by the fact that the Schmitt trigger is employed for the pin signal reading. The transfer characteristic of Schmitt trigger is characterized by the hysteresis, because of this using of such circuit decreases the sensitivity to electromagnetic disturbances. Additionally, this pin is served by CMOS transistor with open drain in the output mode. Therefore, the increased voltage that corresponds to "1" can be provided by this pin. It can reach 8.5 V, however for this purpose an external resistor and external supply source have to be used. The increased voltage of "1" can be used, where it's not enough standard TTL signal, e.g. for supply of relay.

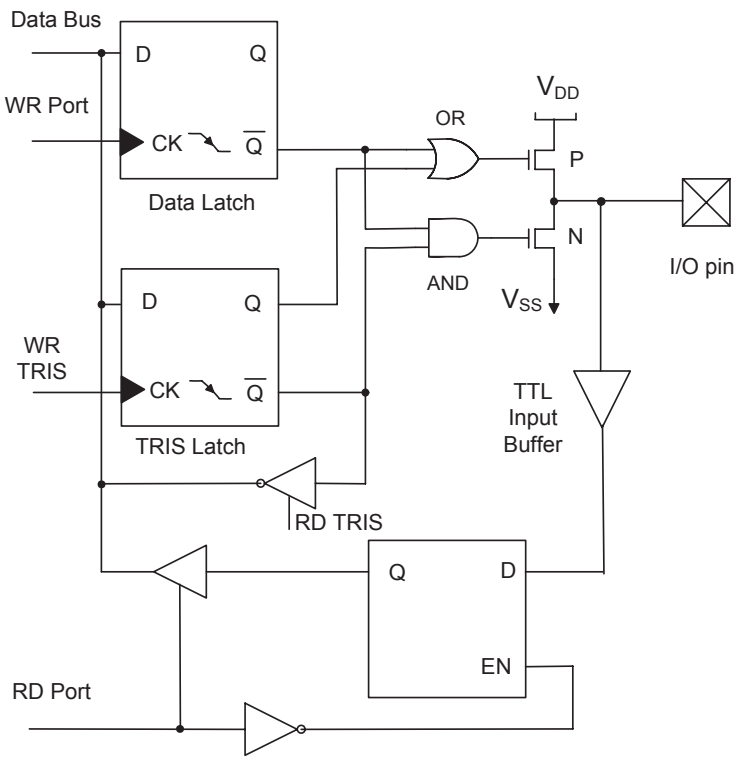


Fig. 3.4. Block diagram of RA0–RA3 pins

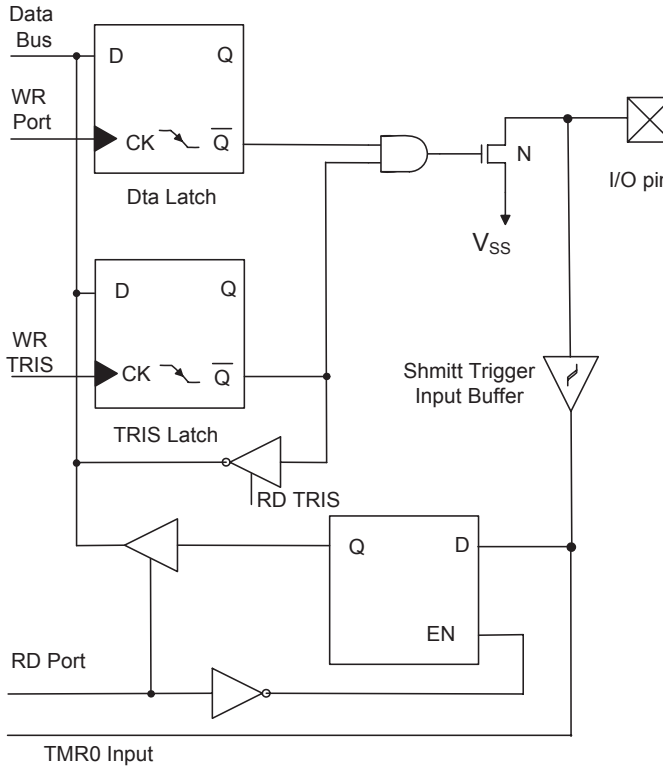


Fig. 3.5. Block diagram of RA4 pin

According the requirements to MC PIC16F84A, the external supply source voltage should be not higher then 8.5 V.

The block diagrams of circuits that are used for port B pins are given in the Figs. 3.6 and 3.7. The CMOS transistors that work as the Weak Pull-up resistors are connected between port B pins and positive terminal V_{DD} of supply. The resistance of the resistors is approximately 20 kΩ. They can be employed in the situation when pins operate in the input mode. The external resistor is unnecessary in such a case. Since CMOS transistors are characterized by the high impedance, the connection of the Weak Pull-up resistors

increases the Electromagnetic disturbance resistance of the MC inputs. The RBPU bit of OPTION_REG register is used to activate the Weak Pull-up resistors. To connect resistors the “0” should be sent to RBPU bit. There is no possibility to connect Weak Pull-up resistor separately to the one pin. The Weak Pull-up resistors are disconnected automatically if pins are set to output mode.

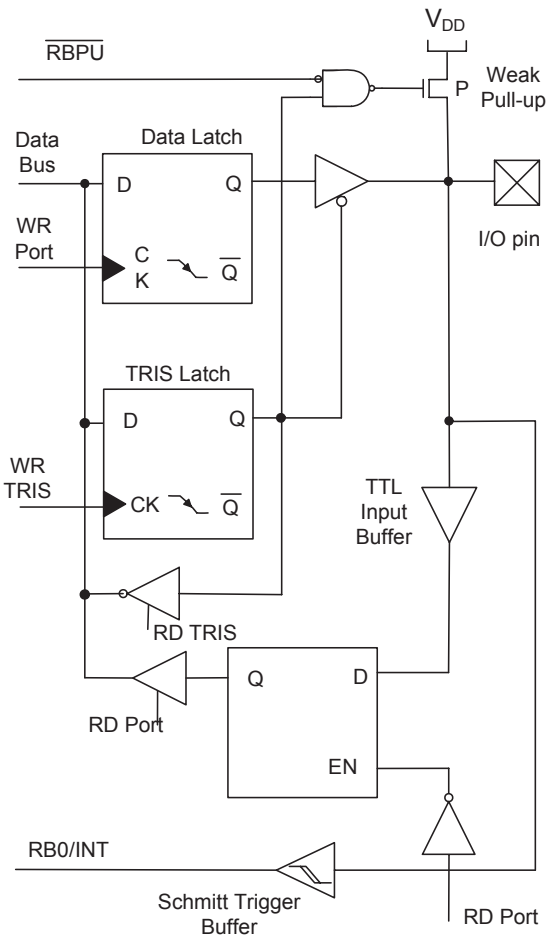


Fig. 3.6. Block diagram of RB0–RB3 pin

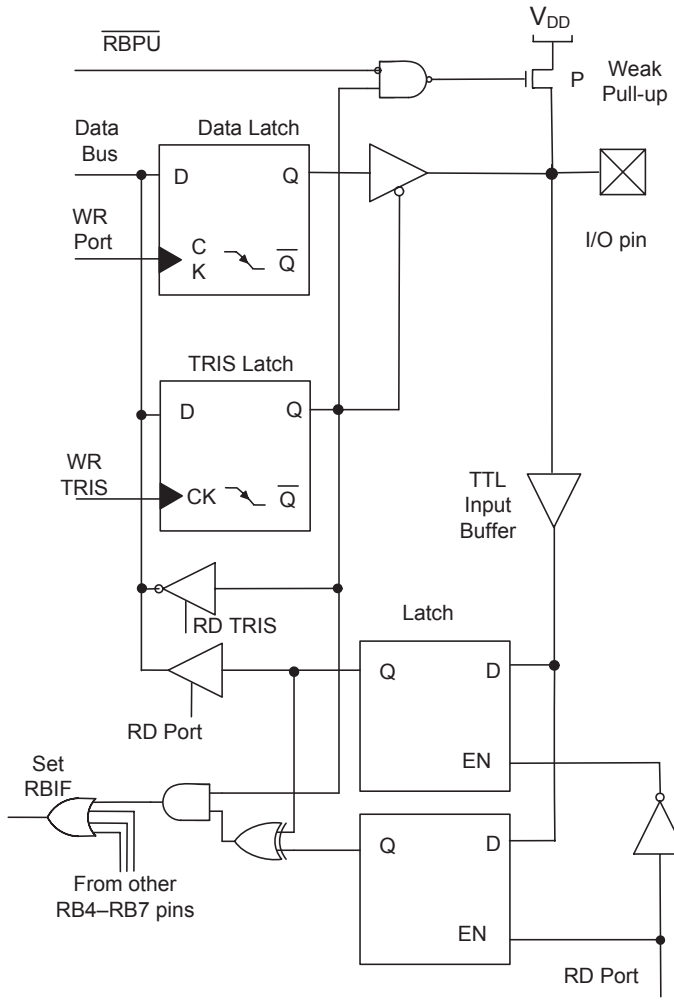


Fig. 3.7. Block diagram of RB4–RB7 pins

Pin RB0/INT can be used for external interrupt. It responds to the pulse edge. If "1" is sent to the 6th bit of the OPTION_REG register, the interrupt is triggered by the rising edge, if "0" – by the

falling one. The Shmitt trigger is used for the reading of the interrupt signal.

When pins operate as outputs, the output voltage is delivered to the port B pins using the push-pull CMOS transistor stages. These stages are the same as these used for the port A pins RA0-RA3 (Fig. 3.4).

The RB4-RB7 pins can be used as external interrupt inputs. The interrupt is triggered by the potential change (by change from "0" to "1" or vice versa). During the every MC machine cycle the potential on the pin is compared with the potential that was before. If the potential level has changed in the one of the pins, the interrupt occurs. If all RB4-RB7 pins are used for interrupt, there is no possibility to estimate, which pin triggered the interrupt.

4. INSTRUCTION SET OF 8-BIT MID-RANGE PIC MICROCONTROLLERS

Instructions are the means that allow human (programmer) to form the job for MC. They are as some language for the communication between the human and a MC. 8-bit mid-range PIC MCs including MC PIC16F84A, have 35 instruction set, so it is assigned to the RISC MC group. All instructions are grouped into three categories:

1. Byte-oriented instructions;
2. Bit-oriented instructions;
3. Literal and control instructions.

Every 8-bit mid-range PIC MC instruction is 14 bit word, divided into the operation code (OPCODE), which specifies instruction type that shows what has to be done, and one or more operands. Operand specifies the operation of the instruction. Letters that present the name of the instruction are called mnemonic. The general form of the instruction in assembler programs is as follows:

Instruction mnemonic operand A, operand B

It should be stressed that part of the instructions does not include operand B and there are some instructions that do not include any operand. For example, in the Byte-oriented instructions operand A presents data or address (name) of the register where the data are stored, Operand B shows the place where the result after the instruction execution has to be saved.

The lists of instructions are presented in the Tables 4.1, 4.2 and 4.3. The following designations are used in the instructions:

- for the byte-oriented instructions, “f”” represents a register designator and “d” represents a destination designator. The register designator specifies which register contents is to be used by the instruction.

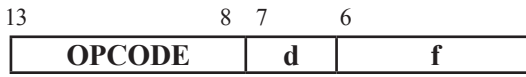
The destination designator “d” specifies where the result of the operation is to be placed. If “d” is zero, the result is placed in the W register. If “d” is one, the result is placed in the register specified in the instruction.

- for the bit-oriented instructions, “b” represents a bit field designator, which selects the number of the bit affected by the operation, while “f” represents the number of the register, in which the bit is located.

- for the literal and control operations, “k” represents an eight or eleven bit constant or literal value.

The formats of the instructions are presented in Fig. 4.1.

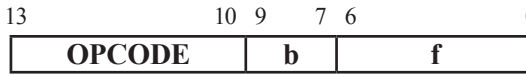
Byte-oriented instructions



d – shows where the result is to be placed

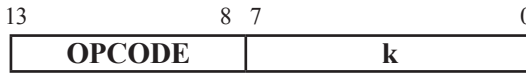
7-bit register address

Bit-oriented instructions



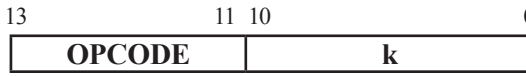
b – 3-bit bit address, 7-bit register address

Literal and Control instructions



k – 8 bit constant, under which the instruction is executed

CALL and GOTO instructions



k – 11 bit constant, which corresponds to the appropriate address in the program memory.

Fig. 4.1. The instruction formats

Table 4.1. Byte-oriented instructions

Mnemonic, Operands	Instruction description	Flag bits affected
ADDWF f, d	Add W and f	C, DC, Z
ANDWF f, d	AND W with f	Z
CLRF f	Clear f	Z
CLRW –	Clear W	Z
COMF f, d	Complement f contents	Z
DECF f, d	Decrement f	Z
DECFSZ f, d	Decrement f, skip if 0	
INCF f, d	Increment f	Z
INCFSZ f, d	Increment f, skip if 0	
IORWF f, d	Inclusive OR W with f	Z
MOVF f, d	Move f	Z
MOVWF f	Move W to f	
NOP –	No Operation	
RLF f, d	Rotate Left f through Carry	C
RRF f, d	Rotate Right f through Carry	C
SUBWF f, d	Subtract W from f	C, DC, Z
SWAPF f, d	Swap nibbles in f	
XORWF f, d	Exclusive OR W with	Z

Table 4.2. Bit-oriented instructions

Mnemonic, Operands	Instruction description
BCF f, b	Bit Clear f
BSF f, b	Bit Set f
BTFSC f, b	Bit Test f, Skip if Clear
BTFSS f, b	Bit Test f, Skip if Set

Table 4.3. Literal and Control instructions

Mnemonic, Operands	Instruction description	Flag bits affected
ADDLW k	Add literal and W	C, DC, Z
ANDLW k	AND literal with W	Z
CALL k	Call subroutine	
CLRWDT –	Clear Watchdog Timer	TO, PD
GOTO k	Go to address	
IORLW k	Inclusive OR literal with W	Z
MOVLW k	Move literal to W	
RETFIE –	Return from interrupt	
RETLW k	Return with literal in W	
RETURN –	Return from Subroutine	
SLEEP –	Go into standby mode	TO, PD
SUBLW k	Subtract W from literal	C, DC, Z
XORLW k	Exclusive OR literal with W	Z

All instructions are executed in one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction. In these cases, the execution takes two instruction cycles with the second cycle executed as an NOP. One instruction cycle consists of four clock oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 μs. However, there are control instructions (CALL, GOTO, RETFIE, RETLW and RETURN) that are executed during the 2 machine cycles. The conditional instructions DECFSZ, INCFSZ, BTFSC and BTFSS are executed during the one machine cycle. However, when the requirements of the conditions are met and the instruction makes to skip the next instruction, the execution takes 2 machine cycles.

5. THE INTEGRATED DEVELOPMENT ENVIRONMENT MPLAB IDE

The MPLAB IDE software, which is named Integrated Development Environment, is developed for the creating, editing, adjusting and machine code generation using Windows operating system. MPLAB IDE software is developed by Microchip firm.

The MPLAB IDE is used for the following purposes:

- source code editing;
- project management;
- machine code generation (from assembly or "C");
- MC simulation;
- MC emulation;
- MC programming.

The MPLAB IDE includes following tools:

Project Manager – used for the creating of projects and operating with the project files.

MPLAB Editor – dedicated for the creating and editing of program texts and machine codes.

MPLAB SIM – developed for the simulation of the MC program code execution and of I/O ports operation using computer.

MPLAB ICE – used for the program operating analysis in the real time by the employment of the special hardware – emulator.

MPLAB Assembler – used for the translating of the MC program (source file) to hex file, which is converted by the programmer into the machine code and loaded into the MC program memory.

MPLINK Object Linker – dedicated for the creating of the MC program from the several assembler programs or from the assembler and C language programs.

MPLINK Object Librarian – is used for the creating and administration of program libraries.

The use of MPLAB IDE software. The main product, which has to be obtained using software MPLAB IDE is the hex file of created program. To start work and reach this goal the following steps must be done using window of MPLAB IDE software:

1. *Selection of MC*. The type of MC, which will be used for the accomplishment of the project, is selected.

1.1. Choose *Configure>Select Device.*

1.2. Select the appropriate MC (in our situation PIC16F84A) in the *Device* dialog line.

1.3. Press *OK*.

The type of the MC that was chosen should appear in the line on the bottom of the window.

2. *Setting of MC Configuration Bits*. The appropriate MC configuration, which meets the requirements of the project, is provided for the selected MC.

2.1. Choose *Configure>Configuration Bits.*

2.2. The type of the resonator have to be entered up in the line *Oscillator*: LP, XT, HS for the crystal or ceramic resonator with the resonance frequency (32–200) kHz, (0.455–4) MHz, (8–10) MHz, accordingly; RC – in the case when the RC network is used as resonator.

2.3. The Watchdog Timer is switched on or switched of using the line *Watchdog Timer*, by choosing *On* or *Off*, accordingly.

2.4. The Power up Timer, which keeps the MC in the Reset state for the 72 ms, is switched on or switched of using the line *Power up Timer*, by choosing *On* or *Off*, accordingly.

2.5. The Code Protection function is switched on or switched of using the line *Code Protect*, by choosing *On* or *Off*, accordingly.

Note. The MC Configuration Bits can be set using the MPASM directive *__CONFIG* as well. This can be performed by entering of the appropriate text, e.g.: *__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF*. This text shows that the MC is configured following: resonator with the frequency in range 0.455 to

4 MHz is used, the Watchdog Timer and Code Protection function are switched of and Power up Timer is switched on.

The only one MC Configuration Bits Setting method from the described above should be chosen to avoid misunderstanding during the MC configuration.

3. *Entering of assembler program text*. The Program text is entered or loaded.

3.1. Create the new folder in the root of C disk. The files that are required for creating of MC machine code will be saved in this folder.

3.2. Choose *File>New*. The Program text has to be entered or loaded.

3.3. Choose *File>Save As* to save the program text. Enter the file title and save it as the Assembly source file, i.e. with the extension *.asm*. This file has to be saved in the created folder (point 3.1).

4. Creating a project. The project contains fails, which are required for the creating of the MC machine code, which is loaded into the MC program memory.

4.1. Choose *Project>Project Wizard*. Click *Next* to continue. Verify that the MC, which was selected, is shown here. Click *Next* to continue.

4.2. The *Language Toolsuite* window with the language toolsuite contents and roots to the toolsuites location should appear. If the root for the any toolsuite is not given, the appropriate line is marked. The roots for the all toolsuites must be presented to continue the project creating process. As an example, the roots must be as follows for the case when the software MPLAB IDE is located in the folder *Program Files* of the C disk:

– MPASM Assembler: C:\Program Files\MPLAB IDE\MCHIP_Tools\mpasmwin.exe ;
– MPLINK Object Linker: C:\Program Files\MPLAB IDE\MCHIP_Tools\mplink.exe ;
– MPLIB Librarian: C:\Program Files\MPLAB IDE\MCHIP_Tools\mplib.exe .

4.3. Click *Next*. After that, in the line *Project Name* of the window, which appeared, write the title of the project. In the line *Project Directory* point to the folder, which was created before (point 3.1). This can be done by clicking on the *Browse*. After that point to the folder and click *Next*.

4.4. Mark the files that have to be included into the project and click *Add*. The files can be removed by clicking on the *Remove*. It is obligatory that the file with the assembler program text, which has extension *.asm*, would be included into the project. Click *Next*.

4.5. The main information of the project should be presented in the window *Summary*, which appears: type of MC; language toolsuite; the title and root of the project file with the extension *.mcp*. If the presented information is not correct, click *Back* and make appropriate changes. If the information is correct, click *Finish* to finish the creating of the project.

4.6. The project window should appear in the MPLAB IDE window. The project title and files of the project should be displayed. Click *View>Project* on the MLAB IDE menu to open the project.

4.7. Click *Build All* button, which is located in the right hand side on the top of the MPLAB IDE window, to create the *hex* file. The green ribbon, which shows the dynamics of file creation, should appear for the short period on the screen. The red ribbon appears if the errors are found. The *Output* window is opened automatically after the process is over. The information about warnings and errors is presented in this window. The *hex* file is not created if the errors are found. The additional three files with the following extensions are created during this process: *.lst* – program listing; – *.err* error file; – *.cod* file, which is needed for debugging of the program, using simulator or emulator. All these files can be opened and viewed by clicking *File/Open* and choosing *Look in* catalog and after that by clicking on the *Files of type*.

6. THE PIC16F84A MICROCONTROLLER DEVELOPMENT BOARD

The PIC16F84A microcontroller development board is designed for the programming of microcontroller (for the loading of the programs into the MC program memory), testing of program operation, checking of program errors, investigation of MC operation with the external devices as LEDs, LCD display, keyboard. Using the CON10 connector the work of the MC with other external devices can be investigated.

The circuit diagram of the PIC16F84A microcontroller development board is presented in Fig. 6.1. It includes MC PIC16F84A, 4MHz crystal resonator Q1, buffer 74HCT125, LEDs D4, D5, D10–D22, LCD display LCD1601LC and keyboard M1–M3. It has the supply connector K1, the connector DB9 for computer serial interface RS232 and CON10 for connection of various external devices. The switches J1 and J2 are used for switching of development board operating modes. Additionally, the development board includes 5V supply voltage stabilizer 7805 and Reset button R.

Programming mode. The program is transmitted from computer using RS232 serial interface (COM port). The voltage (13 V) for the writing of MC program into the Flash program memory is produced by the TL497 integrated circuit. The buffer 74HCT125 is used for interfacing between the COM port standard and MC (TTL) standard digital signal voltage levels. The switch J1.1 is used to switch the development board from programming mode to operating mode and vice versa. This switch has to be in position ON during the MC programming.

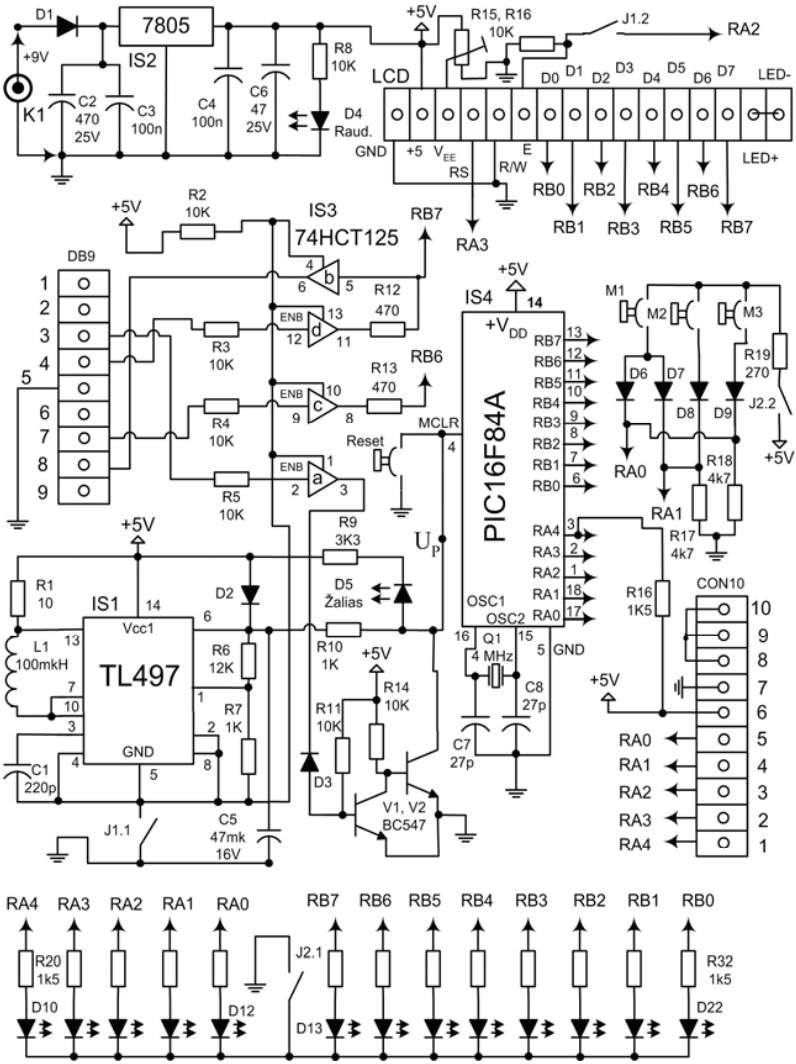


Fig. 6.1. Circuit diagram of the PIC16F84A microcontroller development board

Operating mode. The testing of the program operating is performed in the operating mode of the development board. The switch J1.1 has to be put to the position 1 to switch the development board to the operating mode. The supply of TL497 integrated circuit is disconnected in such a case and conducting of programming voltage to the MC is stopped. The states of the MC port pins in the MC operating mode can be observed using LEDs connected to the pins. If the appropriate LED emits the light the voltage on pin corresponds to logical "1" voltage, if not – the logical "0" voltage is indicated on the pin. The LEDs can be disconnected by switch J2.1. The LCD indicator LCD1601LC is used to display numbers, letters and various signs. The contrast of indicator can be adjusted using potentiometer R15.

The buttons M1–M3 act as keyboard. The voltage of logical "0" is conducted to MC pins RA0 and RA1 if the buttons are not pushed down. If they are pushed the voltage of logical "1" is conducted to these pins. The keyboard buttons can be disconnected using switch J2.2.

The external 9V power supply has to be used for the PIC16F84A microcontroller development board. The diode D1 is used for the development board protection in the situation when the polarity of the supply voltage is changed.

The PIC16F84A microcontroller development board is used for the accomplishment of laboratories presented in this laboratory works methodology.

References

1. Baskys, A. The Course of Microcontrollers Oriented to Practical Skills. *Solid state phenomena*, 2010, vol. 165, p. 410–413.
2. *PICmicro mid-range MCU family reference manual*, 2005. http://www.microchip.com.

3. *MPLAB IDE user's guide*, 2006. http://ww1.microchip.com/downloads/en/DeviceDoc/51519B.pdf.
4. Valdes-Perez, F. E.; Pallas-Areny, R. *Microcontrollers: Fundamentals and Applications with PIC*, 2009.
5. Katzen, S. *The Quintessential PIC microcontroller*, 2005.
6. Sanchez, J.; Canton, M. *Microcontroller programming*, 2007.

7. LABORATORY WORKS

7.1. Laboratory 1

Introduction to PIC16F84A microcontroller development board and software MPLAB IDE

1. Aim

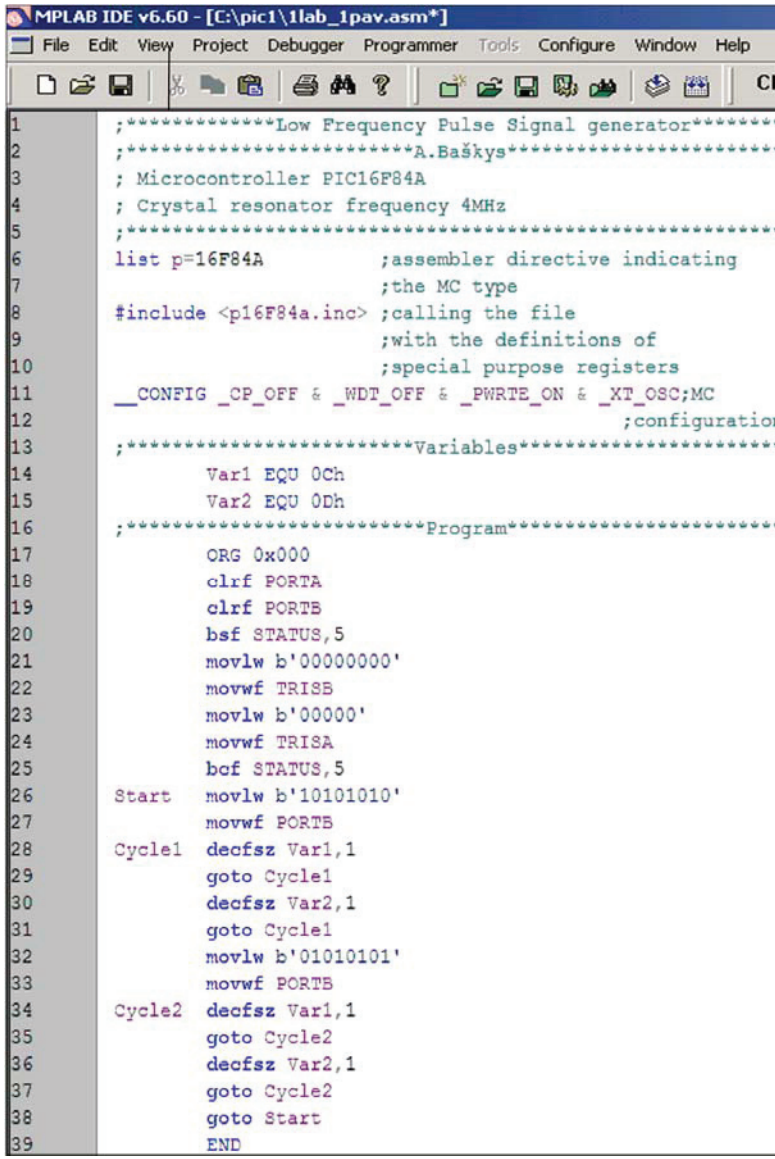
To study the PIC16F84A microcontroller development board and software MPLAB IDE.

2. Task

Learn to create a project, enter program text, translate it to the *hex* file and program MC (write the program into the MC program memory). Investigate the operating of the MC with the loaded program, learn to read and erase the MC program.

3. Proceeding

1. Learn the purpose of PIC16F84A MC development board connectors, switches, buttons and LEDs (Chapter 6).
2. Learn the purpose and structure of MPLAB IDE software (Chapter 5).
3. Create the new folder in the root of C disk.
4. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window and enter MC program text given in the Fig. 7.1. It is the low frequency pulse generator program, which allows generating the pulse signals in all B port pins. The material about the entering of program text is presented in the MPLAB Editor Help and is given in [3].
5. Choose *File>Save As* to save the program text. Enter the file title *generator* and save it as the Assembly source file, i.e. as *generator.asm.* This file has to be saved in the created folder (point 3).




```
1       ;*************Low Frequency Pulse Signal generator*********
2       ;************************A.Baškys************************
3       ; Microcontroller PIC16F84A
4       ; Crystal resonator frequency 4MHz
5       ;*********************************************************
6       list p=16F84A           ;assembler directive indicating
7                               ;the MC type
8       #include <p16F84a.inc> ;calling the file
9                               ;with the definitions of
10                              ;special purpose registers
11      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC;MC
12                                                  ;configuration
13      ;************************Variables************************
14              Var1 EQU 0Ch
15              Var2 EQU 0Dh
16      ;*************************Program************************
17              ORG 0x000
18              clrf PORTA
19              clrf PORTB
20              bsf STATUS,5
21              movlw b'00000000'
22              movwf TRISB
23              movlw b'00000'
24              movwf TRISA
25              bcf STATUS,5
26      Start   movlw b'10101010'
27              movwf PORTB
28      Cycle1  decfsz Var1,1
29              goto Cycle1
30              decfsz Var2,1
31              goto Cycle1
32              movlw b'01010101'
33              movwf PORTB
34      Cycle2  decfsz Var1,1
35              goto Cycle2
36              decfsz Var2,1
37              goto Cycle2
38              goto Start
39              END
```

Fig. 7.1. The text of the low frequency pulse generator assembler program

6. Create a project using the information given in the Chapter 5. Load the file *generator.asm* in it. Save the project in the created folder.
7. Create the *hex* file *generator.hex* and files *generator.lst, generator.err* and *generator.cod.* according the information presented in point 4.7 of Chapter 5.

Note. If there in the program text are errors, the *hex* file will be not created. The errors are indicated in the *Output* window, in the lines with the note *Error.* When errors are checked, the creating process of the *hex* file must be repeated.

8. Acquaint with the information given in *Output* window. Using *File>Open*, in the field *Look In* choose your directory, and by the choosing *All Files* in the field *Files of Type*, review the created files. Save the content of the hex file for the laboratory report.
9. Connect the PIC16F84A MC development board (Chapter 6) to the COM port of personal computer and switch on the power supply of the board. The red LED indicates that the power supply is on.
10. Switch the development board to the programming mode by setting the switch J1.1 to the state “On”.
11. Open the software NT PIC PROGRAMMER. Mark the appropriate COM port number in the *Communications port* field. Choose the MV16C84 in the field *Mode*. Read and analyze the contents of MC program and data (EEPROM) memories and contents of MC configuration bits. The fact of the exchange of data between the computer and development board is indicated by the blinking of green color LED, which is located on the board next to the COM port connector.
12. Erase the MC program memory contents using *Erase* field. Make sure that the code is erased. The *Read* field must be used for this purpose. The number 3fff must be in all program memory cells if the content of the memory is erased properly.

13. Switch the development board to the operating mode (toggle switch J1.1 to the state 1). Watch the diodes, connected to the A and B ports. All LEDs connected to the MC ports should not emit the light.
14. Load the *hex* file of the MC program. Push the field *Load* for this purpose, choose your directory and the file *generator. HEX* in it. Mark the file and push field *Open*. Examine the program code contents in the window of the software NT PIC PROGRAMMER.
15. Switch the development board to the programming mode (toggle switch J1.1 to the state ON). Write the program code into the MC program memory using field *Write*.
16. Switch the development board to the operating mode (toggle switch to the state 1). Watch the LEDs connected to the port B. Make sure that the low frequency pulse signals are generated in the port B pins (LEDs are blinking). Draw voltage transient diagrams of all B port pins (not less than 3 periods). Assume that LED emitting light shows voltage corresponding to "1" (5 V) and not emitting light – "0" (0 V). There is no need to put time terms in the time axis of the voltage transient diagrams.
17. Switch the development board to the programming mode and doing the same operations as in the point 12 of the task erase the MC program. Switch the development board to the operating mode and make sure that the program is deleted, i.e. the signals are not generated (the LEDs connected to the port B do not emit the light).
18. Performing operations described in the points 14 and 15 of the task, again write the program into the MC program memory and make sure that it operates.

4. Report content

1. The aim of the work.

2. The characterization of the MPLAB IDE software purpose and structure.
3. The text and *hex* file of the MC program (points 4 and 8 of the task).
4. The voltage transient diagrams of all B port pins (point 16 of the task).
5. Conclusions.

5. Control questions

1. What is the purpose of the PIC16F84A microcontroller development board?
2. What kind of the memory is used as the program memory in the PIC16F84A microcontroller?
3. What is forced to do the microcontroller if the *Reset* button is pushed down?
4. What indicates the LEDs connected to the MC ports?
5. What is the purpose of the crystal resonator connected to the MC pins?
6. What are the voltage levels of the logical "0" and "1" of the TTL signals and signals provided by the COM port of the personal computer?
7. What is the purpose of the MPLAB IDE software?
8. What programs includes the MPLAB IDE software?
9. What is the purpose of the MPLAB SIM program?
10. What is the purpose of the MPLAB ICE program?
11. What is the purpose of the programmer?
12. What information is set during the MC configuration?
13. What additional files are build during the creation of the *hex* file?

References

1. Weber, R. PICee development system. *Elektor Electronics*, No 2, 2002, p. 14–18.
2. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
3. MPLAB IDE user's guide, DS51519A, Microchip Technology Inc., http://www.microchip.com, 2005. 240 p.

7.2. Laboratory 2

Writing the data into the microcontroller ports

1. Aim

Study the MC PIC16F84A ports, special purpose registers STATUS, TRISA, TRISB, assembler directives *ORG,* LIST, INCLUDE, CONFIG, END and instructions *clrf, bsf, bcf, movlw, mowwf.*

2. Task

Configure MC, set the pins of ports for data input and output and write data into the controller ports.

3. Proceeding

1. Analyze the structure, circuit diagrams and properties of MC PIC16F84A ports.
2. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. In the beginning of the program type text, with title of the program, author, type of MC and resonator frequency.

```
;Writing the data into the microcontroller ports
;******************N. Surname*****************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;*****************************************
```

3. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC.

```
LIST p=16F84              ;Setting type of MC
```

```
#INCLUDE <p16F84a.inc> ;Calling the file,
                       ;with the definitions of
                    ;special purpose registers
_ _ CONFIG _ XT _ OSC&   ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

Comment. More information about the MC configuration is given in chapter 5, (2 point). Semicolon in program shows the beginning of comments. The text to the right of the semicolon doesn't influence the program operation.

4. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000;setting the beginning
         ;address of the program
```

Comment. This directive points at the program beginning address. Program memory address 0×000 (0000h) is called MC Reset Vector. After the Reset the MC starts to execute the program, which begins from this address.

5. Clear the PORTA and PORTB registers using instruction *clrf*:

```
clrf PORTA        ;clear PORTA register
clrf PORTB        ;clear PORTB register
```

Comment. Instructions *clrf* that are presented in the strings written above clears the contents of appropriate registers, i.e. write "0" into all bits of PORTA and PORTB registers.

6. Using bit RPO (5th bit) of STATUS register and instruction *bsf* set the work of MC for operating with the registers located in Bank 1.

```
bsf STATUS, 5           ;move to  Bank 1
```

Comment. By default “0” is stored in bit RPO of STATUS register, i.e. the MC is set to work with the data memory registers that are located in Bank 0. Instruction *bsf (bit set file bit n)* is used to set “1” in the n-th digit of the register. In this case it configures MC to operate with the registers of Bank 1.

7. Set Port B pins for signal output using TRISB register:

```
   movlw b'00000000'     ;write  the  binary  digit
to
                  ;W register
   movwf TRISB           ;move W register content
                  ;to the TRISB register
```

Comment. Instruction *movlw (move literal to W)* is used to move 8 digit number to W (*Working*) register. Letter *b* shows that the number is binary.

Instruction *movwf (move W to f)* moves content of *W* register to the register *f* (in our case to the register TRISB).

The register TRISB is used for the port B configuration, i.e. to set the pins of port as inputs or outputs. To configure the pin for input, it is necessary to set the corresponding bit of TRISB register (to write “1”). To configure the pin for output, it is necessary to clear the corresponding bit of TRISB register (to write “0”).

The register TRISA is used in the same way for configuration of port A pins.

8. Using bit RPO (5th bit) of STATUS register and instruction *bcf* set the work of MC for operating with the registers located in Bank 0:

```
   bcf STATUS, 5  ;move to Bank 0
```

Instruction *bcf (bit clear file bit n)* is used to set “0” in n-th bit of the register. In this case it configures MC to operate with the registers located in the Bank 0.

9. Write the following fragment of program:

```
   movlw b'11111111'      ;write  the  binary  digit
to
                    ;W register
   movwf PORTB            ;move  content  of  W  regis-
ter
                    ;to the register PORTB
```

Comment. The instructions used above set all PORTB register bits, i.e. write "1". Since all port B pins are configured as outputs, the potentials, which corresponds to "1" potential of TTL standard will appear on all port B pins. The instructions used in the analyzed program fragment are the same as these discussed in point 7.

10. Indicate the end of the program using directive END:

```
   END       ;end of the program
```

11. By choosing *File>Save As* save the program as assembly source file. Before that entitle it.
12. Using *Project>Open*, open project that was created during 1 laboratory work. Mark assembler file saved in *Source files* folder and remove it by executing *Remove.* Then mark the catalog *Source Files* and executing *Add File*, move to project the assembler file of the created program.
13. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file and using MC PIC16F84A development board load it into the MC.
14. Switch the development board to operating mode (see point 16 of 1 laboratory work) and make sure that program operates properly, i.e. the "1" written in to the all PORTB register bits, should be indicated by the emitting of light from the all LED's that are connected to the pins.

Comment. When power supply is connected to the MC, all pins of ports A and B are set for input by the default (the impedance between pins and supply source is high), so LEDs connected to port A pins should not emit the light.

15. Change program so that the MC pins RB0, RB2, RB4 and RB6 would be set for data input, and the pins RB1, RB3, RB5 and RB7 – for data output. The voltage of "1" must be conducted to RB1, RB3, RB5 and RB7 pins. Leave all A pins set to input. Accomplish actions described in points 13 and 14 of this laboratory work. By observing of LEDs connected to the pins make sure that program works properly. Choose *File>Save As* and save the text of the program for the laboratory report.
16. Modify the program in such a way that all port A and port B pins would be set as outputs and to all pins the voltage, which corresponds to "1" would be conducted. Execute the actions 13 and 14 of this laboratory work. Make sure that program works properly by observing the state of LEDs connected to the pins.
17. By executing of actions similar to these presented in point 16 of this laboratory work make that the voltage, which corresponds to "1" would be conducted to pins of port A RA1 and RA2 and to the pins of port B RB0, RB2, RB4 and RB6. The voltage that corresponds to "0" must be conducted to all remaining pins. Using MC PIC16F84A development board load the program into the MC. Save the text of the program for the laboratory report.
18. Set the port A pins RA0 and RA1 for data input. The remaining pins leave in the state as it was made in the point 17 of this laboratory work. Load the program into the MC. Pushing the buttons M1-M3 of MC PIC16F84A development board, find out which logic level voltage is connected to the pin when button is pressed and not pressed and which button commutates which terminal voltage.

4. Report content

1. The aim of the work.
2. The block diagram of circuit used for pin RA0 control and its analysis.
3. The block diagram of circuit used for pin RB0 control and its analysis.
4. The description of purpose of MC PIC16F84A registers STATUS, TRISA and TRISB, assembler directives ORG, LIST, INCLUDE, CONFIG, END and instructions *clrf, bsf, bcf, movlw* and *movwf*.
5. The texts and comments of the programs that were saved during the execution of tasks given in points 15 and 17.
6. Conclusions.

5. Control questions

1. What is the purpose of MC ports?
2. How many ports has MC PIC16F84A?
3. How many bits (pins) includes port B?
4. What is the difference of the circuit that is used for control of pin RA4 in comparison with these that are employed for control of remaining pins of port A?
5. What is the difference of the circuits that are used for control of pins RB4–RB7 in comparison with these that are employed for control of remaining pins of port B?
6. What is the purpose of assembler directives ORG and END?
7. What is the purpose of RPO bit of STATUS register?
8. How many banks includes the SRAM data memory of MC PIC16F84A?
9. Clarify the meaning and purpose of instruction *clrf*.
10. What shows the semicolon in the program text string?
11. Clarify the meaning and purpose of instructions *bsf* and *bcf*.
12. Clarify the meaning and purpose of instructions *movlw* and *movwf*.

13. To the what mode (as inputs or outputs) a set all port A and port B pins by default after the MC supply is switched on?
14. What is the purpose of TRISA and TRISB registers?
15. In to the what two groups the SRAM data memory registers of MC PIC16F84A are partitioned?

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.3. Laboratory 3

Creating a program loops

1. Aim

To study instructions *goto, nop*, *decfsz,* assembler directive EQU and the program loop creating principles.

2. Task

Create the MC PIC16F84A programs of high and low frequency pulse signal generators, test and investigate them.

3. Proceeding

A. High frequency pulse signal generator program

1. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. In the beginning of the program type text, with title of the program, author, type of MC and resonator frequency.

```
; High frequency pulse signal generator
;******************N. Surname****************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;*****************************************
```

2. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC.

```
LIST p=16F84                ;Setting type of MC
#INCLUDE <p16F84a.inc>      ;Calling the file,
```

```
                              ;with the definitions of
                        ;special purpose registers
_ _ CONFIG _ XT _ OSC&          ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

3. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000    ;setting the beginning
      ;address of the program
```

4. Clear the PORTA and PORTB registers using instruction *clrf*.

```
clrf PORTA          ;clean PORTA register
clrf PORTB          ;clean PORTB register
```

5. Using bit RPO (5th bit) of STATUS register and instruction *bsf* set the work of MC for operating with the registers located in Bank 1:

```
bsf STATUS, 5             ;move to  Bank 1
```

6. Set all port A and port B pins for signal output, set the work of MC for operating with the registers located in Bank 0 and set all bits of PORTA and PORTB registers to “1”:

```
movlw b‘00000000‘       ;write the binary digit to
                        ;W register
movwf  TRISB            ;move the content of W
                        ;register into TRISB
                        ;register
movlw b‘00000‘          ;write the binary digit to
                        ;W register
```

```
movwf  TRISA            ;move the content of W
                        ;register into TRISB
                        ;register
bcf STATUS, 5           ;move to Bank 0
movlw b'11111'          ;write the binary digit to
                        ;W register
movwf PORTA             ;move the content of
                        ;register W into register
                        ;PORTA
Start movlw b'11111111' ;write the binary digit to
                        ;W register
movwf PORTB;            ;move the content of
                        ;register W into register
                        ;PORTB
```

Comment. The word S*tart* acts as a label which marks the appropriate string, which shows where to move the execution of a program.

7. Write "0" into all bits of PORTB register

```
movlw b'00000000'   ;enter the binary number written
                    ;in commas into register W
movwf PORTB         ;move content of W register
                    ;to the register PORTB
```

8. Using instruction *goto* (go to address) move the execution of the program to the line marked by the label *Start*.

```
goto Start          ;go to the string with the label
                    ;Start
```

Comment. A closed loop, which is executed permanently, is made using instruction *goto*. It includes the segment of a program between the beginning of the cycle (between the string marked by label *Start*) and the instruction *goto Start*, which forces to get back to the string marked by label *Start*. Any informative short word can be used as a

mark (it must not coincide with the instruction or assembler directive). Since “1” and “0” are being written alternately into the bits of the PORTB, the pulse signal with the same phase is generated in the all pins of port B. The period of the pulse signals is determined by the instruction execution duration. The MCs of 8 bit PIC Mid-range family including MC PIC16F84A perform one instruction (except the control instructions *call, goto, retfi, retlw, return*) during the one Machine Cycle (MC), which lasts 4 periods of the Clock Generator signal. Since the frequency of the Clock Generator in development board, which is used for the laboratories is 4 MHz, the duration of MC is 1 μs. The duration of control instructions is equal to twoMC. The loop analyzed above includes 5 instructions. One of them is control instruction (*goto*) so the period of generated signal is 6 μs and the frequency is approximately 0.17 MHz. It is impossible to observe the blinking of LEDs at such a high frequency of pulse signal. However, the presence of pulse signal can be indicated by lower light intensity emitted by LEDs as compared to the case when voltage of “1” is conducted constantly to port pins.

9. Indicate the end of the program using directive END.

```
END   ;end of the program
```

10. By choosing *File>Save As* save the program as assembly source file. Before that entitle it.
11. Using *Project>Open*, open project that was created during 1 laboratory work. Mark assembler file saved in *Source files* folder and remove it by executing *Remove.* Then mark the catalog *Source Files* and executing *Add File*, move to project the assembler file of the created program.
12. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file and using MC PIC16F84A development board load it into the MC.

13. Switch the development board to operating mode (see point 16 of 1 laboratory work) and make sure that program operates properly, i.e. the intensity of the light produced by the LEDs that are connected to the pins of port A is higher than intensity of light generated by the LEDs connected to the pins of port B.
14. Using instruction *nop* (no operation), which is employed for the introduction of the delay that is equal to 1 MC, increase the signal period duration (the duration of MC is 1 μs in the development board used for the laboratories). Introduce ten identical strings with the instruction *nop* just before the string *goto Start*:

```
nop   ;introduce a delay of 1μs
.
.
.
nop   ;introduce a delay of 1μs
```

Comment. The introduction of the instructions written above allows us to increase by 10 μs the duration of the state, during which the voltage that corresponds to "0" is conducted to the port B pins. The duration of the state, at which the voltage that corresponds to "1" is conducted, remains the same. *Remark:* if the period duration should be increased significantly, i.e. if it would be necessary to generate low frequency pulse signal, the very high number of strings with the instruction *nop* should be introduced. However, this is not acceptable because of limited volume of the program memory. This fact is the reason why another solutions should be used in such a case.

15. Create the *hex* file of a program and load the code into MC.
16. Switch the development board to operating mode and make sure that the intensity of the light produced by the LEDs that are connected to the port B pins is lower than this one obtained before the introduction of strings with the *nop* instructions.

17. Save the text of the program for report of the laboratory work.

B. Low frequency pulse signal generator program

18. Using an Assembler directive EQU define the names for registers, in which the values of variables of the delay loops will be stored. In order to do that enter the fragment of a program just after the line, which is used for the configuration of MC:

```
;********************Variables*******************

Var1    EQU    0Ch          ;gives name Var1 to 0Ch
Var2    EQU    0Dh          ;gives name Var2 to 0Dh

;************************************************
```

Comment. Using an assembler directive EQU, which is used for setting of constants, is convenient to name the registers. It lets us to use instead of an address of the register the name, which has the informative name related to the variable purpose and can be easy remembered. Any free data memory general purpose register can be used to store a variable.

19. Enter the strings of a program just under every string, which contains the instruction *movwf* PORTB. Under the first string with instruction *movwf* PORT*B* enter:

```
Cycle1 decfsz Var1,1  ;subtracts 1 from variable
                      ;Var1 and when it becomes
                      ;equal to 0 skips the next
                      ;instruction goto Cycle1
goto Cycle1
                      ;skips to the string with
                      ;the label Cycle1
```

```
decfsz Var2,1
                    ;subtracts 1 from variable
                    ;Var2 and when it becomes
                    ;equal to 0 skips the next
                    ;instruction goto Cycle1
goto Cycle1         ;skips to the string with
                    ;the label Cycle1
```

Under the second line with instruction *movwf* PORTB enter

```
Cycle2 decfsz Var1,1  ;subtracts 1 from variable
                      ;Var1 and when it becomes
                      ;equal to 0 skips the next
                      ;instruction goto Cycle2
goto Cycle2           ;skips to the string with
                      ;the label Cycle2
decfsz Var2,1         ;subtracts 1 from variable
                      ;Var2 and when it becomes
                      ;equal to 0 skips the next
                      ;instruction goto Cycle2
goto Cycle2           ;skips to the string with
                      ;the label Cycle2
```

Comment. Instruction *decfsz* (decrement f, skip if zero) subtracts one from the variable f, the value of which is stored in the appropriate data register, and when it becomes equal to 0, forces to skip the next instruction (the next string) of the program.

The 8 strings of the program presented above create two double loops, which are called the delay loops. They are used to increase significantly the duration of the states, during the which the “0” and “1” are stored in the PORTB register bits. Because of this, the frequency of generated pulse signal in port B pins is reduced proportionately.

A random number (its' value depends on asymmetry of trigger arms, which acts as SRAM memory bit cell) appears in the general purpose registers after the supply of the MC is switched on. Therefore, the numbers stored in registers 0Ch and 0Dh (the values of the variables Var1 and Var2) and, as a consequence, the duration of the first delay cycle in the delay loops is unknown. Since the first cycle ends when the content of the register decreases till 00, the duration of the following cycles is determined. At the beginning, when the digit 1 is subtracted from 00 the result is hexadecimal number FF (decimal 255). Because of this, instructions *decfsz Kint1,1* and *decfsz Kint2,1* in all cycles except the first one are repeated 256 times. During each cycle 1 is subtracted from the variable and the result is moved to the variable register. When the value of the variable reaches 00, instruction *decfsz* forces to skip the next instruction (instruction *goto*). Since the duration of MC is 1 μs and two instructions are executed during one cycle and one of them is control instruction *goto*, the duration of the execution of the top loop is 3 μs x 256 = 768 μs. The total duration of double delay loop execution (768 μs x 256) + 768 μs = 197 376 μs. When the skipping condition is being performed the duration of an instruction *decfsz* increases by 1 μs, so the total duration of a double delay loop is 197 376 μs + 256 μs = 197 632 μs ≈ 0,2 s. During this period the voltage that corresponds to "1" is conducted to the port B pins. After that the voltage, which corresponds to "0" is provided for the same duration. Because of this, the period of the generated pulse signal using discussed double delay loops is 0,4 s, i.e. the frequency of the signal is 2,5 Hz.

20. Delete the rows, which contain the *nop* instructions.
21. Change the title of the program (the top line of the program) to the following: "Low frequency pulse signal generator".
22. Create the *hex* file of a program and load the code into MC.
23. Switch the development board to operating mode and make sure that program works (LEDs that are connected to the

port A pins emit the light constantly and the LEDs that are connected to port B pins blink with the 2,5 Hz frequency).

24. Modify the program in such a way that the signals in the pins RB0, RB2, RB4 and RB6 would be inverted in respect of the signals in the RB1, RB3, RB5 and RB7 pins. Load the code into MC, observe the blinking of LEDs and make sure that the program works, i.e. when one group of LEDs emits the light the another – does not emits and vice versa.
25. Modify the program, which was created in section 24 in such a way that low frequency pulse signals would be generated in the all port A and port B pins. The signals generated in pins of every port must be synchronous, however, the signals generated in port A pins must be in opposite phase as compared to the signals generated in the port B pins. Load the code of the modified program into MC, observe the blinking of LEDs and make sure that the program works (LEDs that are connected to the port A pins emit the light during one half of the period and LEDs connected to the port B pins emit the light during the another half of the period).
26. Modify the program, which was made in section 25, using the *nop* instruction (execution duration of this instruction is 1 μs) in such a way that the period of generated pulse signal would increase 4 times (the frequency would be approximately 1 Hz). Load the code into MC, observe the blinking of LEDs and make sure that the frequency of blinking has decreased up to 1 Hz. Save the text of the program for report of the laboratory work

4. Report content

1. The aim of the work.
2. Purpose of instructions *goto, nop, decfsz* and the directive EQU.
3. The texts and comments of the programs that were saved during the execution of tasks given in points 17 and 26.
4. Conclusions.

5. Control questions

1. What is the purpose of the directive EQU?
2. In what registers, of which MC memory, the variables of a program are stored?
3. What numbers appear in the general purpose registers after the power supply of MC is turned on?
4. What is the purpose of the instruction *goto*?
5. Explain how the program cycle is created.
6. What is the relation between the clock signal period and Machine Cycle?
7. How many Machine Cycles are needed for the execution of control instructions?
8. What is the purpose of the instruction *nop*?
9. What is the purpose of the instruction *decfsz*?
10. How to calculate the program loop duration?
11. Explain the principle of pulse signal generation using MC.

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.4. Laboratory 4

Creating of subroutines and reading the data from ports

1. Aim

To study instructions *call, return, btfsc, btfss,* subroutine creating principles and methods of data reading from MC ports.

2. Task

To create, test and investigate the pulse signal generator programs for the MC PIC16F84A based on the subroutines, which include the possibility of pulse signal parameters control by the port input signals.

3. Proceeding

1. Analyze the following 10 ms delay loop subroutine:

```
:**************** 10 ms delay subroutine **************
tenms
      movlw d'13'          ;writes the number into
                           ;W register
movwf Var4                 ;moves the content of W
                           ;register into Var4
                           ;register
Cycle decfsz Var3,1        ;subtracts 1 from variable
                           ;Var3 and when it becomes
                           ;equal to 0 skips the next
                           ;instruction
goto Cycle                 ;skips to the string with
                           ;the label Cycle
decfsz Var4,1              ;subtracts 1 from variable
                           ;Var4 and when it becomes
```

```
                    ;equal to 0 skips the next
                    ;instruction
    goto Cycle      ;skips to the string with
                    ;the label Cycle
    return          ;return to main program
;***********************************************
```

Comment. The subroutine must be entitled (the analyzed subroutine is entitled *tenms*). Instruction *call* (Call Subroutine) is used in the main program to call up the subroutine (eg. *call tenms*). When execution of the main program reaches the instruction call, the execution of appropriate subroutine is started. Instruction *Return (Return from Subroutine)* must be written at the end of a subroutine. It returns back to continue the execution of the main program. The execution of the main program is continued starting from instruction, which follows the *call* instruction. It is convenient to use the subroutine when the same program fragment is repeated several times. The program becomes shorter and clear and program memory space is saved if this fragment is presented as subroutine. However, the employment of subroutine increases the duration of program execution, because the two additional instructions *call* and *return* must be included. The subroutine text usually is placed at the end of the program text, just before the assembler directive END. The program can include lot of subroutines.

The upper cycle of the subroutine includes 256 repetitions. The duration of 1 cycle is equal to 3 Machine Cycles (MC), because the duration of the *decfsz* instruction execution is equal to 1 MC (except the case when skiping is being performed). The execution of instruction *goto* takes 2 MC. The number of repetitions of the lower cycle is defined by the number loaded into Var4 register. In the analyzed subroutine the number 13 is loaded. The 1MC=μs, because of this, the total time of the analyzed delay loop execution is (3 μs x 256 x 13) + (3 μs x 13) = 10 023 μs. We should also add

the execution duration of instructions *movlw* and *movwf* (1 μs + 1 μs), time for calling up the subroutine (execution of instruction *call)* (2 μs) and time for returning to the main program, i.e., execution of instruction return (2 μs). Consequently, the total time of execution of the analyzed delay loop is 10 043 μs ≈ 10 ms.

It is convenient to use this subroutine for the creating of long duration delay loops with defined duration. For example, a delay loop with the duration of 1 s could be made by involving this subroutine into a cycle with the 100 repetitions. Using such a principle the delay loops with the various execution durations can be created. These delay loops can be employed for the realization of the pulse signal generators with the given pulse signal frequency and pulse duty cycle.

2. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. In the beginning of the program type text, with title of the program, author, type of MC and resonator frequency.

```
;Low frequency signal generator program
;based on the subroutine
;******************N. Surname*****************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;*****************************************
```

3. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC.

```
LIST p=16F84                ;Setting type of MC
#INCLUDE <p16F84a.inc>      ;Calling the file,
                            ;with the definitions of
                    ;special purpose registers
```

```
_ _ CONFIG _ XT _ OSC&        ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

4. Define the names for registers (introduce variables):

```
;*****************Variables***********************

Var1    EQU    0Ch          ;gives name Var1 to 0Ch
Var2    EQU    0Dh          ;gives name Var2 to 0Dh
Var3    EQU    0Eh          ;gives name Var3 to 0Ch
Var4    EQU    0Fh          ;gives name Var4 to 0Dh

;***********************************************
```

5. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000    ;shows the initial address
             ;of the program
```

6. Clear the PORTA and PORTB registers:

```
clrf PORTA          ;clean PORTA register
clrf PORTB          ;clean PORTB register
```

7. Set the work of MC for operating with the registers located in Bank 1:

```
bsf STATUS, 5       ;move to Bank 1
```

8. Set all port A and port B pins for signal output, set the work of MC for operating with the registers located in Bank 0, clear all bits of PORTA register and set all bits of register PORTB:

```
movlw b‘00000000‘  ;write the binary digit to
                   ;W register
movwf  TRISB       ;move the content of W
                   ;register into TRISB
                   ;register
movlw b‘00000‘     ;write the binary digit to
                   ;W register
movwf TRISA        ;move the content of W
                   ;register into TRISB
                   ;register
bcf STATUS, 5      ;move to Bank 0
Start movlw b‘00000‘;write the binary digit to
                   ;W register
movwf PORTA        ;move the content of
                   ;register W into register
                   ;PORTA
movlw b‘11111111‘  ;write the binary digit to
                   ;W register
movwf PORTB;       ;move the content of
                   ;register W into register ;PORTB
```

9. Enter the fragment of low frequency pulse signal generator program:

```
      movlw d‘50‘  ;writes the binary digit
                   ;into W register
      movwf Var1   ;moves the content of W
                   ;register into Var1 register
Cycle1 call tenms ;calls subroutine tenms
      decfsz Var1,1      ;subtracts 1 from variable
```

```
                    ;Var1 and when it becomes
                    ;equal to 0 skips to
                    ;instruction goto Cycle1
      goto Cycle1   ;goes to the string of a
                    ;program, which is marked
                    ;Cycle1
      movlw b'00000000'  ;writes the binary digit
                         ;to W register
      movwf PORTB   ;moves the content of W
                    ;register into register PORTB
      movlw b'11111'     ;writes the binary digit
                         ;into W register
      movwf PORTA   ;moves the content of
                    ;register W into register
                    ;PORTA
      movlw d'50'   ;writes the decimal digit
                    ;into W register
      movwf Var2    ;moves the content of W
                    ;register into register Var2
Cycle2 call tenms   ;calls subroutine tenms
      decfsz Var2,1      ;subtracts 1 from variable
                         ;Var2 and when it becomes
                         ;equal to zero skips
                         ;instruction goto Cycle2
      goto Cycle2   ;goes to the string of a
                    ;program, which is marked
                    ;Cycle2
      goto Start    ;goes to the string of a
                    ;program, which is marked
                    ;Start
```

Comment. The program fragment presented above differs from the analogous fragment of low frequency generator program analyzed

in the laboratory 3 by the fact that subroutine *tenms* (it is presented in the first section of this laboratory) has been employed in the delay loops Cycle1 and Cycle2 and the values of variables Var1 and Var2 have been defined also. It lets us obtain desired duration of delay loops, i.e. desired pulse signal frequency and duty cycle. Variables Var1 and Var2 have values of 50 (the execution duration of each delay loop is 10 *ms* · 50 = 500 *ms*) in this fragment of a program, so MC will generate pulse signal with the frequency 1 Hz and duty cycle is 0.5. Actually, frequency and duty cycle will be slightly different because each of the cycles adds an additional delay and execution duration of subroutine *tenms* differs a little bit from 10 ms.

10. Enter the subroutine *tenms* (it is presented in the point 1) after the string *goto Start* on the bottom of the program fragment given in the point 9.
11. Indicate the end of the program using directive END.

```
END   ;end of the program
```

12. Save the program as assembly source file. Before that entitle it. Open the project that was created during the 1 laboratory work. Remove assembler file saved in it, then mark the catalog *Source Files* and executing *Add File*, move to project the assembler file of the created program.
13. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file and using MC PIC16F84A development board load it into the MC.
14. Switch the development board to operating mode (see point 16 of 1 laboratory work) and make sure that program operates properly, i.e. LEDs are blinking, the frequency of signals is close to 1 Hz, the duty cycle – 0.5 and signals generated in the port A pins are inverse to these generated in the port B pins.
15. Modify the program in such a way that the duty cycle of signals would remain 0.5, but frequency would be: 5 Hz, 0,5 Hz.

16. Load the programs into the MC and test them.
17. Modify the program so that the frequency of the signal would be 0,5 Hz and the duty cycle of the "1" voltage in the port B would be 0,2.
18. Load the program into the MC and test it.
19. Save the program for report of the laboratory work.
20. Modify the program in such a way that the frequency of the signal would be 1 Hz, duty cycle 0.5 and the signals generated in the pins RB0, RB2, RB4, RB6 would be inverted in respect to the signals generated in the pins RB1, RB3, RB5, RB7.
21. Using the register TRISA and appropriate instructions set all port A pins for signal input.
22. In the text of the main program bellow every string with the instruction *call tenms* enter the following fragment:

```
btfsc PORTA,0  ;reads the input signal in the pin
               ;RA0, if there is "1" ;the next
               ;instruction is executed, if "0" -
               ;the next instruction is skipped
call tenms     ;calls the subroutine tenms
```

Comment. Instruction *btfsc* is used to read the data of the appropriate register bit. In order to do that, the register (e.g. PORTA) and bit (e.g. 0) must be indicated in the operands of the instruction.

If the "1" is read, the next instruction of the program is executed, i.e. the program is executed in the same way as if it would be executed if this instruction would be not included into the program. The instruction *btfsc* cause to skip the next instruction if the "0" was read. The use of the instruction *btfss (bit test f, skip if set)* is the same as *btfsc* instruction. The only difference is that the next instruction of the program is executed if "0" is read, and it is skipped if "1" is read. The buttons M2 and M3 on the development board switches the voltage in the pins RA0 and RA1, respectively.

When the button is pressed the voltage that corresponds to "1" is conducted, if it is not pressed, the voltage of "0" is provided. Button M1 switches summarily the voltage of both these pins.

The instructions *btfsc, btfss* are used to link the MC with the keyboard or other device whose data have to be processed and indicated or used it in order to control other devices. The MC PIC16F84A is not able to read analogue signals because it doesn't include the ADC.

Instruction *call tenms* is included twice in the delay loops, therefore, the duration of delay loop execution can be elongate twice, i.e. the frequency can be twice lower. How many times the subroutine *tenms* will be executed in the delay loop depends on the fact what logical level will be read in the RA0 pin by the instruction *btfsc*: if "0" – one of the program strings with the instruction *call tenms* will be skipped and frequency of the generated signal will be higher, if not – the subroutine *tenms* will be executed twice, i.e. the signal frequency will be twice lower. Since the voltage level on the pin RA0 is controlled by the button M3, pushing of this button will change the frequency of the pulse signal generated by the MC.

23. Load the program into the MC. Make sure that it works, i.e. 1 Hz frequency pulse signals with the 0.5 duty cycle are generated in the port B pins and the signals generated in the pins RB0, RB2, RB4, RB6 are inverted in respect of the signals generated in the pins RB1, RB3, RB5, RB7. When the button M3 is pressed the LED connected to the pin RA0 should light and the frequency of generated pulse signal should decrease twice.
24. Modify the program by entering of the instruction *btfss* instead of instruction *btfsc*. Make sure that the pressing of a button M2 make inverse impact to frequency change, i.e. when the button is pressed the signal frequency increases from 0.5 to 1 Hz.
25. Using instruction *btfsc*, modify the program in such a way that when the button M3 is not pressed, the signals generated in the pins RB0, RB2, RB4, RB6 would be inverted in respect

of the signals generated in the pins RB1, RB3, RB5, RB7. The frequency of the signal must be 2 Hz, the duty cycle – 0.5. When the button is pressed the duty cycle of the "1" of the generated signals in the outputs RB0, RB2, RB4, RB6 should decrease up to 0,1, but frequency must remain the same.

26. Load the program into the MC and by the observing the flashing of the LEDs and pressing the button M3 make sure that it works properly.
27. Using instruction *btfss* modify the program in such a way, that when the button M2 in not pressed, the same phase 1 Hz frequency pulse signals would be generated in the all port B pins. However, when this button is pressed the frequency must remain the same but signals generated in the pins RB0, RB2, RB4, RB6 must be inverted in respect of the signals generated in the pins RB1, RB3, RB5, RB7.
28. Load the program into the MC and by the observing the flashing of the LEDs and pressing the button M2 make sure that it works properly.
29. Modify the program so that additionally the signal frequency could be changed by pressing of M3 button from 1 Hz (when the button M3 is not pressed) to 0.5 Hz (when the button M3 is pressed).
30. Load program to MC and make sure that it works by observing the flashing of the LEDs and pressing the buttons M2 and M3. Button M3 should change the frequency while button M2 should change the phase of the generated signals.
31. Save the program for the report of the laboratory. Name the program following: "Low frequency pulse generator with variable phase and frequency".

4. Report content

1. The aim of the work.
2. Analysis of instructions *call, return, btfsc* and *btfss.*

3. The texts and comments of the programs that were created during the execution of tasks given in points 19 and 31.
4. Conclusions.

5. Control questions

1. Explain the purpose of instruction *call*.
2. Explain the purpose of instruction *return*.
3. When is it useful to use the subroutines? What advantages provide the using of the subroutine?
4. Explain how to create the subroutine.
5. What is the purpose of instructions *btfsc* and *btfss*?
6. What registers are used in order to configure the pins of the ports for signal input and output?
7. Is it possible to read the analog signals using the MC PIC16F84A?
8. Comment the fragment of the MC program, which allows changing the frequency of the generated pulse signals by pressing a button.
9. Comment the fragment of the MC program which allows changing the phase of the generated signals by pressing a button.

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.5. Laboratory 5

Investigation of complementation, swap, rotation and logic functions instructions

1. Aim

To study complementation, swap, rotation instructions *comf, swapf, rlf, rrf* and logic function instructions *andwf, andlw, iorwf, iorlw, xorwf, xorlw.*

2. Task

Create MC PIC16F84A programs of high and low frequency pulse signal generators with complementation, swap, rotation and logic function instructions, test and investigate them.

3. Proceeding

1. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. In the beginning of the program type text, with title of the program, author, type of MC and resonator frequency.

```
;********Low frequency pulse signal generator******
;******************N. Surname*********************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;************************************************
```

2. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC.

```
LIST p=16F84              ;Setting type of MC
#INCLUDE <p16F84a.inc>    ;Calling the file,
```

```
                              ;with the definitions of
                                ;special purpose
                                ;registers
_ _ CONFIG _ XT _ OSC&       ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

3. Define names of registers:

```
;*****************Variables***********************

Var1    EQU    0Dh          ;give name Var1 to 0Dh
Var2    EQU    0Eh          ;give name Var2 to 0Eh

;***********************************************
```

4. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000    ;setting the beginning
             ;address of the program
```

5. Clear the PORTA and PORTB registers using instruction *clrf*.

```
clrf PORTA          ;clear PORTA register
clrf PORTB          ;clear PORTB register
```

6. Using bit RPO (5th bit) of STATUS register and instruction *bsf* set the work of MC for operating with the registers located in Bank 1:

```
bsf STATUS, 5       ;move to  Bank 1
```

7. Set all port B pins for signal output, set the work of MC for operating with the registers located in Bank 0 and upload binary number 11100000 to the PORTB register:

```
      movlw b'00000000'   ;write a binary number to w
                          ;register
      movwf TRISB         ;move the contents of w
                          ;register into TRISB
                          ;register
      bcf STATUS, 5       ;move to Bank 0
Start movlw b'11100000'   ;write a binary
                          ;number to w register
      movwf PORTB         ;move the contents of w
                          ;register into PORTB
                          ;register
```

8. Enter the fragment of a program which calls the delay loop

```
call Delay          ;call the subroutine Delay
```

9. Using instruction *comf (complement f),* invert PORTB register content:

```
comf PORTB,1 ;invert PORTB register content
```

Comment. The instruction *comf* inverts f register content, i.e. it change "0" to "1" and "1" to "0". In this case the binary number 11100000 in the PORTB register is changed to 00011111. If 1 is written in the operand after comma, the result of the execution of this instruction is written into the f register (in this case into the PORTB register), if 0 – the result is written into the W register.

10. Call the delay loop subroutine and call back the program operation to the line marked by the *Start* label:

```
call Delay  ;call the subroutine Delay
goto Start  ;go to the string of the program
            ;marked by the label Start
```

11. Enter the subroutine of delay loop *Delay*

```
;***************Subroutine******************
Delay
Cycle decfsz Var1,1      ;subtract 1 from variable
                         ;Var1 and when it is
                         ;equal to 0, skip
                         ;instruction goto Cycle
      goto Cycle         ;go to the string
                         ;marked by label Cycle
      decfsz Var2,1      ;subtract 1 from variable
                         ;Var2 and when it is
                         ;equal to 0, skip
                         ;instruction goto Cycle
      goto Cycle         ;go to the string
                         ;marked by label Cycle
      return             ;return to the main program
;*****************************************
```

12. Indicate the end of the program using directive END.

```
END          ;the end of the program
```

13. By choosing *File>Save As* save the program as assembly source file. Before that entitle it. Using *Project>Open*, open project that was created during 1 laboratory work. Mark

assembler file saved in *Source files* folder and remove it by executing *Remove*. Then mark the catalog *Source Files* and executing *Add File* move to project the assembler file of the created program. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file and using MC PIC16F84A development board load it into the MC.

14. Switch the development board to the operating mode (see point 16 of 1st laboratory work) and make sure the program operates properly, i.e. in all pins of PORTB pulse signals with approximately 2,5 Hz frequency are generated. Signals in pins RB0, RB1, RB2, RB3, RB4 must be inverted as compared to the signals in pins RB5, RB6, RB7.
15. Change the instruction *comf* by the instruction *swapf* in the main program.

Comment. Instruction *swapf* (swap nibbles in f) changes the contents of the higher nibble of register f with the contents of lower nibble, i.e. the contents of the four higher bits is moved to the four lower bits and vice versa. In this case the binary number 11100000, which is stored in PORTB register, changes to 00001110. If 1 is written in the operand after comma, the result of the execution of this instruction is saved in f register, i.e. to PORTB register, if 0 – the result is saved in W register.

16. Create the *hex* file of the program and upload it into the MC. Make sure that the program operates properly, i.e. in all pins of PORTB, except pins RB0 and RB4, pulse signals with approximately 2,5 Hz frequency are generated. The signals generated in the pins RB5, RB6 and RB7 must be inverted to the signals in pins RB1, RB2 and RB3. The pins RB0 and RB4 must provide DC voltage, which corresponds to “0”, i.e. close to 0 V (LEDs should not emit the light).
17. Study the instructions of logic functions *andwf, andlw, iorwf, iorlw, xorwf, xorlw* using truth table of the logic operations (Table 7.1).

Table 7.1. The truth table of the logic operations

Combination of numbers	The result of logic operation		
	AND	IOR	XOR
00	0	0	0
01	0	1	1
10	0	1	1
11	1	1	0

18. Investigate instruction *andwf* (AND W with f), which executes logic operation AND. Change the string with the instruction *swapf* in the main program with these two strings:

```
movlw b‘00111111‘   ;write a binary number to w
                    ;register
andwf PORTB,1       ;execute logic operation AND
                    ;between contents of w register
                    ;and PORTB register
```

Comment. Instruction *andwf* executes logic operation AND between numbers of corresponding bits of the registers W and f (in this case PORTB). Register W contains binary number 00111111 and PORTB register – 11100000. The result of the logic operation, according to the table 7.1, must be 00100000. If 1 is written in the operand after comma, the result of the execution of this instruction is written into the f register, if 0 – the result is written into the W register.

19. Create the *hex* file of the program and upload it into the MC. Make sure that the program operates properly, i.e. port B pins RB0, RB1, RB2, RB3, RB4 provide DC voltages that correspond to logical “0”, pin RB5 – voltage that corresponds to logical “1”, and the pulse signals of approximately 2,5 Hz with the same phase are generated in pins RB6, RB7.

20. Investigate instruction *andlw* (AND literal with W). Change the instructions in the main program, which were presented in point 18, by instructions:

```
andlw b'10000000'  ;execute logic operation AND
                   ;between given number and the
                   ;contents of W register
movwf PORTB        ;move the contents of W register
                   ;to the register PORTB
```

Comment. Instruction *andlw* executes logic operation AND between corresponding bits of given number and binary number stored in the register W. In this case the given number is 10000000 and the register W contains binary number 11100000. The result of the logic operation, according to the table 7.1 has to be 10000000 and it is stored in the W register.

21. Create the *hex* file of the program and upload it into the MC. Make sure that the program operates properly, i.e. port B pins RB0, RB1, RB2, RB3, RB4 provide DC voltages that correspond to logical "0", pin RB7 – voltage that corresponds to logical "1", and the pulse signals of approximately 2,5 Hz with the same phase are generated in pins RB6, RB7.
22. In the same manner investigate instructions *iorwf (inclusive OR W with f), iorlw (inclusive OR literal with W), xorwf (exclusive OR W with f), xorlw (exlusive OR literal with W),* which execute logic operations IOR and XOR (7.1 table). Create the *hex* files and upload the programs into the MC PIC16F84A. Make sure that the programs operate properly according to the truth table of the logic operations. Save the *xorlw* instruction investigation program text for the laboratory report. Entitle the program as "Low frequency pulse signal generator based on the *xorlw* instruction".

23. Change a fragment of the main program, which begins with instruction labeled *Start* and ends with *goto Start* by the following fragment based on the instruction *rlf* (rotate left f through carry)*:*

```
Start movlw b‘00000001‘ ;write a binary number into
                        ;w register
movwf PORTB             ;move the content of w
                        ;register to register PORTB
call Delay              ;call subroutine Delay
rlf PORTB,1             ;rotate the content
                        ;of PORTB register to the
                        ;left
      call Delay
      rlf PORTB,1
      call Delay
      rlf PORTB,1
      call Delay
      rlf PORTB,1
      call Delay
      rlf PORTB,1
      call Delay
      rlf PORTB,1
      call Delay
      rlf PORTB,1
      call Delay
      goto Start      ;go to the string marked
                      ;by the label Start
```

Comment. *rlf* instruction performs the rotation of the register contents to the left, i.e. shifts the binary number of the register one bit to the left including the flag bit C, which is stored in the STATUS register. If 1 is written in the operand after comma, the result of the

execution of this instruction is stored in the f register, if 0 – in the W register. The rotation to the left instruction is performed 7 times in the given fragment of the MC program and the result is stored in the f register (in PORTB register). During the execution of the first rotation instruction the binary number in PORTB 00000001 is changed to 00000010. The execution of the next rotation instruction changes the contents of register to 00000100 and so on. During execution of all *rlf* instructions the "1" is moved from the least significant bit (LSB) to the most significant bit (MSB). Shifting binary number to the left multiplies the number by 2. Using this instruction it is possible to achieve 2^n multiplication, where *n* is the number of executions *of rlf* instruction.

24. Create the *hex* file of the program and upload it into the MC. Make sure that the program operates properly, i.e. pulse signals are generated in port B pins, however only one pin at a time has logical "1" voltage. Moreover, the voltage of logical "1" is moving through all the pins of PORTB from RB0 to RB7.
25. Modify the program changing all *rlf* instructions by *rrf (rotate right f through carry)* instructions. Additionally, change the string marked by label *Start* as follows:

```
Start movlw b'10000000'  ;write a binary number
                         ;to w register
```

Comment. *rrf* instruction is analogous to the *rlf* instruction. The only difference is that it performs the rotation of the register contents to the right, i.e. shifts the binary number of the register one bit to the right including the flag bit C, which is stored in a special purpose register STATUS. If 1 is written in the operand after comma, the result of the execution of this instruction is sent to the f register, if 0 – to the W register. The *rrf* instruction is performed 7 times in the given fragment of the program and the result is sent to the f register (to the PORTB register). Therefore, during the first execution

of the instruction *rrf* the binary number 10000000, which is stored in PORTB, is changed to 01000000. During the next execution of *rrf* instruction it changes to 00100000 and so on. After the *rrf* instruction is executed 7 times, the "1" is shifted from most significant bit (MSB) to the least significant bit (LSB). Shifting binary number to the right means division of number by 2. Using this instruction it is possible to divide the number by 2^n, where *n* is the number of executions *of rrf* instruction.

26. Create the *hex* file of the program and upload it into the MC. Make sure that the program operates properly, i.e. pulse signals are generated in all port B pins and only voltage of one pin at a time corresponds to logical "1" voltage. Moreover, the voltage of logical "1" is moving through all the pins of PORTB from RB7 to RB0.
27. Save the text of the program for the report of the laboratory work. Entitle the program "Low frequency pulse generator based on the *rrf* instruction".

4. Report content

1. The aim of the work.
2. Analysis of instructions *comf, swapf, rlf, rrf, andwf, andlw, iorwf, iorlw, xorwf* and *xorlw.*
3. The truth table of the logic operations.
4. The texts and comments of the programs that were created during the execution of tasks given in points 22 and 27.
5. Conclusions.

5. Control questions

1. Explain the purpose of *comf* instruction.
2. What number would we get after the inversion of the number 11110000?
3. Explain the purpose of *swapf* instruction.

4. What number would we get after execution logical operation AND between binary numbers 11110000 and 10000000?
5. Explain the purpose of *rlf* and *rrf* instructions.
6. Explain the purpose of *andwf, andlw, iorwf, iorlw, xorwf* and *xorlw* instructions.
7. What number would we get after the execution of logical operation IOR between two binary numbers 11000111 and 10000100?
8. What number would we get after execution of logical operation XOR between two binary numbers 11010000 and 00010001?

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.6. Laboratory 6

Investigation of arithmetic instructions

1. Aim

To study arithmetic instructions *addlw*, *sublw*, *addwf*, *subwf* and usage of binary and hexadecimal numbers in MC.

2. General knowledge

Physically data are stored in MC in two-state memory cells, i.e. in a form of binary numbers. Therefore, MC performs operations with binary numbers and a single data unit is a bit. Other popular data unit is a nibble, which consists of four bits. Because nibble (physically it is presented in the MC memories by four bit memory cells) stores 2^4=16 different states and usually memory cells (registers) include 2, 4, or 8 nibbles, theretofore, it is convenient to present numbers during the writing MC programs in a hexadecimal form. This enables us to display numbers in a compact form. For example, to display data, stored in nibbles in binary number system one needs to use a 4 digit number, while in hexadecimal number system it is enough to use just 1 digit number, e.g. the binary number 1010 in a hexadecimal number is denoted by letter A, which represents the decimal number 10. Because byte (8 bit register) consists of two nibbles, to display data stored in it, it is enough to use two digit hexadecimal numbers. E.g. number (byte) 11111111 stored in MC´s data memory register, in hexadecimal number system is represented by a number FF, which is equal to 255 in a decimal number system (necessary to remember, that first number in the scale of notation that is used in the digital systems is not 1, but 0).

In spite of the fact that number depiction in binary scale of notation is not so compact, often it is more informative than these one presented in a hexadecimal form, because it directly shows the states of bit cells of the memory register. As an example, to set “0”

voltage levels in PORTB pins RB0, RB2, RB4, RB6 and voltage "1" in the remaining pins, the binary number 10101010 must be written as operand of the appropriate instruction. The binary number represents clearly what voltage levels are in which pins. On the other hand, using hexadecimal scale of notation, the hexadecimal number AA as instruction operand should be used in such a case. However, it does not directly represent that information.

Binary and hexadecimal number system equivalence is presented in the table 7.2.

Table 7.2. Decimal number equivalence

Decimal number	Binary number	Hexadecimal number
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Binary, octal, decimal and hexadecimal numbers are used for writing of MCs programs. Numbers must be written between quotes,

and appropriate letter or symbol combination must be written before it. Binary numbers are indicated by the letter b, e.g. b´10101010´, for octal – letter o, e.g. o´52´. To indicate hexadecimal numbers, letter h or symbols combination 0x, (zero and x) is used, e.g. h'9F´ or 0x9f. For decimal number letter d is used, e.g. d´17´. The capital letters can be used for numbers indication as well.

To determine eight segment binary number $b_7b_6b_5b_4b_3b_2b_1b_0$ value in decimal form the formula $\mathbf{D_b = b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0}$ is used. As an example, binary number 10101010 value in decimal number system is $D_{10101010} = 1x2^7 + 0x2^6 + 1x2^5 + 0x2^4 + 1x2^3 + + 0x2^2 + 1x2^1 + 0x2^0 = 128 + 0 + 32 + 0 + 8 + 0 + 2 + 0 = 170$. The value of hexadecimal number X_1X_0 in decimal scale of notation can be calculated using formula $\mathbf{D_X = X_1 16^1 + X_0 16^0}$. E.g. hexadecimal number AA value in decimal number system $D_{AA} = Ax16^1 + Ax16^0$. Hexadecimal number A corresponds to decimal number 10 (table 7.2), therefore $D_{AA} = 10x16^1 + 10x16^0 = 160 + 10 = 170$. We see that binary number 10101010 corresponds to hexadecimal number AA. This can be found using data presented in the Table 7.2, where it can be seen that hexadecimal number A corresponds the binary nibble 1010. Since AA is two digit hexadecimal number, it is obvious that it contains two nibbles where every digit corresponds to one nibble. Consequently, two digit hexadecimal number AA corresponds to eight digit binary number 10101010 in the binary system. Similarly it can be found that hexadecimal number C9 corresponds to binary number 11001001. The value of this number in decimal system equals to 201.

3. Task

To create, test and investigate the MC PIC16F84A programs dedicated to adding and subtracting of numbers.

4. Proceeding

1. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs.

Type the text with title of the program, author, type of MC and resonator frequency in the beginning of the program:

```
;**********Program for adding of numbers**********
;******************N. Surname*********************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;************************************************
```

2. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC.

```
LIST p=16F84              ;Setting type of MC
#INCLUDE <p16F84a.inc>    ;Calling the file,
                          ;with the definitions of
                          ;special purpose registers
_ _ CONFIG _ XT _ OSC&    ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF& _ CP _ OFF;
```

3. Define the names for registers:

```
;****************Variables**********************

Var1   EQU   0Eh          ;give name Var1 to 0Ch

;************************************************
```

4. Using assembler directive ORG set the beginning address of the program:

```
ORG 0×000         ;shows the initial address
                  ;of the program
```

5. Clear the PORTA and PORTB registers using instruction *clrf*:

```
clrf PORTA          ;clear PORTA register
clrf PORTB          ;clear PORTB register
```

6. Using bit RPO (5th bit) of STATUS register and instruction *bsf* set the operation of MC for operating with the registers located in Bank 1:

```
bsf STATUS, 5       ;move to  Bank 1
```

7. Set all port B pins and RA2 – RA4 pins of port A for signal output. Set port A pins RA0, RA1for signal input and switch MC for operating with the registers located in to Bank 0:

```
movlw b‘00000000‘     ;write a binary number to w
                      ;register
movwf TRISB           ;move the contents of w
                      ;register to TRISB register
movlw b‘00011‘        ;write a binary number to w
                      ;register
movwf TRISA           ;move the contents of w
                      ;register to TRISA register

bcf STATUS, 5         ;move to Bank 0
```

8. Enter fragment of the program with the addition instruction *addlw*, (*add literal and W*):

```
Start movlw b´00001100´  ;write binary number into
                         ;the register w
```

```
addlw b´11100001´  ;add given number to the
                   ;register w contents
movwf PORTB        ;move register w contents
                   ;to the register PORTB
                   ;(sum of the numbers is
                   ;transferred to port B)
goto Start         ;go to the string, which is
                   ;marked by the label Start
```

9. Indicate the end of the program using directive END.

```
END   ;the end of the program
```

Comment. The program fragment presented above adds given number *l* with the content of the W register using instruction *addlw*. The result is sent to the W register and then is stored in the register PORTB.

10. Save the program as assembly source file by choosing *File>Save*. Before that entitle it. Open the project that was created during 1 laboratory work by using *Project>Open*. Mark assembler file saved in the source files folder and remove it by executing *Remove*. Then mark the catalog *Source Files* and executing *Add File*, move to project the assembler file of the created program. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the hex file and using MC PIC16F84A development board load it into the MC.
11. Switch the development board to the operating mode (see point 16 of 1st laboratory work) and observing the LEDs connected to the port B pins, make sure that the program operates properly, i.e. the obtained sum of binary numbers is correct.

12. Make the addition of the several other numbers, also entering them in decimal and hexadecimal numbering systems.
13. Change instruction *addlw* by the subtraction instruction *sublw* (*subtract W from literal*) in the previously created program. The *sublw* instruction subtracts contents of the W register from the given number *l* and saves the result in the W register.
14. Perform the subtraction operations with several other numbers, when *l* is bigger than the number loaded into the W register, i.e. when the result number is positive. Load the programs into MC memory and, by the help of LEDs connected to the port B pins assure that obtained subtraction results are correct.
15. Subtract binary number 00000101 (number stored in the W register) from the binary number 00000010 (number *l*), i.e. investigate the case when the subtraction result is negative. Load the program into MCs memory and assure that the result of subtraction is 11111101.

Comment. The addition operation with the negative number is used in the MCs instead of subtraction, i.e. result = *l* + (– W). Negative W value is calculated using the equation – W = (W $\oplus$ 0xFF) + 1, where $\oplus$ sign indicates logical *exclusive OR* ("sum in modulus 2") operation with values that are in the brackets (because number 0xFF in binary system coincides with the number 11111111, this operation performs the inversion of W register content). Whole expression for calculation of subtraction is as follows: **result = l + (W $\oplus$ 0xFF) + 1**. The subtraction result obtained in the case, which was presented in the point 15 of this laboratory work: **result** = 00000010 + (00000101 $\oplus$ 11111111) + 00000001 = 00000010 + 11111010 + 00000001 = 11111101.

The sign of subtraction result is indicated by the Z, DC and C flag bits located in the STATUS register. If the subtraction result is positive, flag bits obtain the values C = 1, Z = 0, if result is zero Z = 1, if result is negative C = 0, Z = 0.

16. Make subtraction operations with several numbers, when *l* is smaller than the number loaded into the W register, i.e. when the result is negative. Load the programs into MC memory and by the help of LEDs connected to the port B pins, using equation presented in the point 15, assure that obtained subtraction results are correct.
17. Save the program for the report of the laboratory work.
18. Modify the number addition program in such a way that not just result, but also the numbers that are added up could be displayed by the LEDs when the keyboard buttons are pushed down. Make the following changes of the program for this purpose:

a) The program fragment, which defines the names for registers, modify as follows:

```
Num1 EQU 0Ch       ;give name Num1 to 0Ch
Num2 EQU 0Dh       ;give name Num2 to 0Dh
Var1 EQU 0Eh;give name Var1 to 0Eh
```

b) The part of the program, which begins with the label *Start* and ends with the instruction *goto Start*, replace with the following program fragment:

```
Start movlw b´00010001´    ;set value of variable Num2
      movwf Num2
      movlw b´01100010´    ;set value of variable Num1
      movwf Num1
      movf Num1,0          ;move content of the Num1
                           ;into the W register
      movwf PORTB          ;move content of the Num1
                           ;into the register PORTB
      btfss PORTA,0        ;read the signal of pin RA0
                           ;and if the result is 1
```

```
                        ;skip the next instruction
      call Subrout1     ;call subroutine  Subrout1
      btfss PORTA,1     ;read the signal of pin RA1
                        ;and if the result is 1
                        skip the next instruction
      call Subrout2     ;call subroutine Subrout2
      call Delay        ;call subroutine Delay
      goto Start        ;go to the string marked by
                        ;the label Start
```

c) Enter the subroutine:

```
;*****************Subroutine1*******************
Subrout1
movf Num2,0       ;move the Num2 value into the W
                  ;register
movwf PORTB       ;move the Num2 value into the
                  ;register PORTB
return            ;return to the main program
;***********************************************
```

d) Enter the subroutine with the addition instruction *addwf* (*Add W and f*):

```
;*****************Subroutine2*******************
Subrout2
movf Num2,0       ;move the Num2 value into the W
                  ;register
addwf Num1,0      ;add Num1 and Num2 and save the
                  ;result in W register
movwf PORTB       ;move the contents of W register
                  ;into the register PORTB
return            ;return to the main program
;***********************************************
```

e) Enter the delay loop subroutine:

```
;*****************Subroutine3********************
Delay
Cycle decfsz Var1,1       ;subtracts 1 from variable
                          ;Var1 and when it becomes
                          ;equal to 0 skip the next
                          ;instruction goto Cycle
goto Cycle                ;go to the string with
the                       ;label Cycle
return                    ;return to the main program
;************************************************
```

Comment. The modified program adds two binary numbers, which are set as *Num1* and *Num2*. Addition is performed using the subroutine *Subrout2*. The instruction *addwf* is used for this purpose. It adds contents of the registers W and f (if in the operand after the comma is written 1, then this instructions result is loaded into the register f, if 0 – into the register W). The numbers that are added and addition result one by one are loaded into the register PORTB, so they can be shown in the binary counting system by the help of LEDs, which are connected to the PORTB. Visible is that number, which remains much longer in the register PORTB as compared to the two others. The delay loop subroutine *Delay* is used for this purpose. The number that should be displayed by the LEDs connected to the port B is chosen by pressing the buttons of the development board keyboard. The voltage of pins RA0 and RA1 are changed from "0" to "1" by pressing of buttons. The button M1 commutates the voltage of both pins RA0 and RA1 and M2 commutates voltage of pin RA1. The MC executing instruction *btfsss* (*bit test f skip if set*) checks the state of RA0 and RA1 pins and if 1 is detected the next instruction is skipped. If the button M2 is pressed, the subroutine *subrout2* is skipped and delay loop is executed after the subrou-

tine *subrout1*, therefore the LEDs display the number *Num2*. Both subroutines *subrout1* and *subrout2* are skipped if the button M1 is pressed. Therefore, the delay loop is executed after instruction, which loads the contents of register *Num1* into the PORTB register, i.e. LEDs show the number saved in the register *Num1*. If the buttons are not pressed, LEDs display the sum of the numbers. This happens because the delay loop is executed after the subroutine *subrout2*, which performs the addition of numbers.

19. Create the hex file of the program and load it into the MC. Make sure that the program operates properly, i.e. the entered numbers and the result of addition are right. The number saved in the register *Num1* must be displayed if the button M1 is pressed. If the button M2 is pressed, the number stored in the register *Num2* should be shown. The sum of *Num1* and *Num2* must be displayed if the buttons are not pressed.
20. By changing values of variables *Num1* and *Num2* in the created program perform several adding operations with equal to each other numbers, i.e. 00001111, 01010101 and etc. Verify that the sum is twice bigger than initial numbers, i.e. the same as initial number just rotated to the left by one digit.
21. Perform the number adding operations with several other numbers entering the numbers in binary, decimal and hexadecimal number systems.
22. Substitute instruction *addlw* by subtraction instruction *subwf* (subtract W from f) in subroutine *Subrout2*.

Comment. Instruction *subwf* subtracts content of the W register from the *f* register. If in the operand after the comma is written 1, then this instructions result is loaded into the register f, if 0 – into the register W.

23. Subtract number 01010101 (*Num2*) from the twice higher number 10101010 (*Num1*). Load the program into the MC and verify that attained result is the same as smaller number (as number loaded in *Num2*). This fact shows that division

of binary number by two coincides with the rotating of this number to the right by one bit.

24. Make subtraction operations with various numbers, selecting them in such are way that the result would be negative and positive. Enter numbers every time in different number system: binary, decimal and hexadecimal. Load the programs into the MC and verify that entered numbers and the results are correct.
25. Save the text of the program for the report of the laboratory work. Entitle it "Improved number addition program".

5. Report content

1. The aim of the work.
2. Analyses of the MV16F84A instructions *addlw, sublw, addwf* and *subwf*.
3. Table of the binary and hexadecimal numbers equivalence.
4. The texts and comments of the programs that were saved during the execution of tasks given in points 17 and 25.
5. Conclusions.

6. Control questions

1. Explain the purpose of instructions *addlw* and *addwf*.
2. What is the nibble?
3. Explain the purpose of instructions *sublw* and *subwf*.
4. How are binary, octal, decimal and hexadecimal numbers noted in the text of the MC program?
5. Write the formula for the calculation of the number in the decimal counting system from the number presented in the octal system.
6. In what scale of notation the data are presented on the MC pins?
7. Explain how subtraction of two numbers is calculated in MC?

8. Write the formula for calculation decimal number from two segment hexadecimal number.
9. How will change the value of the binary number if it is rotated to the left by 3 digits?

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.7. Laboratory 7

Creating and investigation of timer programs

1. Aim

To study instructions *decf*, *incf* and creating principles of timer programs.

2. Task

To create and analyze the timer programs for MC PIC16F84A when time is displayed in binary scale of notation by the LEDs connected to MC ports.

3. Proceeding

1. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. Type text with title of the program, author, type of MC and resonator frequency:

```
;****************** Stopwatch*******************
;******************N. Surname********************
;Microcontroller PIC16F84A
;Crystal resonator frequency 4 MHz
;***********************************************
```

2. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC:

```
LIST p=16F84              ;Setting type of MC
#INCLUDE <p16F84a.inc>    ;Calling the file,
                          ;with the definitions of
                ;special purpose registers
```

```
_ _ CONFIG _ XT _ OSC&     ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

3. Define the names for registers (introduce variables):

```
;*****************Variables**********************

Var1    EQU    0Ch          ;gives name Var1 to 0Ch
Var2    EQU    0Dh          ;gives name Var2 to 0Dh
Var3    EQU    0Eh          ;gives name Var3 to 0Eh
Var4    EQU    0Fh          ;gives name Var4 to 0Fh
Var5    EQU    10h          ;gives name Var5 to 10h

;************************************************
```

4. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000         ;shows the initial address
                  ;of the program
```

5. Clear the PORTA and PORTB registers:

```
clrf PORTA          ;clear PORTA register
clrf PORTB          ;clear PORTB register
```

6. Set the work of MC for operating with the registers located in Bank 1:

```
   bsf STATUS, 5  ;move to Bank 1
```

7. Set B and A ports pins configuration via TRISB and TRISA registers:

```
movlw b‘00000000‘   ;write the binary number into
                    ;W register
movwf TRISB         ;move the content of W regis-
                    ter ;into TRISB register (set
                    all ;port B terminals for data
                    ;output)
movlw b‘00001‘      ;write the binary digit to
                    ;W register
mowf TRISA          ;move register W content to
                    ;TRISA register (set port A
                    ;output terminal RA0 for data
                    ;input and terminals RA1, RA2,
                    ;RA3, RA4 – for data output)
   bcf STATUS, 5    ;move to Bank 0
```

8. Enter following fragment of the program:

```
Start call Timeunit     ;call subroutine Timeunit
      incf PORTB,1      ;increment PORTB register
                        ;content by 1
      goto Start        ;go to the program string
                        ;marked by label Start
```

Comment. The program fragment of the program given above presents cycle, the execution duration of which and, as a consequence, the duration of stopwatch time unit is determined by the subroutine *Timeunit.* Instruction *incf* (*Increment f*) increments register f content (in this case PORTB) by 1. If in the operand after the comma is written 1, then the result is loaded into the register f, if 0 – into the register W. For example, if execution duration of sub-

routine *Timeunit* is 1 second, then content of the register PORTB is incremented by one every second. Before the execution of the cycle, the content of the register PORTB is cleared, therefore, in the beginning the binary number 00000000 is stored in register PORTB. During the 255 cycles the contents of PORTB register reaches the number 11111111 and in the next cycle overflows, i.e. the counting again begins from the number 00000000. Content of the register PORTB, which represents time, is shown by the LEDs connected to port B pins.

The subroutine *Timeunit* is called before the instruction *incf PORTB,1* because firstly time interval equal to time unit should elapse and just then the time should be displayed by the LEDs connected to port B.

9. Enter subroutine *Timeunit*:

```
;*****************Timeunit subroutine*************
Timeunit
       movlw d'5'    ;write the binary digit into
                     ;W register
       movwf Var3    ;move register W content
                     ;to Var3
Cycle1 decfsz Var1,1;substract 1 from variable Var1
                     ;and if result is 0 skip the
                     ;next (goto Cycle1)instruction
       goto Cycle1   ;go to the string marked by
                     ;the label Cycle1
       btfsc PORTA,0;read the signal from terminal
                     ;RA0 and if the result is 0
                     ;skip next instruction
       call Stop     ;call subroutine Stop
       decfsz Var2,1;substract 1 from variable
                        ;Var2 and if result is 0
                     skip ;the next instruction
```

```
        goto Cycle1   ;go to the string marked
                      ;by the label Cycle1
        Decfsz Var3,1  ;subtract  1  from  variable
                       ;Var3 and if result is 0 ;skip
                       the next instruction
        goto Cycle1    ;go to the string marked
                       ;by the label Cycle1
        return         ;return to the main program
;************************************************
```

Comment. The subroutine *Timeunit* includes triple delay loop. The execution time of this delay loop is approximately $3x10^{-6}$ s x 256 x 256 x 5 ≈ 1 s. Method of precise calculation of delay loop execution duration is described in the laboratory 4. The instruction *btfsc* in the subroutine checks the voltage of the terminal RA0. If 0 is read the instruction *btfsc* forces to skip the next instruction of the program, if 1 – the next instruction is executed. Since button M3 controls RA0 terminal voltage, the time calculation can be stopped and fixed. While the button is not pushed down the voltage that corresponds to "0" (voltage close to 0 V) is conducted to the RA0 terminal of the port A. Because of this, the instruction *call Stop* is skipped. If the button M3 is pushed down, the voltage that corresponds to "1" (voltage close to 5 V) is conducted to the terminal and, as a consequence, after the instruction *btfsc PORTA,0* the instruction *call Stop* is executed, i.e. the subroutine *Stop* is called. The microcontroller starts to execute timeless cycle in the *Stop* subroutine, because of this, the change of the PORTB register content is stopped and LEDs connected to the port B display the time that was at the moment when the button was pushed down.

Instructions *btfsc* and *call Stop* are included in the second cycle of the *Timeunit* subroutine. This guarantees the stopping of the time count with the acceptable delay on the one hand and, on the other hand, the execution of the instruction *btfs* influences the duration of

subroutine *Timeunit* performance marginally in such a case. As an example, if these instructions would be inserted in the first cycle, the time count stopping would be performed with the minimal delay, however, the execution *Timeunit* subroutine would be lengthen twice, i.e. the time count unit would increase up to 2 seconds.

10. Enter the *Stop* subroutine:

```
;*****************Stop subroutine*************
Stop
Start1 movlw b'01110'   ;write the binary number
                        ;into W register
       movwf PORTA      ;move the content of W ;reg-
                        ister to PORTA register
Cycle2 decfsz Var4,1    ;subtract 1 from variable
                        ;Var4 and if the result is 0
                        ;skip next instruction
       goto Cycle2      ;go to the string marked by
                        ;label Cycle2
       decfsz Var5,1    ;subtract 1 from variable
                        Var5 and if the result is 0
                        ;skip next instruction
       goto Cycle2      ;go to the string marked by
                        ;label Cycle2
       movlw b'00000'   ;write the binary digit ;into
                        W register
       movwf PORTA      ;move the content of W ;reg-
                        ister to PORTA register
Cycle 3 decfsz Var4,1   ;subtract 1 from variable
                        ;Var4 and if the result is 0
                        ;skip next instruction
       goto Cycle3      ;go to the string marked by
                        ;label Cycle3
```

```
        decfsz Var5,1    ;substract 1 from variable
                         ;Var5 and if the result is 0
                         ;skip next instruction
        goto Cycle3      ;go to the string marked by
                         ;label Cycle3
        goto Start1      ;go to the string marked by
                         ;label Start1
        return           ;return to the
                         ;main program
;***********************************************
```

Comment. Subroutine *Stop* practically is the same low frequency pulse generator program, which was analyzed in the Laboratory 3. The subroutine is executed if briefly the M3 button is pushed down. The subroutine presents the timeless cycle and generates low frequency pulse signal in the terminals RA1–RA3. Blinking LEDs connected to these terminals show that the time count has been stopped and the time is fixed. To start the count of the time from the beginning again (to leave the timeless cycle of the subroutine *Stop*), the *Reset* must be done for the microcontroller (the *Reset* button on the microcontroller development board must be pushed down).

11. Indicate the end of the program using directive END:

```
END   ;end of the program
```

12. Save the program as assembly source file. Before that entitle it. Open the project that was created during 1 laboratory work. Remove assembler file saved in it, then mark the catalog *Source Files* and executing *Add File*, move to project the assembler file of the created program.
13. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file of the program and using MC PIC16F84A development board load it into the MC.

14. Switch the development board to the operating mode (see point 16 of 1 laboratory work) and observing the LEDs connected to the B port assure that the stopwatch program is working properly, i. e. content of the register PORTB increases by one every second and after overflow of the register time count starts again from zero.

Push down the button M3 during the stopwatch operation and assure that time stopping feature is working, i.e. the count of the time is stopped without delay and the LEDs connected to the terminals RA1–RA3 start to blink.

15. Push down the *Reset* button and assure that after release of this button time is calculated from the beginning (from the 00000000) again and that the number 00000001 at the port B appears approximately after the one second after the button was released.
16. Change program in such a way that the time unit would be: 0,2s; 2s. Create the *hex* file and load the program into the MC. Assure that the program works correctly.
17. Save the stopwatch program with the 0,2s time unit for the laboratory report.
18. Move the instructions *btfsc PORTA,0* and *call Stop* in the subroutine *Timeunit* just before the string with the instruction *return* (time unit, i.e. subroutine *Timeunit* execution duration, has to be 2s). Load the program into the MC and assure that it is necessary not only to press button M3 but it is needed to keep it pressed for awhile as well to stop the time count. Explain why significant response delay occurs after these changes.
19. Counterchange the instruction *call Timeunit*, which is marked by the label *Start* with the subsequent instruction *incf PORTB,1* (the label *Start* must be in line with the instruction *incf* PORTB,1). Load the program into the MC and, by pressing the *Reset* button, assure that number 00000001 in the port B

occurs immediately after the button release, as to correctly calculate the time, necessary that it should occur just after the delay equal to time unit. Explain why this modification produces such response change.

20. Change the created program fragment, which begins with the string marked by the label *Start* and ends with the instruction *goto Start* by the following countdown timer fragment:

```
      movlw d'15'  ;write the decimal digit into
                   ;W register
                   ;(set the initial time of ;count-
                   down timer)
      movwf PORTB ;move the content of the W ;reg-
                   ister into the register PORTA
Start call Timeunit   ;call subroutine Timeunit
      decf PORTB,1    ;subtract 1 from the content
                      ;of the register PORTB
      btfss STATUS,2  ;check the content of ;regis-
                      ter STATUS Z bit (2 ;bit) skip
                      the next ;instruction if 1 is
                      found
      goto Start      ;go to the string marked by
                      ;the label Start
      call Stop       ;call subroutine Stop
```

Comment. The number loaded into the register PORTB is decreased every second by one using the instruction *decf (Decrement f)* in this program fragment. The execution time of cycle and, as a consequence, the countdown timer time unit is set by the subroutine *Timeunit*. When the content of the register PORTB becomes zero, the 1 is loaded into the Z bit (2nd bit) of the STATUS register, therefore, instruction *btfss STATUS,2* forces to skip the instruction *goto Start*. Because of this, the execution of program cycle is over

and subroutine *Stop* is called. Since the subroutine *Stop* presents the endless program cycle the time count is stopped. The Reset button must be pushed down to exit this cycle and again start to count the time by the countdown timer.

21. Set the value of variable Var3 in the subroutine *Timeunit* equal to 5 (make the time unit equal to 1s).
22. Load the created stopwatch program into MC and assure that it works properly, i.e. after release of the button *Reset*, the binary number 00001111 is shown and after that it is periodically decreased by 1 until it reaches the result 0. This fact should be indicated by blinking of LEDs connected to the terminals RA1–RA3.
23. Set various countdown initial time values also in binary and hexadecimal formats. Load the programs into the MC and assure that they work correctly.
24. Improve (reduce) created program of countdown timer using just appropriate one instruction instead of two instructions *decf PORTB,1* and *btfss STATUS,2*. Load the program into the MC and assure that it works correctly. Name the program "Countdown timer" and save it for the laboratory report.
25. Change the program in such a way that after the countdown timer reaches zero value, the time count would not be stopped but would continue time countdown. The duration of the time unit should be 0,2s. Load the program and test it.

4. Report content

1. The aim of the work.
2. Analysis of instructions *decf* and *incf*.
3. Description of the STATUS register Z flag bit purpose.
4. The texts and comments of the programs that were created during the performance of tasks given in points 17 and 24.
5. Conclusions.

5. Control questions

1. Explain the purpose of instructions *decf* and *incf*.
2. How many MC clock signal periods are needed to execute the instruction *decfsz*?
3. What is the purpose of the Z flag bit?
4. What determines the stability of the time unit?
5. How many machine cycles take execution of *btfss* instruction*?*
6. Which one instruction can be used instead of two instructions *decf* and *btfss*?
7. What is the purpose of the subroutine *Timeunit*?
8. Comment the structure of the subroutine *Timeunit*.
9. What is the purpose of the subroutine *Stop*?
10. Comment the structure of the subroutine *Stop*.

References

1. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005. 86 p.
2. PIC16F84 tutorial, http://home.planet.nl/~midde639/tutorial.pdf, 2005. 58 p.

7.8. Laboratory 8

Investigation of control of Liquid Crystal Display LCD1601LC

1. Aim

To study the Liquid Crystal Display (LCD) LCD1601LC control using microcontroller.

2. Task

To create, test and investigate the MC PIC16F84A programs for control of LCD1601LC display.

3. Proceeding

1. Analyze the parameters of liquid crystal display LCD1601LC, purpose of terminals, memories, their purpose and control instructions.
2. Open the MPLAB IDE software. Using *File>New* open MPLAB Editor window intended for editing of MC programs. In the beginning of the program type text, with title of the program, author name, type of MC and resonator frequency.

```
;***************Control of LCD display*************

   ;******************N. Surname*********************
   ;Microcontroller PIC16F84A
   ;Crystal resonator frequency 4 MHz
;************************************************
```

3. Using assembler directives LIST, INCLUDE and _CONFIG set the type of MC, call the file with the definitions of special purpose registers and configure the MC:

```
LIST p=16F84        ;Setting type of MC
#INCLUDE <p16F84a.inc>    ;Calling the file,
```

```
                              ;with the definitions of
                        ;special purpose registers

_ _ CONFIG _ XT _ OSC&      ;MC configuration
_ WDT _ OFF& _ PWRTE _ OFF&
_ CP _ OFF;
```

4. Define the names for registers (introduce variables):

```
;*****************Variables***********************

Var0    EQU    0Ch          ;gives name Var1 to 0Ch
Var1    EQU    0Ch          ;gives name Var1 to 0Dh
Var2    EQU    0Dh          ;gives name Var2 to 0Eh
Var3    EQU    0Eh          ;gives name Var3 to 0Fh
Var4    EQU    0Fh          ;gives name Var4 to 10h
Var5    EQU    10h          ;gives name Var5 to 11h
Var6    EQU    0Ch          ;gives name Var1 to 12Ch

;***********************************************
```

5. Introduce the constants:

```
;******************* Constants********************

E EQU 2      ;the value 2 is assignet to constant E
             ;(LCD display pin E is controlled
             ;by RA2 pin)
RS EQU 3     ;the value 3 is assignet to constant RS
             ;(LCD display pin RS is controlled
             ;by RA3 pin)

;***********************************************
```

6. Using assembler directive ORG set the beginning address of the program.

```
ORG 0×000          ;shows the initial address
                   ;of the program
```

7. Clear the following registers:

```
clrf PORTA          ;clear PORTA register
clrf PORTB          ;clear PORTB register
   clrf  Var1                  ;clear Var1 register
   clrf  Var3                  ;clear Var3 register
   clrf  Var5                  ;clear Var5 register
```

8. Set the work of MC for operating with the registers located in Bank 1:

```
   bsf STATUS, 5               ;move to Bank 1
```

9. Set all port A and port B pins for signal output via TRISB and TRISA registers. Clear the RBPU bit (7th bit) of the OPTION_REG:

```
movlw b‘00000000‘     ;write the binary digit to
                      ;W register
movwf  TRISB          ;move the content of W ;regis-
                      ter into TRISB register
movlw b‘00000‘        ;write the binary digit to
                      ;W register
movwf  TRISA          ;move the content of W ;regis-
                      ter into TRISA ;register

bcf  OPTION _ REG,7   ;connect the pull up
```

```
                    ;resistors to B port pins
bcf  STATUS, 5      ;move to Bank 0
```

10. Enter the LCD display initiation program:

```
;************LCD initiation*****************
call Delay200ms          ;wait for end of
                         ;transition processes
movlw b'00110000'        ;write the binary digit to
                         ;W register
call Instruction         ;call subroutine, which
                         ;executes instruction for
                         ;LCD display controller
call Delay5ms            ;call 5ms delay subroutine

movlw b'00110000'
call Instruction
call Delay5ms

movlw b'00110000'
call Instruction
call Delay5ms

movlw b'00111000'   ;communication with the MC
                    ;using 8 data/instructions
                    ;lines
call Instruction
call Delay5ms
movlw b'00001000'   ;turn off the display
call Instruction
call Delay5ms

movlw b'00000001'   ;clear display
```

```
call Instruction
call Delay5ms

movlw b'00000100'
call Instruction
call Delay5ms

movlw b'00001100'  ;turn on the display, cursor
call Instruction;  and turn off cursor's blinking
call Delay5ms
```

Comment. The fragment of the program presented above is created according the recommendations of LCD1601LC display manufacturer. The set of instructions is executed to initiate the operation of the display. The display controller is much slower as compared to the MC, because of this the 5ms delay loops are inserted in the MC program between the instructions that are sent to display. The last instructions presented in the program set the configuration of display, i. e. turn on the display and set the status of cursor.

11. Create the program fragment, which sends the symbols to LCD display:

```
;********** Loading of symbols***************

movlw b'00000001'         ;clear display
call Instruction
call Delay5ms

movlw b'10000100'         ;indicates display DDRAM
                          ;memory address
                          ;for sending of symbol code
call Instruction          ;call subroutine
                    ;Instruction
```

```
call Delay100mks

movlw 'A'           ;indicates the code of
                    ;the letter A symbol that is send
                    ;to display
call Data           ;calls subroutine, which loads
                    ;data of symbol into display
call Delay100mks

goto $              ;endless cycle, which involves
                    ;just this instruction
```

Comment. The first instruction of the program fragment presented above clears the display (deletes the symbols in all display segments) and the second one shows address of DDRAM register, where the symbol code will be sent. Symbol place on the display is determined by its code place in the display DDRAM memory. In this case the symbol should be loaded into 0000100 (in decimal notation 04) address, so the symbol will be shown in 5 segment of display (see description of the display in the sources [1, 2] given in the reference list). After the instruction, which determines the segment of the display where the symbol will be placed, the symbol code must be sent. The assembler of PIC MC allows working with ASCII symbols, so instead of the code the symbol can be written as the operand of the instruction. In the analyzed fragment the symbol of letter A is given. Symbols and their codes for the LCD1601LC display are given in the data sheets of the display [1, 2].

Instruction *goto $* presents endless loop, which involves just this instruction. This instruction is used to stop the program in the right place.

Some delay between the instructions that are sent to the display has to be introduced. This is necessary because the controller of the display is slow. Therefore, between the instructions or data,

which are sent to the display, the delay loop subroutines have to be executed. For example, it takes up to 1.6 ms to clear the display. The time that is needed to load the instruction or data into the display controller is 40 μs. The clock frequency of display controller is determined by the internal RC network. Because of this the instruction execution duration depends on the dispersion of R and C values, on supply voltage and ambient temperature variations. Therefore, for reliable control of the display the delays introduced by the MC must be longer than these discussed above.

12. Enter instruction subroutine:

```
;************Instruction Subroutine **************

Instruction
bcf PORTA,RS       ;switch the LCD to instruction
                   ;receiving mode
movwf PORTB        ;send the instruction to display
nop
nop
bsf PORTA,E        ;write the instruction
nop
nop
nop
bcf PORTA,E        ;ignore signals from MC
nop
nop
return
```

Comment. The MC executing this subroutine generates signals, which force the display controller to execute the instruction. The set of signals must be formed according to recommendations of display manufacturer. At the beginning the "0" must be sent to the display terminal RS, which shows that to the data/instructions terminals of the display D0-D7, which are connected to port B of MC, instruc-

tion signals will be sent. After that the “1” (enable signal) must be sent to display terminal E to enable the loading of instruction. After the instruction loading is accomplished, the “0” must be sent to terminal E, which forbids display to receive signals. Instructions *nop* introduce the delays between the instructions.

13. Enter data (symbol, which must be indicated, code) subroutine:

```
;*****************Data subroutine***************

Data
bsf PORTA,RS        ;switch the display to data
                    ;receiving mode
movwf PORTB         ;send the data to display
nop
nop
bsf PORTA,E         ;write the data
nop
nop
nop
bcf PORTA,E         ;ignore signals from MC
nop
nop
return
```

Comment. The MC executing this subroutine generates the signals, (the code of the symbol that has to be indicated), which are sent to the display controller. The structure of this subroutine is the same as this one used in the instruction subroutine. The difference is just that instead of “0”, the “1” has to be sent to terminal RS, which shows that the data signals will be sent to the data/instructions terminals of the display.

14. Enter the following subroutines of delay loops:

```
;**********Subroutines of delay loops************

;******************100mks delay*****************
Delay100mks
movlw d'33'
movwf Var0
cycle0decfsz Var0,1
goto cycle0
return

;*********************5ms delay*****************
Delay5ms
movlw d'7'
movwf Var2
cycle2decfsz Var1,1
goto cycle2
decfsz Var2,1
goto cycle2
return

;******************200ms delay******************
Delay200ms
Cycle3decfsz Var1,1
goto cycle3
decfsz Var3,1
goto cycle3
return

;****************0.8-200ms delay*****************
Delay1
movlw d'19'
movwf Var4
```

```
cycle4decfsz Var1,1
goto cycle4
decfsz Var4,1
goto cycle4
return

;*****************0.2-50s delay******************
Delay2
movlw d'5'
movwf Var6
cycle5cfsz Var1,1
goto cycle5
decfsz Var5,1
goto cycle5
decfsz Var6,1
goto cycle5
return
;***********************************************
```

Comment. The subroutines, which present the delay loops with the various delay duration, are needed to create and analyze LCD display control programs. The principles of the delay loops are discussed in the details in laboratories 3 and 4.

15. Indicate the end of the program using directive END:

```
END                 ;end of the program
```

16. Save the program as assembly source file. Before that entitle it. Open the project that was created during 1 laboratory work. Remove assembler file saved in it, move to project the assembler file of the created program. Executing the actions presented in the 7 to 15 points of the 1st laboratory work create the *hex* file of the program and using MC PIC16F84A development board load it into the MC.

17. Switch the development board to the operating mode (see point 16 of 1 laboratory work) and make sure that created LCD control program works, i.e. letter A is indicated in the 5th segment of LCD display. The contrast of the display can be adjusted using the potentiometer, which is next to the display.
18. Using created program load any symbol into 12th segment of display. Load program into the MC and make sure that symbol is indicated in chosen segment of display.
19. Making the changes in the initiation fragment of the program, change the configuration of the display in such a way that the cursor would be shown. Load the program into the MC and make sure that the cursor appeared near the symbol.
20. Making the changes in the initiation fragment of the program, change the configuration of the display in such a way that the blinking cursor would be shown. Load the program into the MC and make sure that the blinking cursor near the symbol is shown.
21. Analyze instruction, which shifts symbol by one segment. For this purpose, modify the program fragment presented in the 11th point of this laboratory by entering the label *Start* on the left of the first line. Additionally, remove instruction *goto $* and instead of this instruction enter following program fragment:

```
call Delay2              ;call delay subroutine
movlw b'00011000'        ;shift symbol to the left
call Instruction
call Delay5ms
call Delay2
goto Start
```

22. Load the program into the MC and make sure that first of all symbol appears in the chosen segment of the display, and after that it is shifted by one segment to the left.
23. Modify the program fragment in such a way that symbol would be shifted to the right. Load program into the MC and make sure that it is working properly.
24. Using created program load any 6 symbols word into the display. For this reason restore the program in such a way that it would be the same as it is presented in the 11th point. Enter the appropriate instructions that are needed to indicate 6 symbols word before the instruction *goto$*, like it is done with letter A symbol. In the instruction, which indicates the address of the first sign, change the address to 00.

Comment. Entering the code sequence of the symbols, which must be indicated in the display, it is enough to indicate just first symbol code address. Symbols, which are in 1–8 segments of display, matches addresses 00-07, and in 9–16 segments – addresses 64-71 of the DDRAM memory registers. Therefore, to load 16 symbols into the display it is not enough to indicate just first symbol code address 00. It is necessary before ninth symbol to load instruction, which indicates 64th DDRAM memory address. If DDRAM memory indicator was cleared before the loading of symbol code, symbol code by default is loaded into DDRAM register 00.

25. Load program into the MC and make sure that it operates properly, i.e. the word is indicted in the 1–6 segments of the display.
26. Modify the program fragment, which sends the symbols to LCD display (11th point of this laboratory), as follows: change instruction *goto$* by *goto Start*; enter the label *Start* on the left of the first line; enter instructions that shift symbols by one segment to the right above the instruction *goto Start*. Enter instructions, which call delay loop subroutine *Delay2* (instructions *call Delay2*) above and bellow the strings that

were entered in this point of the laboratory. Load the program into the MC and make sure that it works properly, i.e. the 6 symbols word firstly is indicated in the 1–6 segments and later is shifted to 2–7 segments and this shifting cycle is continually repeated.

27. Change the program, which was created during the execution of point 26 of the laboratory in such a way that after that when the 6 symbols word was shifted to 2–7 segments, it would be shifted to 3–8 and then again to 1–6 segments. Call the subroutine *Delay2* between the every shift to introduce the shift delay. Load program into the MC and make sure that it works. Save the program for laboratory work report.
28. Referring to information given in the point 24 of this laboratory modify the program in such a way that that chosen symbols would be indicated in all 16 segments of display. Load program into MC and make sure that it works.
29. Using delay loop subroutine *Delay2* modify the program created in 28 point in such a way that between the displaying of every symbol the delay would be introduced. Load program into the MC and make sure that it works, i.e. symbols from first till last appear in the segments of the display one by one with defined delay.
30. Modify the program so that chosen words would appear in the display one by one with defined delay. At one time just one word on the display should be seen. Save the program for laboratory work report.

4. Report content

1. The aim of the work.
2. The purpose and parameters of liquid crystal display LCD1601LC.
3. Functions of LCD1601LC display terminals and instructions of display controller.

4. The texts and comments of the programs that were created during the execution of tasks given in points 27 and 30.
5. Conclusions.

5. Control questions

1. How many symbols can be displayed on the LCD1601LC display?
2. How many dots forms one symbol matrix?
3. What is the purpose of display terminals D0–D7?
4. In what memory are stored the data of symbols, which can be displayed?
5. What is the purpose of display terminal R/W?
6. What is the purpose of instruction *goto$*?
7. How many bits of the memory are needed for the storage of one LCD1601LC display symbol?

References

1. Dot matrix liquid crystal display controller/driver HD44780U (LCD-II), Hitachi, http://www.jkmicro.com/documentation/pdf/hd44780u.pdf, 2005, 60 p.
2. LCD graphical module LCD1601LC, http://acesystems.nl/Menu/Electronica/Displays/LCD-displays/LCD1601LC.htm.
3. LCD simulator, http://www.geocities.com/dinceraydin/lcd/ index.html.
4. PIC16F84A data sheet, DS35007B, Microchip Technology Inc., http://www.microchip.com, 2005, 86 p.