S

Sri Indu
College of Engineering & Technology
UGC Autonomous Institution
Recognized under 2(f) & 12(B) of UGC Act 1956,
NAAC, Approved by AICTE &
Permanently Affiliated to JNTUH

Estd.2001

NAAC
NATIONAL ASSESSMENT AND ACCREDITATION COUNCIL

EMBEDDED SYSTEM LAB
(R22CSE3227)

LAB MANUAL

III Year II Semester

DEPARTMENT OF INFORMATION TECHNOLOGY

SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY
B. TECH –INFORMATION TECHNOLOGY

INSTITUTION VISION

To be a premier Institution in Engineering & Technology and Management with competency, values and social consciousness.

INSTITUTION MISSION

IM1 Provide high quality academic programs, training activities and research facilities.

IM2 Promote Continuous Industry-Institute Interaction for Employability, Entrepreneurship, Leadership and Research aptitude among stakeholders.

IM3 Contribute to the Economical and technological development of the region, state and nation.

DEPARTMENT VISION

To be a recognized knowledge centre in the field of Information Technology with self - motivated, employable engineers to society.

DEPARTMENT MISSION

The Department has following Missions:

DM1 To offer high quality student centric education in Information Technology.

DM2 To provide a conducive environment towards innovation and skills.

DM3 To involve in activities that provide social and professional solutions.

DM4 To impart training on emerging technologies namely cloud computing and IOT with involvement of stake holders.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: **Higher Studies:** Graduates with an ability to apply knowledge of Basic sciences and programming skills in their career and higher education.

PEO2: **Lifelong Learning:** Graduates with an ability to adopt new technologies for ever changing IT industry needs through Self-Study, Critical thinking and Problem solving skills.

PEO3: **Professional skills:** Graduates will be ready to work in projects related to complex problems involving multi-disciplinary projects with effective analytical skills.

PEO4: **Engineering Citizenship:** Graduates with an ability to communicate well and exhibitsocial, technical and ethical responsibility in process or product.

PROGRAM OUTCOMES (POs) & PROGRAM SPECIFIC OUTCOMES (PSOs)

PO	Description
PO 1	**Engineering Knowledge:** Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.
PO 2	**Problem Analysis:** Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)
PO 3	**Design/Development of Solutions:** Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)
PO 4	**Conduct Investigations of Complex Problems:** Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).
PO 5	**Engineering Tool Usage:** Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6)
PO 6	**The Engineer and The World:** Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).
PO 7	**Ethics:** Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)
PO 8	**Individual and Collaborative Team work:** Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.
PO 10	**Project Management and Finance:** Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.
PO 11	**Life-Long Learning:** Recognize the need for, and have the preparation and ability for i) independent and life-long learning ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)
Program Specific Outcomes	
PSO 1	**Software Development:** To apply the knowledge of Software Engineering, Data Communication, Web Technology and Operating Systems for building IOT and Cloud Computing applications.
PSO 2	**Industrial Skills Ability:** Design, develop and test software systems for world-wide network of computers toprovide solutions to real world problems.
PSO 3	**Project implementation:** Analyze and recommend the appropriate IT Infrastructure required for the implementationof a project.

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time),those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with thesynopsis / program / experiment details.
3. Student should enter into the laboratory with:
    a) Laboratory observation notes with all the details (Problem statement, Aim, Algorithm,Procedure, Program, Expected Output, etc.,) filled in for the lab session.
    b) Laboratory Record updated up to the last session experiments and other utensils (if any)needed in the lab.
    c) Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation notebook, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain thediscipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out ; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the labafter completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

Course Title	EMBEDDED SYSTEMS LAB	Course Code:	R22CSE4127

Course Outcome No.	Course Outcome Statement
C32L2.1	Identify the functionality of development boards to implement embedded applications.
C32L2.2	Compile bug free assembly or C language programs for microcontrollers to a required task.
C32L2.3	Design an electronic circuit for diverse I/O devices used in real time embedded applications
C32L2.4	Develop a product with all sub systems of functional requirements in optimal hardware and software components.

Course Outcomes (COS)	Program Outcomes (POs)											Program Specific Outcomes (PSOs)		
	PO1	PO2	PO3	PO4	PO5	PO6	PO8	PO9	P10	P11	P12	PSO1	PSO2	PSO3
C32L2.1	-	-	-	3	-	-	-	-	-	-	-	-	-	-
C32L2.2	-	-	-	-	3	-	-	-	-	-	-	-	-	-
C32L2.3	-	-	-	3	2	-	-	-	-	-	-	-	-	-
C32L2.4	-	-	-	3	2	-	-	-	-	-	-	3	-	-
C32L2	-	-	-	3	2.33	-	-	-	-	-	-	3	-	-

SRI INDU COLLEGE OF ENGINEERING & TECHNOLOGY

(An Autonomous Institution under UGC, New Delhi)

B.Tech. - III Year – II Semester

L	T	P	C
0	0	2	1

(R22CSE3227) EMBEDDED SYSTEM LAB

Course Outcomes:

1. Identify the functionality of development boards to implement embedded applications.
2. Compile bug free assembly or C language programs for microcontrollers to a required task.
3. Design an electronic circuit for diverse I/O devices used in real time embedded applications.
4. Develop a product with all sub systems of functional requirements in optimal hardware and software components.

Course Articulation Matrix:

COs/ POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	P10	P11	PSO1	PSO2	PSO3
C22L8 .1	-	-	-	3	-	-	-	-	-	-	-	-	-	-
C22L8 .2	-	-	-	-	3	-	-	-	-	-	-	-	-	-
C22L8 .3	-	-	-	3	2	-	-	-	-	-	-	-	-	-
C22L8 .4	-	-	-	3	2	-	-	-	-	-	-	3	-	-
C22L8	-	-	-	3.00	2.33	-	-	-	-	-	-	3.00	-	-

Laboratory Experiments Mapping with Course Outcomes

S.No.	Experiments to be covered	Mapped CO
1	Introduction to embedded systems lab content.	--
2	Programming using Arithmetic and logical instructions of 8051.	CO 1 & 2
3	Timer programming of 8051.	CO 1 & 2
4	Counter programming of 8051.	CO 1 & 2
5	Serial port programming of 8051.	CO 1 & 2
6	Interfacing of Stepper motor	CO 3 & 4
7	Interfacing of Temperature sensor and Relay control.	CO 3 & 4
8	Using of more complex memory and branch type instructions such as LDMFD/STMFD, Band BL.	CO 1 & 2
9	Basic reg/mem visiting and simple arithmetic/logic computing.	CO 1 & 2
10	Changing ARM state mode by using MRS/MMSR instruction and specify a start address of the text segment by using command line.	CO 1 & 2
11	ARM programming in C language using KEIL IDE	CO 1 & 2
12	Write a random number generation function using assembly language. Call this function from a C program to produce a series of random numbers and save them in the memory	CO 1 & 2
13	Configure and read/write the memory space. Use assembly and C language to read/write words, half-words, bytes, half bytes from/to RAM.	CO 3 & 4
14	Implement the lighting and winking LEDs of the ARM I/O port via programming.	CO 3 & 4
15	ISR (Interrupt Service Routine) programming in ARM based systems with I/O port.	CO 3 & 4

Note:

Minimum of 12 experiments are to be conducted.

1. The Following programs/experiments are to be written for assembler and to be executed the same with 8086 and 8051 kits.
2. ARM programming in C language using KEIL.

CONTENT

1. Introduction to embedded systems lab content.
2. Software Implementation Keil µ Vision 4 for 8051 µc.
3. Programming using Arithmetic and logical instructions of 8051.
4. Timer programming of 8051.
5. Counter programming of 8051.
6. Serial port programming of 8051.
7. Interfacing of Stepper motor
8. Interfacing of Temperature sensor and Relay control.
9. Using of more complex memory and branch type instructions such as LDMFD/STMFD, Band BL.
10. Basic reg/mem visiting and simple arithmetic/logic computing.
11. Introduction to Arm Board (LPC2148)
12. Introduction to Keil µ Vision 4 for LPC2148
13. Changing ARM state mode by using MRS/MMSR instruction and specify a start address of the text segment by using command line.
14. ARM programming in C language using KEIL IDE
15. Write a random number generation function using assembly language. Call this function from a C program to produce a series of random numbers and save them in the memory
16. Configure and read/write the memory space. Use assembly and C language to read/write words, half-words, bytes, half bytes from/to RAM.
17. Implement the lighting and winking LEDs of the ARM I/O port via programming.
18. ISR (Interrupt Service Routine) programming in ARM based systems with I/O port.
19. Additional Programs (Content beyond Syllabus)

List of Experiments

1. Introduction to embedded systems lab content.
2. Programming using Arithmetic and logical instructions of 8051.
3. Timer programming of 8051.
4. Counter programming of 8051.
5. Serial port programming of 8051.
6. Interfacing of Stepper motor
7. Interfacing of Temperature sensor and Relay control.
8. Using of more complex memory and branch type instructions such as LDMFD/STMFD, Band BL.
9. Basic reg/mem visiting and simple arithmetic/logic computing.
10. Changing ARM state mode by using MRS/MMSR instruction and specify a start address of the text segment by using command line.
11. ARM programming in C language using KEIL IDE
12. Write a random number generation function using assembly language. Call this function from a C program to produce a series of random numbers and save them in the memory
13. Configure and read/write the memory space. Use assembly and C language to read/write words, half-words, bytes, half bytes from/to RAM.
14. Implement the lighting and winking LEDs of the ARM I/O port via programming.
15. ISR (Interrupt Service Routine) programming in ARM based systems with I/O port.

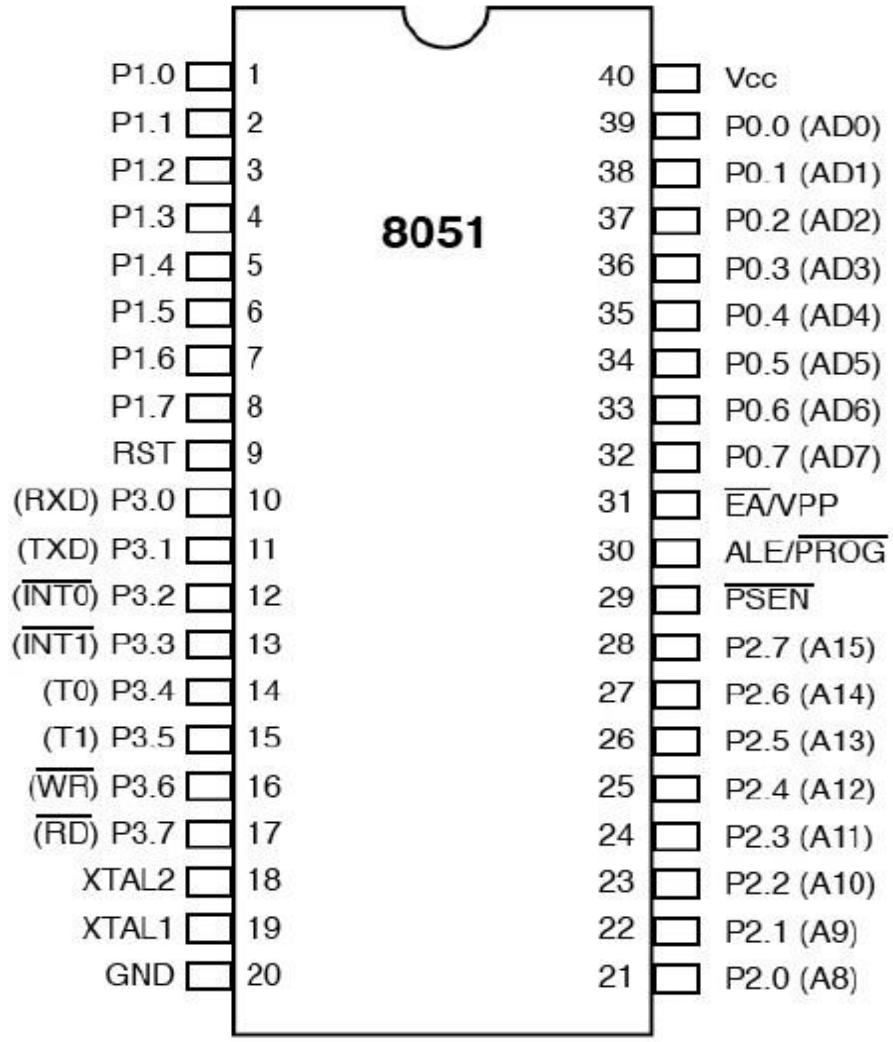
EMBEDDED SYSTEM LAB (R22CSE3227)

EXPERIMENT 1: Introduction to embedded systems lab content.

Introduction to 8051 Microcontroller

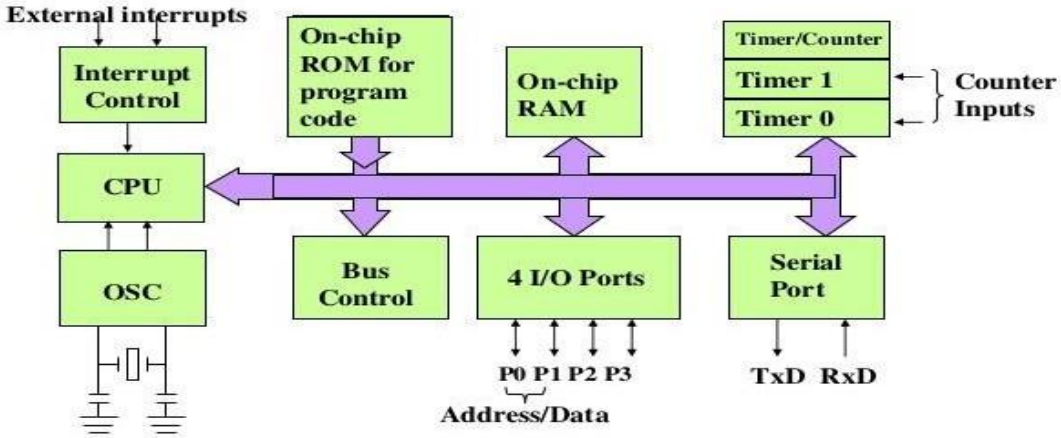
The Intel 8051 is Harvard architecture, single chip microcontroller (µC) which was developed by Intel in 1980 for use in embedded systems. 8051 is an 8-bit micro controller. The Important features of 8051 Architecture:

- 8-bit ALU, Accumulator and Registers;
- 8-bit data bus - It can access 8 bits of data in one operation
- 16-bit address bus - It can access 2^{16} memory locations - 64 kB (65536 locations) each of RAM and ROM
- On-chip RAM - 128 bytes ("Data Memory")
- On-chip ROM - 4 kB ("Program Memory")
- Four-byte bi-directional input/output ports
- UART (serial port)
- Two 16-bit Counter/timers
- Two-level interrupt priority
- Power saving mode
- 8051 have 128-bit addressable memory locations (user defined flags)
- It consists of 16-bit address bus
- It also consists of 3 internal and two external interrupt
- Less power usage in 8051 with respect to other micro-controller
- It consists of 16-bit program counter and data pointer
- 8051 can process 1 million one-cycle instructions per second
- It also consists of 32 general purpose registers each of 8 bits
- ROM on 8051 is 4 Kbytes in size



Pin Diagram of 8051

Block Diagram of 8051



Types of instructions:

Depending on operation they perform, all instructions are divided in several groups:

- Arithmetic Instructions
- Branch Instructions
- Data Transfer Instructions
- Logic Instructions
- Bit-oriented Instructions

Arithmetic instructions:

Arithmetic instructions perform several basic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand. For example:

ADD A, R1 - The result of addition (A+R1) will be stored in the accumulator.

Mnemonic	**Description**
ADD A,Rn	Adds the register to the accumulator
ADD A,direct	Adds the direct byte to the accumulator
ADD A,@Ri	Adds the indirect RAM to the accumulator
ADD A,#data	Adds the immediate data to the accumulator
ADDC A,Rn	Adds the register to the accumulator with a carry flag
ADDC A,direct	Adds the direct byte to the accumulator with a carry flag
ADDC A,@Ri	Adds the indirect RAM to the accumulator with a carry flag
ADDC A,#data	Adds the immediate data to the accumulator with a carry flag
SUBB A,Rn	Subtracts the register from the accumulator with a borrow
SUBB A,direct	Subtracts the direct byte from the accumulator with a borrow
SUBB A,@Ri	Subtracts the indirect RAM from the accumulator with a borrow
SUBB A,#data	Subtracts the immediate data from the accumulator with a borrow
INC A	Increments the accumulator by 1
INC Rn	Increments the register by 1
INC Rx	Increments the direct byte by 1
INC @Ri	Increments the indirect RAM by 1
DEC A	Decrements the accumulator by 1
DEC Rn	Decrements the register by 1
DEC Rx	Decrements the direct byte by 1
DEC @Ri	Decrements the indirect RAM by 1
INC DPTR	Increments the Data Pointer by 1
MUL AB	Multiplies A and B
DIV AB	Divides A by B
DA A	Decimal adjustment of the accumulator according to BCD code

Branch Instructions:

There are two kinds of branch instructions:

Unconditional jump instructions: upon their execution a jump to a new location from where the program continues execution is executed. Conditional jump instructions: a jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.

Mnemonic	Description
ACALL addr11	Absolute subroutine call
LCALL addr16	Long subroutine call
RET	Returns from subroutine
RETI	Returns from interrupt subroutine
AJMP addr11	Absolute jump
LJMP addr16	Long jump
SJMP rel	Short jump (from –128 to +127 locations relative to the following instruction)
JC rel	Jump if carry flag is set. Short jump.
JNC rel	Jump if carry flag is not set. Short jump.
JB bit,rel	Jump if direct bit is set. Short jump.
JBC bit,rel	Jump if direct bit is set and clears bit. Short jump.
JMP @A+DPTR	Jump indirect relative to the DPTR
JZ rel	Jump if the accumulator is zero. Short jump.
JNZ rel	Jump if the accumulator is not zero. Short jump.
CJNE A,direct,rel	Compares direct byte to the accumulator and jumps if not equal. Short jump.
CJNE A,#data,rel	Compares immediate data to the accumulator and jumps if not equal. Short jump.
CJNE Rn,#data,rel	Compares immediate data to the register and jumps if not equal. Short jump.
CJNE @Ri,#data,rel	Compares immediate data to indirect register and jumps if not equal. Short jump.
DJNZ Rn,rel	Decrements register and jumps if not 0. Short jump.
DJNZ Rx,rel	Decrements direct byte and jump if not 0. Short jump.
NOP	No operation

Data Transfer Instructions:

Data transfer instructions move the content of one register to another. The register the content of which is moved remains unchanged. If they have the suffix "X" (MOVX), the data is exchanged with external memory.

Mnemonic	Description
MOV A,Rn	Moves the register to the accumulator
MOV A,direct	Moves the direct byte to the accumulator
MOV A,@Ri	Moves the indirect RAM to the accumulator
MOV A,#data	Moves the immediate data to the accumulator
MOV Rn,A	Moves the accumulator to the register
MOV Rn,direct	Moves the direct byte to the register
MOV Rn,#data	Moves the immediate data to the register
MOV direct,A	Moves the accumulator to the direct byte
MOV direct,Rn	Moves the register to the direct byte
MOV direct,direct	Moves the direct byte to the direct byte
MOV direct,@Ri	Moves the indirect RAM to the direct byte
MOV direct,#data	Moves the immediate data to the direct byte
MOV @Ri,A	Moves the accumulator to the indirect RAM
MOV @Ri,direct	Moves the direct byte to the indirect RAM
MOV @Ri,#data	Moves the immediate data to the indirect RAM
MOV DPTR,#data	Moves a 16-bit data to the data pointer
MOVC A,@A+DPTR	Moves the code byte relative to the DPTR to the accumulator (address=A+DPTR)
MOVC A,@A+PC	Moves the code byte relative to the PC to the accumulator (address=A+PC)
MOVX A,@Ri	Moves the external RAM (8-bit address) to the accumulator
MOVX A,@DPTR	Moves the external RAM (16-bit address) to the accumulator
MOVX @Ri,A	Moves the accumulator to the external RAM (8-bit address)

MOVX @DPTR,A	Moves the accumulator to the external RAM (16-bit address)
PUSH direct	Pushes the direct byte onto the stack
POP direct	Pops the direct byte from the stack/td>
XCH A,Rn	Exchanges the register with the accumulator
XCH A,direct	Exchanges the direct byte with the accumulator
XCH A,@Ri	Exchanges the indirect RAM with the accumulator
XCHD A,@Ri	Exchanges the low-order nibble indirect RAM with the accumulator

Logic Instructions:

Logic instructions perform logic operations upon corresponding bits of two registers. After execution, the result is stored in the first operand.

Mnemonic	**Description**
ANL A,Rn	AND register to accumulator
ANL A,direct	AND direct byte to accumulator
ANL A,@Ri	AND indirect RAM to accumulator
ANL A,#data	AND immediate data to accumulator
ANL direct,A	AND accumulator to direct byte
ANL direct,#data	AND immediate data to direct register
ORL A,Rn	OR register to accumulator
ORL A,direct	OR direct byte to accumulator
ORL A,@Ri	OR indirect RAM to accumulator
ORL direct,A	OR accumulator to direct byte
ORL direct,#data	OR immediate data to direct byte
XRL A,Rn	Exclusive OR register to accumulator
XRL A,direct	Exclusive OR direct byte to accumulator
XRL A,@Ri	Exclusive OR indirect RAM to accumulator
XRL A,#data	Exclusive OR immediate data to accumulator
XRL direct,A	Exclusive OR accumulator to direct byte

XORL direct,#data	Exclusive OR immediate data to direct byte
CLR A	Clears the accumulator
CPL A	Complements the accumulator (1=0, 0=1)
SWAP A	Swaps nibbles within the accumulator
RL A	Rotates bits in the accumulator left
RLC A	Rotates bits in the accumulator left through carry
RR A	Rotates bits in the accumulator right
RRC A	Rotates bits in the accumulator right through carry

Bit-oriented Instructions

Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed upon single bits.

Mnemonic	**Description**
CLR C	Clears the carry flag
CLR bit	Clears the direct bit
SETB C	Sets the carry flag
SETB bit	Sets the direct bit
CPL C	Complements the carry flag
CPL bit	Complements the direct bit
ANL C,bit	AND direct bit to the carry flag
ANL C,/bit	AND complements of direct bit to the carry flag
ORL C,bit	OR direct bit to the carry flag
ORL C,/bit	OR complements of direct bit to the carry flag
MOV C,bit	Moves the direct bit to the carry flag
MOV bit,C	Moves the carry flag to the direct bit

Description of all 8051 instructions:

Here is a list of the operands and their meanings:

- **A** - accumulator; **Rn** - is one of working registers (R0-R7) in the currently active RAM memory bank;

- **Direct** - is any 8-bit address register of RAM. It can be any general-purpose register or a SFR (I/O port, control register etc.);
- **@Ri** - is indirect internal or external RAM location addressed by register R0 or R1;
- **#data** - is an 8-bit constant included in instruction (0-255);
- **#data16** - is a 16-bit constant included as bytes 2 and 3 in instruction (0-65535);
- **addr16** - is a 16-bit address. May be anywhere within 64KB of program memory;
- **addr11** - is an 11-bit address. May be within the same 2KB page of program memory as the first byte of the following instruction;
- **Rel** - is the address of a close memory location (from -128 to +127 relative to the first byte of the following instruction). On the basis of it, assembler computes the value to add or subtract from the number currently stored in the program counter;
- **bit** - is any bit-addressable I/O pin, control or status bit; and
- **C** - is carry flag of the status register (register PSW).

8051 Addressing modes:

The way of specifying the address of the operand is called as addressing mode. The 8051 microcontroller is having four addressing modes for accessing data.

1. Immediate Addressing mode
2. Register Addressing mode
3. Direct Addressing mode
4. Indirect Addressing mode
5. Indexed Addressing mode

1. Immediate addressing mode:

The operand comes immediately after the op-code. The immediate data must be preceded by the hash sign, "#".

```
MOV A,#25H          ;load 25H into A
MOV R4,#62          ;load the decimal value 62 into R4
MOV B,#40H          ;load 40H into B
MOV DPTR,#4521H     ;DPTR=4512H
```

2. Register Addressing mode:

Register addressing mode involves the use of registers to hold the data to be manipulated. In this addressing mode register which is having the data is part of the instruction.

```
MOV A,R0  ;copy the contents of R0 into A
MOV R2,A  ;copy the contents of A into R2
ADD A,R5  ;add the contents of R5 to contents of A
ADD A,R7  ;add the contents of R7 to contents of A
MOV R6,A  ;save accumulator in R6
```

3. Direct Addressing mode:

In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction.

```
MOV R0,40H    ;save content of RAM location 40H in R0
MOV 56H,A     ;save content of A in RAM location 56H
MOV R4,7FH    ;move contents of RAM location 7FH to R4
```

4. Indirect Addressing mode:

A register is used as a pointer to the data. If the data is inside the CPU, only registers R0 and R 1 are used for this purpose. R2 - R7 cannot be used to hold the address of an operand located in RAM when using indirect addressing mode. When RO and R1 are used as pointers they must be preceded by the @ sign.

```
MOV A,@R0 ;move contents of RAM location whose
          ;address is held by R0 into A
MOV @R1,B ;move contents of B into RAM location
          ;whose address is held by R1
```

5. Indexed Addressing mode

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space of the 8051.

- The instruction used for this purpose is :

MOVC A, @ A+DPTR

- The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM.

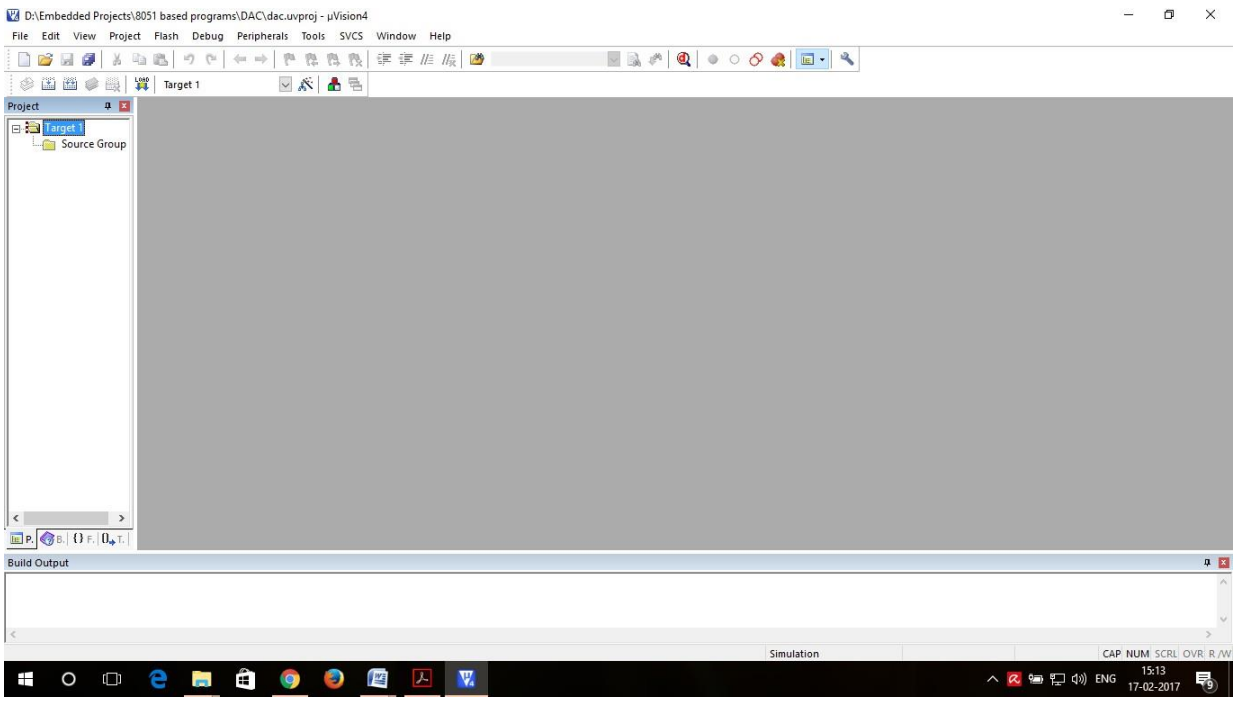
- Because the data elements are stored in the program (code) space ROM of the 8051, the instruction MOVC is used instead of MOV. The "C" means code.

In this instruction the contents of A are added to the 16-bit register DPTR to form the 16bit address of the needed data.

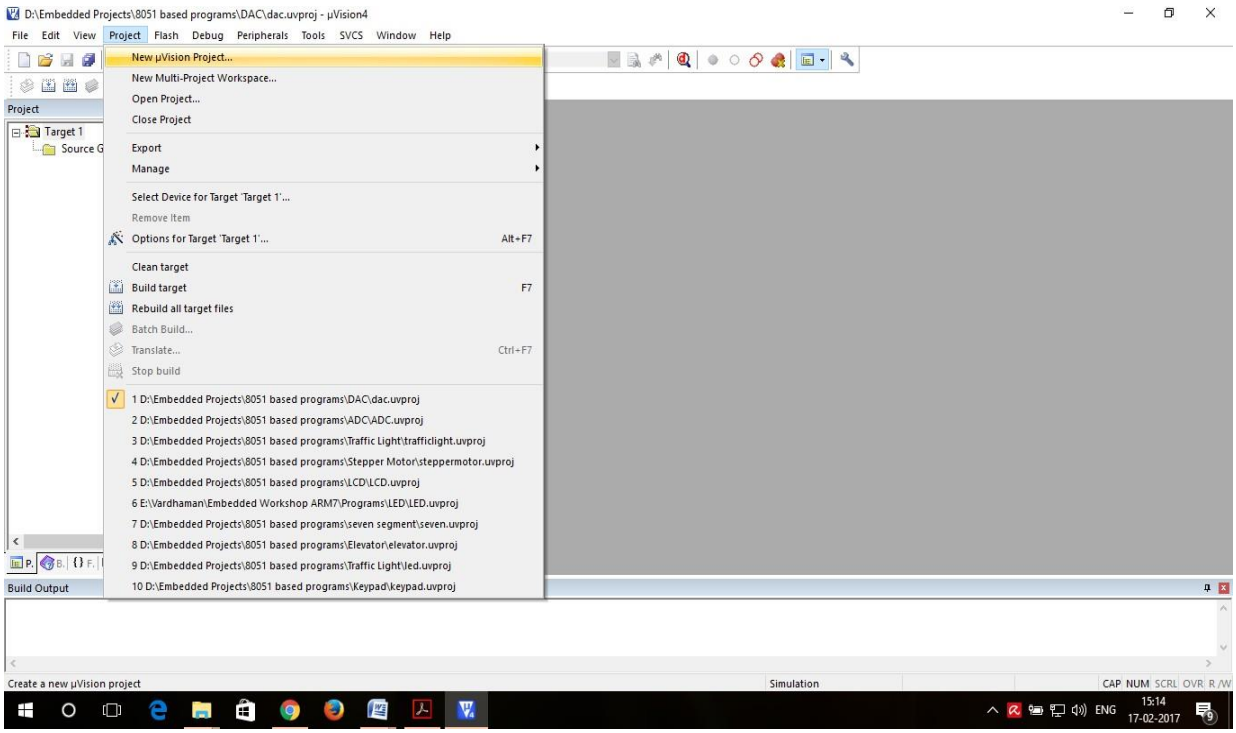
SOFTWARE IMPLEMENTATION KEIL µ VISION 4 FOR 8051 µC

SOFTWARE IMPLEMENTATION KEIL µ VISION 4 FOR 8051 µC

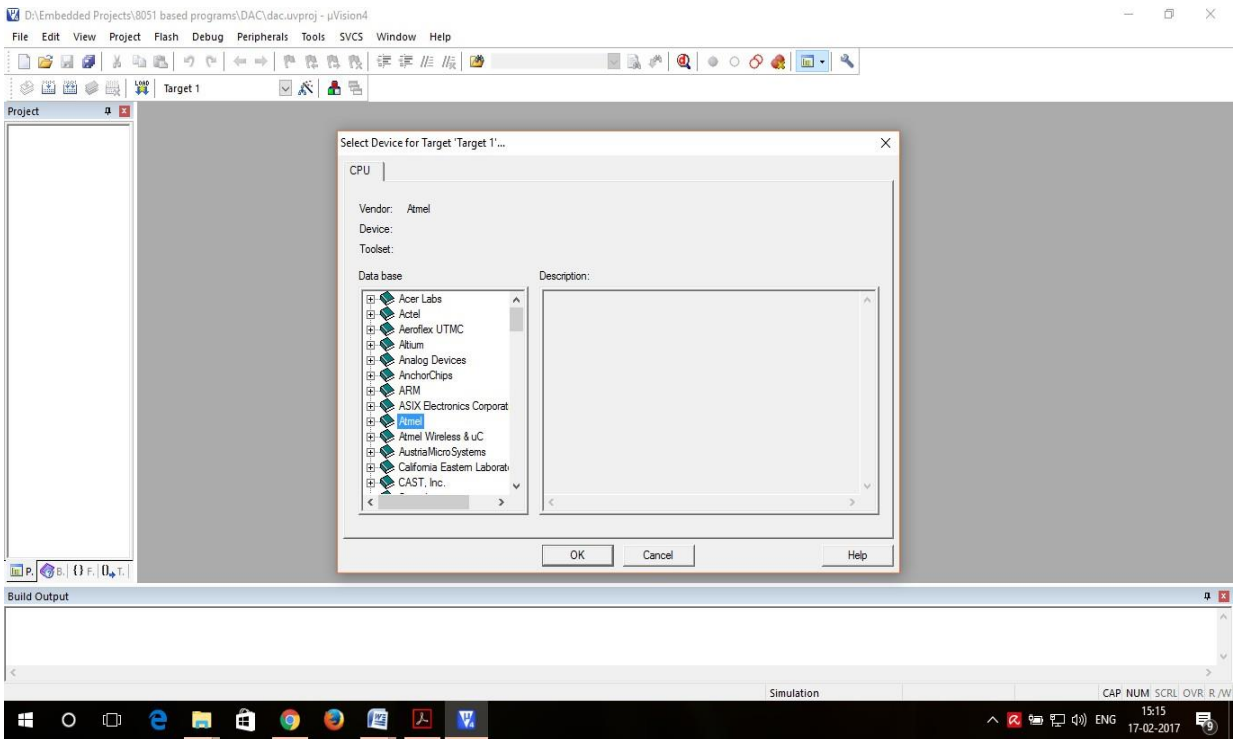
Step 1: Give a double click on µvision4 icon on the desktop; it will generate a window as shown below:



Step 2: To create new project, go to project, select new µvision project.



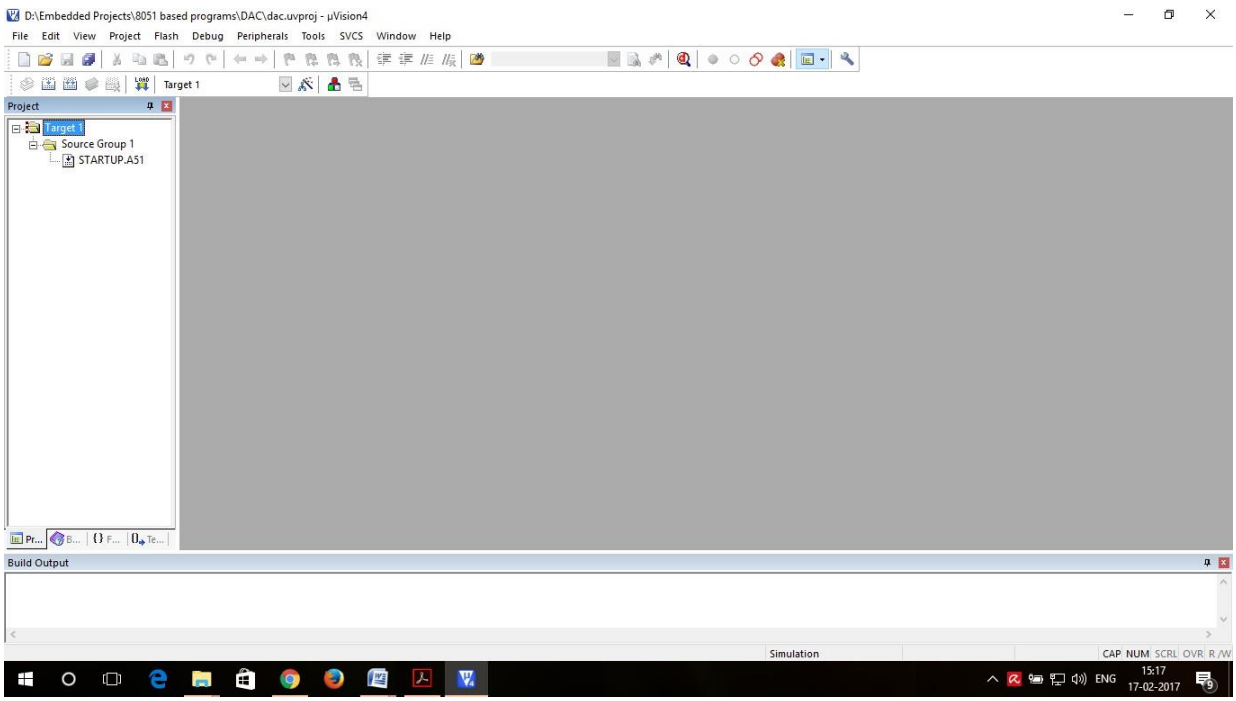
Step 3: Select a drive where you would like to create your project.
Step 4: Create a new folder and name it with your project name.
Step 5: Open that project folder and give a name of your project executable file and save it.
Step 6: After saving, it will show some window there select your microcontroller company i.e. Atmel.



Step 7: Select your chip as AT89C51ED2

Step 8: After selecting chip click on OK then it will display some window asking to add STARTUP file. Select YES.

Step 9: A target is created and start up file is added to your project window and is shown below.



Step 10: To write your project code select a new file from FILE menu bar.

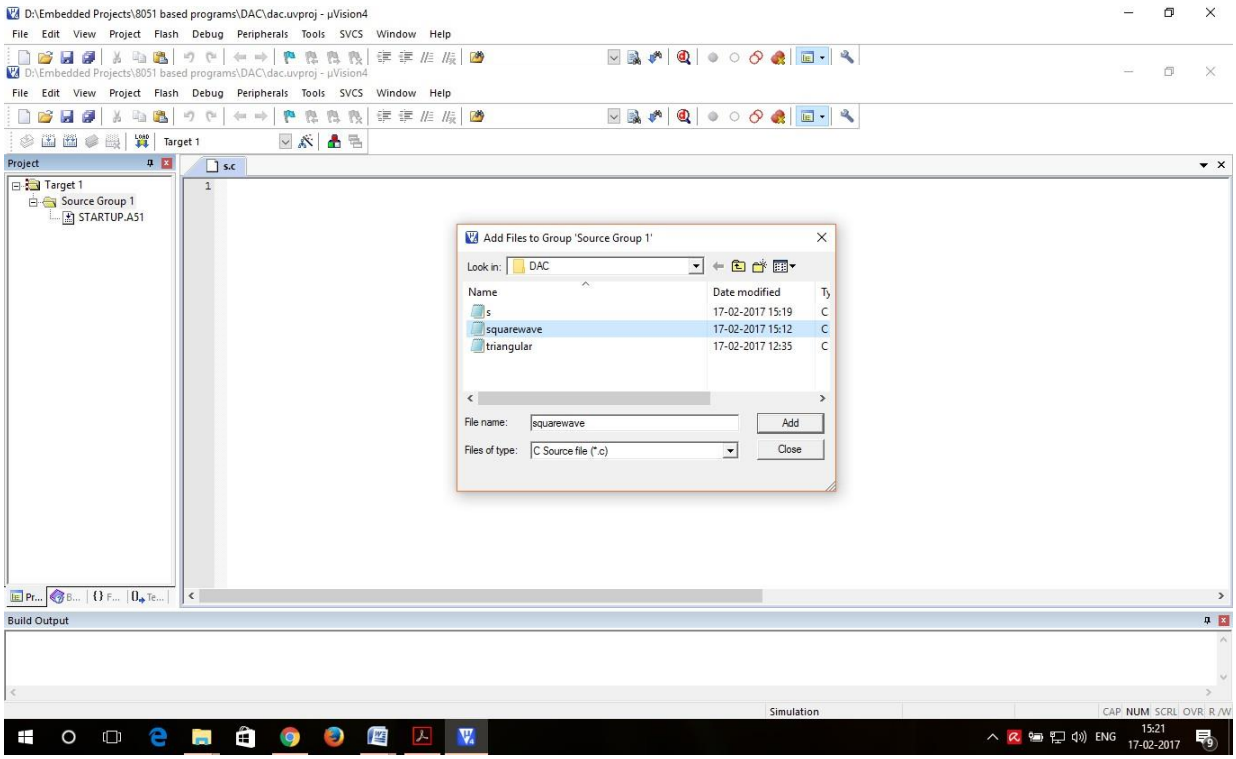
Step 11: It will display some text editor, to save that file select SAVE option from FILE menu bar.

Step 12: Save file name with .c/.asm extension depending on the language using to write the programs.

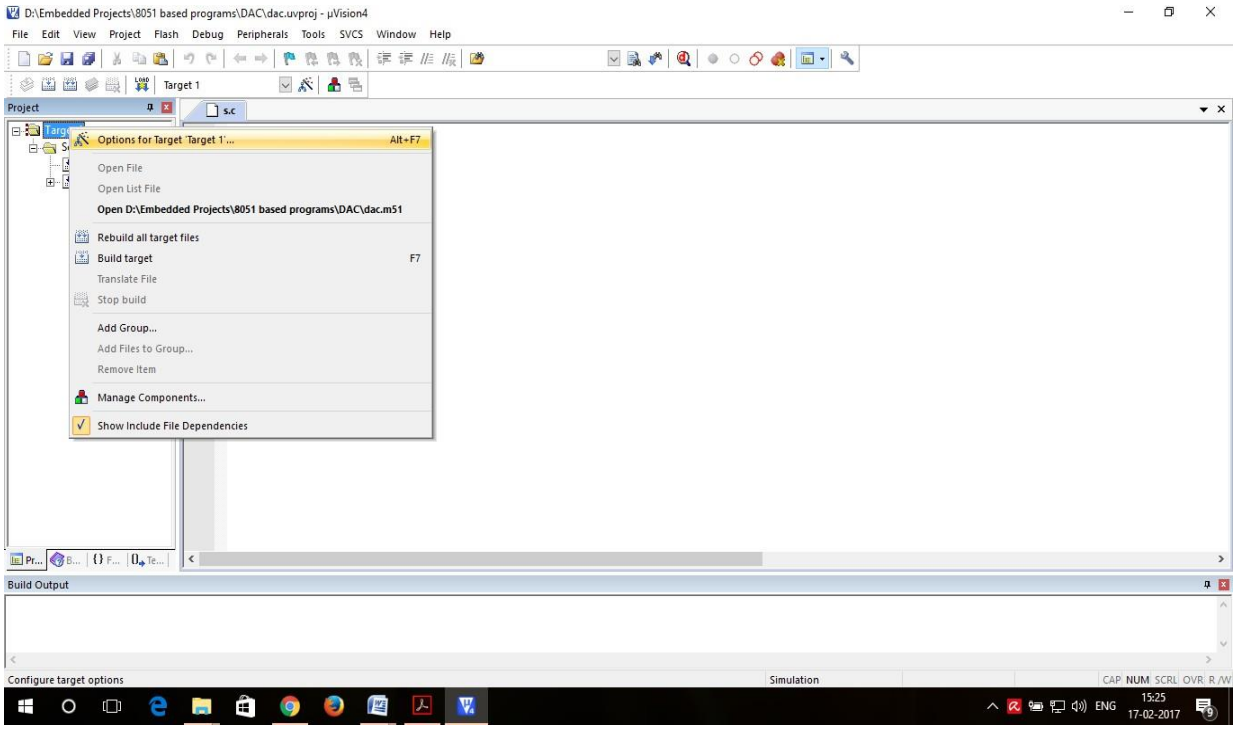
Step 13: Write the code of your project and save it.

Step 14: To add c/asm file to target, give a right click on Source Group, choose "ADD files to Group" option.

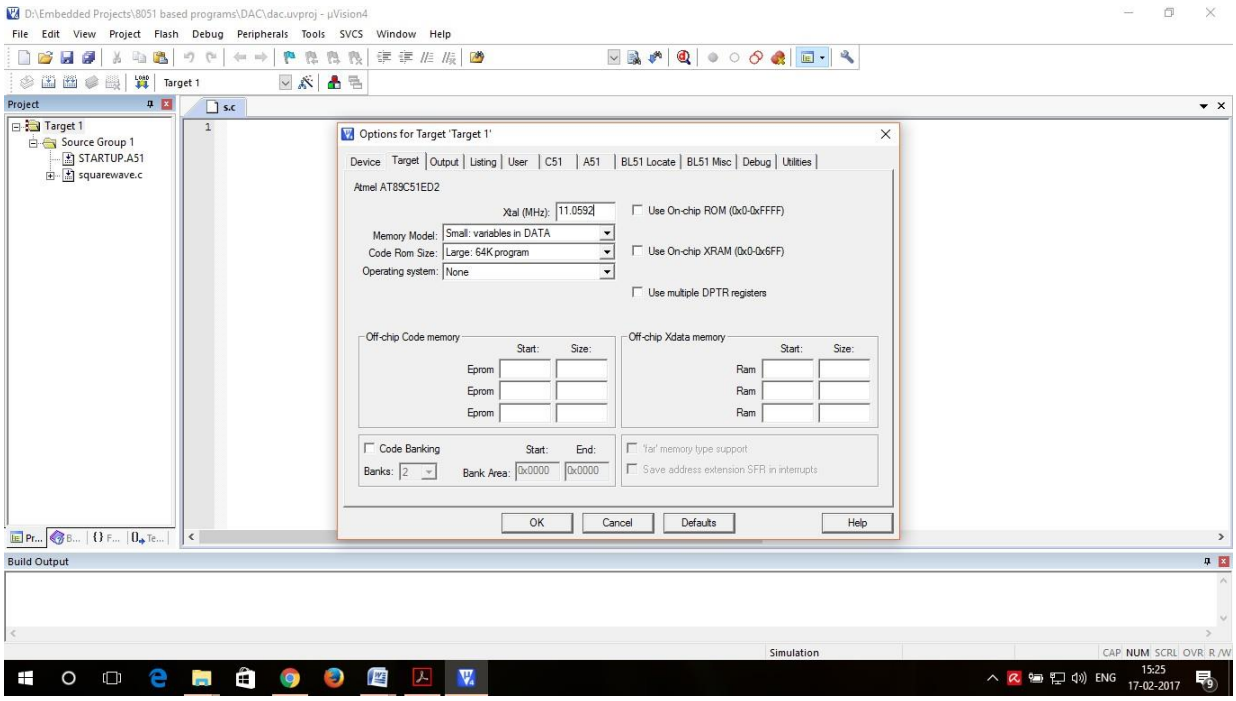
Step 15: It will displays some window there select the file you have to add and click on ADD option.



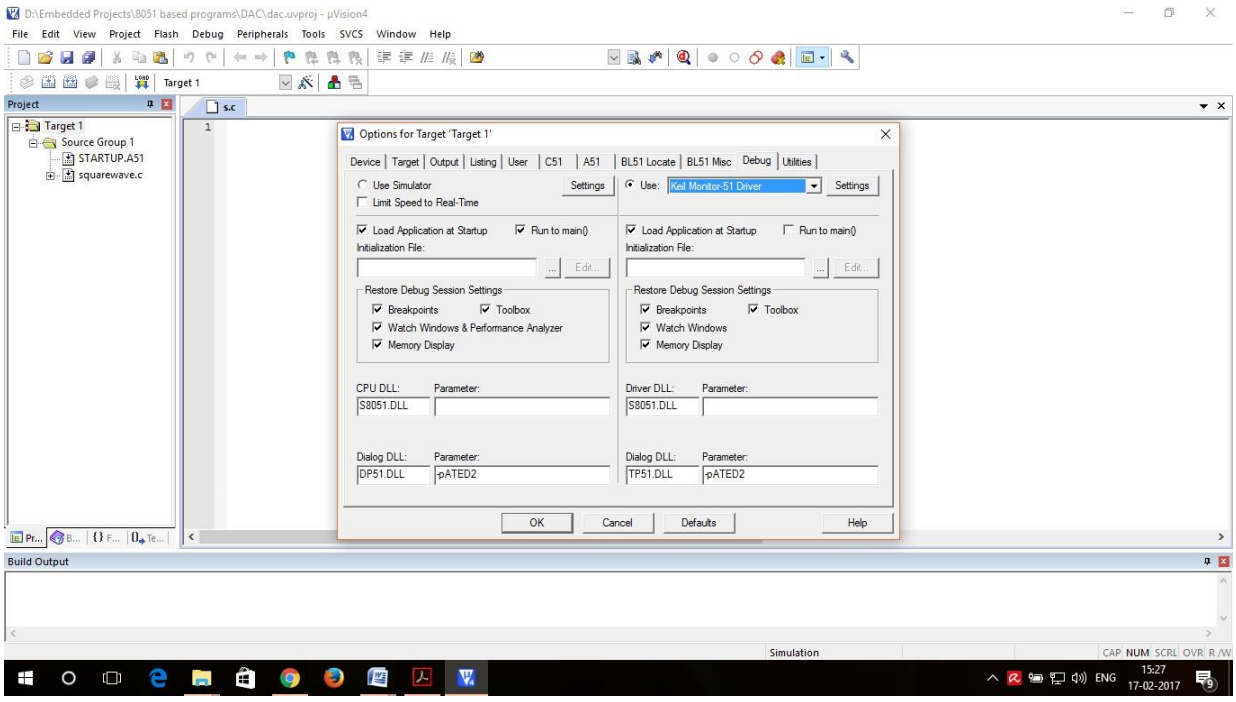
Step 16: The file will be added to target and it is shown in the project window.
Step 17: Now give a right click on target in the project window and select "Options for Target".



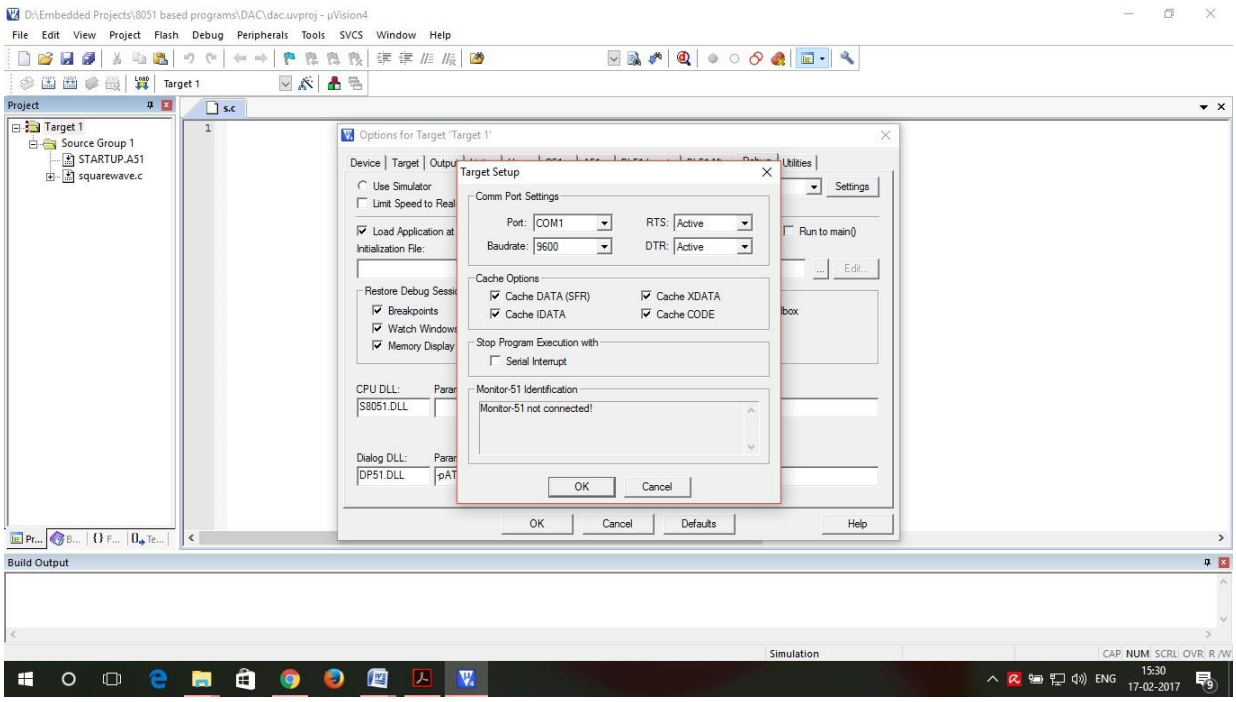
Step 18: It will show some window like below, in that go to target option and type crystal oscillator frequency Xtal (MHz) as 11.0592.



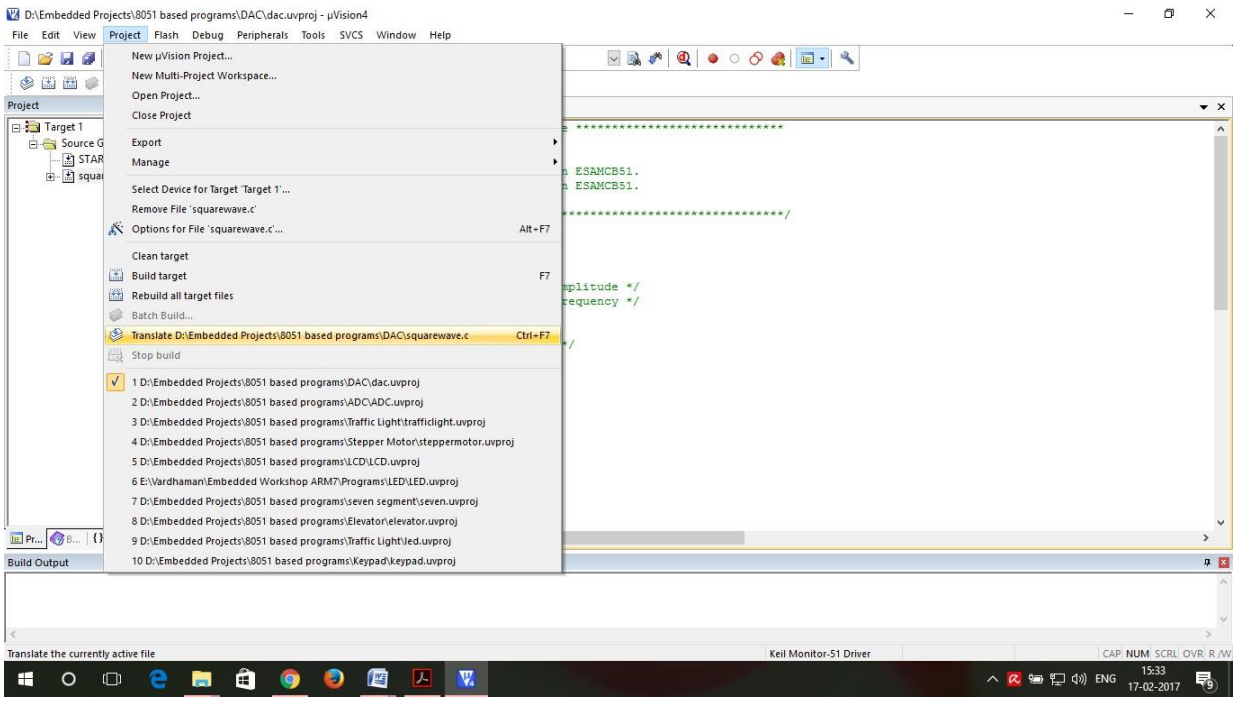
Step 19: In the same window go to debug option and choose use and select Keil Monitor-51 Driver form drop down.



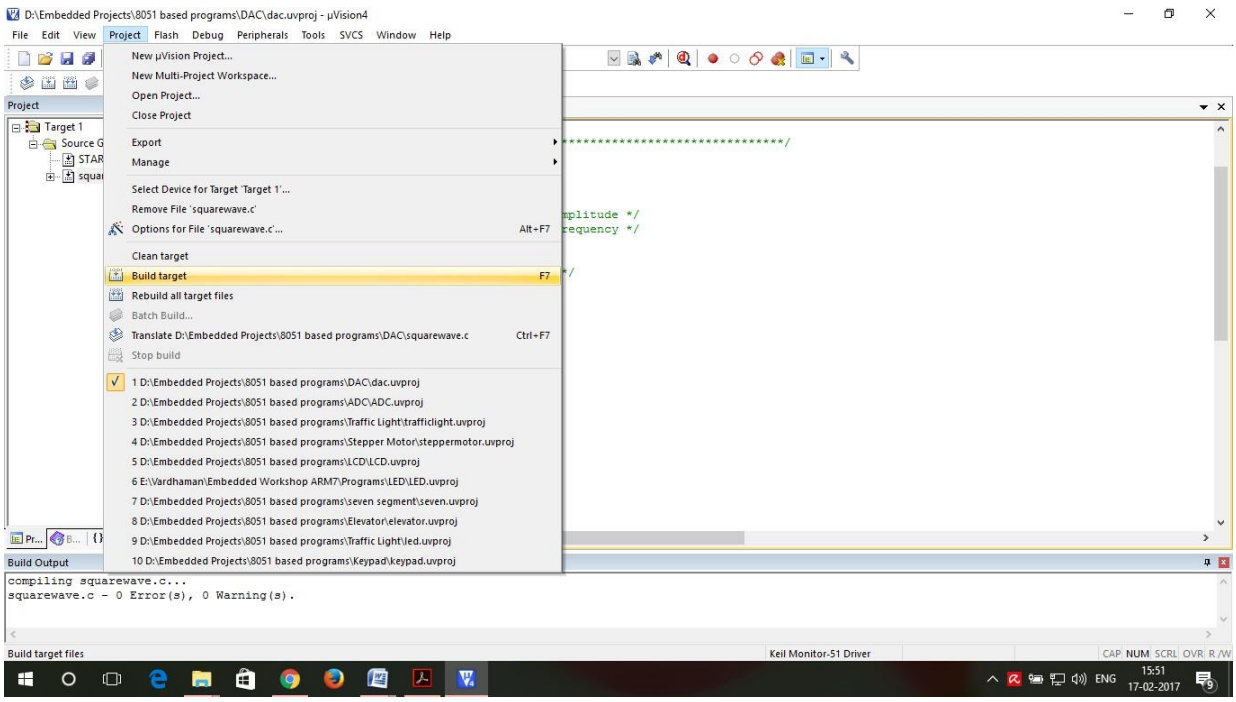
Step 20: In the same window click the settings and it opens another window like below in that select the COM port to which the MCB-51 board is connected and click on OK.



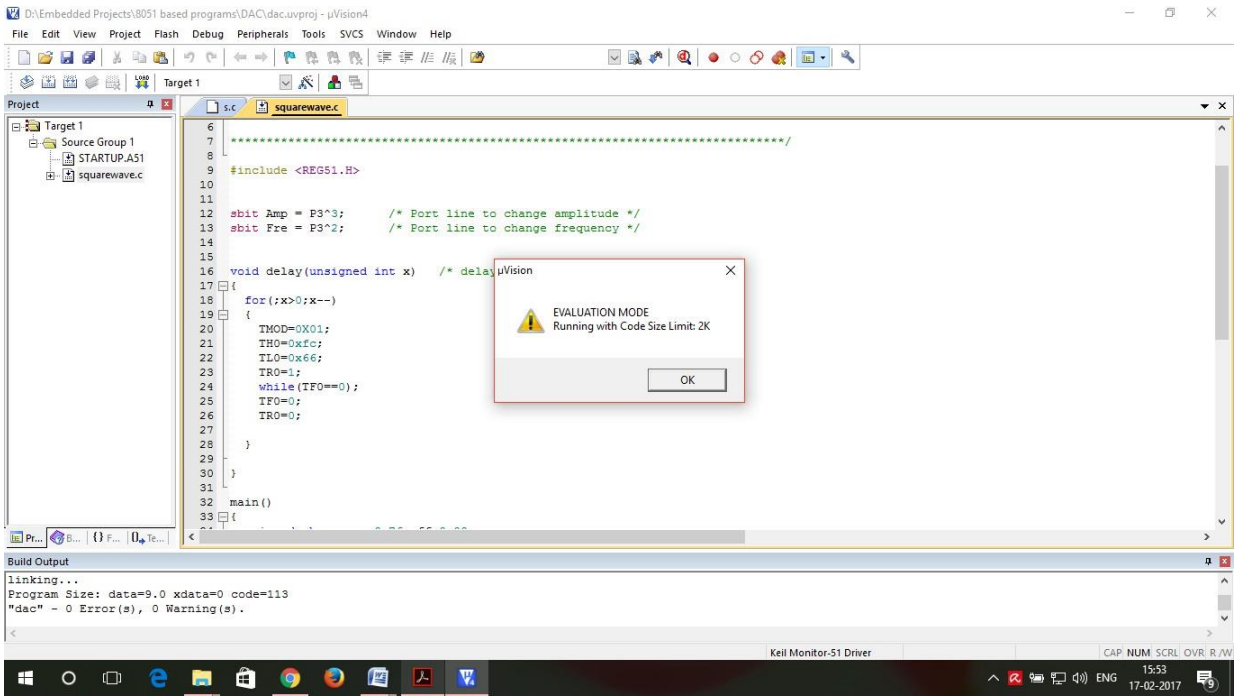
Step 20: It will come back to the previous window and click OK in that window.
Step 21: Now go to project and click on Translate as shown in below figure it shows the errors and warnings in Build output.



Step 22: Clear all the syntax errors and make sure 0 errors and 0 warnings.
Step 23: Now go to project and click on Build as shown in below figure it shows the errors and warnings in Build output and shows target file is created.



Step 24: Go to debug tab and click on start/stop debug it opens the below window and click OK.



Step 25: It opens the following window and click run in the debug tab and the observe the output in interfacing board

EXPERIMENT 2: Programming using Arithmetic and Logical instructions of 8051

Aim: To perform arithmetic and logical operations using 8051 microcontroller.

Apparatus: PC with Keil µvision

Program:

S.No	Label	Instruction	Comment
1		Mov a, #59h	Addition of two numbers is done and result stored in r1 register
2		Mov b, #38h	
3		Add a, b	
4		Mov r1, a	
5		Mov a, #59h	Subtraction of two numbers is done and result stored in r2 register
6		Mov b, #38h	
7		subb a, b	
8		Mov r2, a	
9		Mov a, #09h	Two numbers are multiplied and the result is stored in r3
10		Mov b, #02h	
11		mul ab	
12		Mov r3, a	
13		Mov a, #09h	Division of two numbers is done and results stored in r4, r5 register
14		Mov b, #02h	
15		div ab	
16		Mov r4, a	
17		Mov r5, b	
18		Mov a, #29h	Calculating the ASCII value of lower nibble in ‘a’and stored in r6
19		Mov r0, a	
20		Anl a, #0fh	
21		Orl a, #30h	
22		Mov r6, a	
23		Mov a, r0	Calculating the ASCII value of upper nibble in ‘a’and stored in r7
24		Anl a, #0f0h	
25		Swap a	
26		Orl a, #30h	
27		Mov r7,a	

Result:

Input	Output
A=59h	R1= 91
B=38h	
A=59h	R2=21
B=38h	
A=09h	R3=12
B=02h	
A=09h	R4= 02, R5= 01
B=02h	
A=29h	R6=39, R7=32

Viva questions:

1. Which are the general purposes registers in 8051?
2. What is operating frequency of 8051?
3. What is special function register? Explain its importance.
4. What are the addresses of RAM, reserved for bit addressable?
5. What is the function of 01h of Int 21h?

EXPERIMENT 3: Timer Programming of 8051

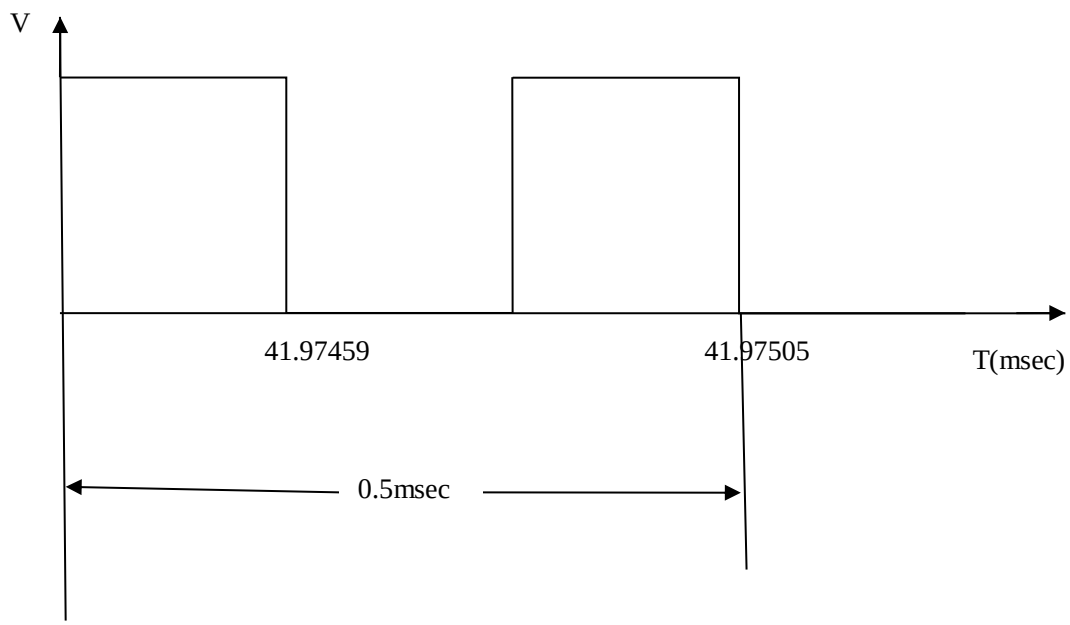
Aim: To verify timer operation in 8051 microcontroller

Apparatus: PC with Keil µvision

Program:

S.No	Label	Instruction	Comment
1		mov tmod, #01h	Initialize the mode of timer(timer0 ,mode1)
2	l1:	mov tl0, #1ah	Initializing timer 0 register
3		mov th0, #0ffh	
4		setb tr0	
5	l2:	jnb tf0, l2	If tf0 =0 ,jump to l2; else continues
6		cpl p1.5	Complement the value at port 1.5
7		clr tf0	Clear timer0 overflow
8		sjmp l1	Short jump to l1

Result:



Viva questions:

1. What is the significance of Timer Mode 2?
2. How many timers are there in 8051?
3. What is the register format of Tmod?
4. What is the register format of Tcon?
5. What is jnb?

EXPERIMENT 4: Counter Programming of 8051

Aim: To verify Counter operation in 8051 microcontroller

Apparatus: PC with Keil µvision

Program:

S.No	Label	Instruction	Comment
1		mov sp, #07h	Initialize stack pointer since we are using subroutine program
2		mov r2, #00h	Initialize the counter with zero
3	back:	mov p1, #00h	Send 00H on port 1 to generate low level of square wave
4		acall delay	Wait for sometime
5		Mov p1,#0ffh	Send FFH on port 1 high level of square wave
6		acall delay	Send FFH on port 1 high level of square wave
7		inc r2	Increment the counter after one clock cycle
8		sjmp back	Repeat the sequence
	delay:	mov r1,#0ffh	Load Count for delay
	again:	djnz r1,again	Decrement count and repeat the process until count is zero
		ret	Return to main page
		end	End the program

Result:

r2 =

Viva questions:

1. What is the significance of Timer Mode 2?
2. How many timers are there in 8051?
3. What is the register format of Tmod?
4. What is the register format of Tcon?
5. What is jnb?

EXPERIMENT 5: Serial Port Programming of 8051. (UART Operations)

Aim: To perform UART operation in 8051 microcontroller using polling and interrupt modes

Apparatus: PC with Keil µvision

Program:

a) **Polling mode:**

S.No	Label	Instruction	Comment
1		org 0000h	Originate at 0000h
2		mov tmod, #20h	Initialize the mode of timer(timer1 ,mode2)
3		mov th1, #0fdh	Initializing timer 1 register
4		mov scon, #40h	Initializing the scon and pcon registers for serial communication (mode 2)
5		anl pcon, #7fh	
6		setb tr1	Enable timer1
7	l1:	clr ti	Reset TI
8		mov sbuf, #'a'	Transmit character ‘a’
9	l2:	jnb ti,l2	If ti =0 ,jump to l2; else continues
10		sjmp l1	Short jump to l1

Result:

The character ‘a’ is transmitted continuously in polling mode.

Viva questions:

1. What is Polling?
2. What is importance of TI flag?
3. In how many modes serial port is operated in 8051?
4. Explain SCON register.
5. What is the difference between microprocessor and microcontroller?

b) Interrupt mode:

S.No	Label	Instruction	Comment
1		org 0000h	Originate at 0000h
2		ljmp main	Long Jump to main
3		org 0023h	Originate at 0023h
4		ljmp isr	Long jump to isr
5		org 0100h	Originate at 0100h
6	main:	MOV sp,#68h	Stack pointer is initialized
7		MOV Tmod,#20h	Initialize the mode of timer(timer0 ,mode1)
8		anl PCON,#7Fh	Initializing the pcon register for serial communication (mode 2)
9		MOV Th1,#0FDh	Baud rate is initialized to 9600
10		setb TR1	Enable timer1
11		MOV scon,#40h	Initializing the scon register for serial communication (mode 2)
12		orl IE,#90h	Enable the serial communication interrupt
13		MOV sbuf,#'a'	Transmit character ‘a’
14	l1:	sjmp l1	Short jump to l1
15	ISR:	clr TI	Clear ti
16		MOV sbuf,#'a'	Transmit character ‘a’
17		RETI	Return from interrupt service routine

Result:

The character ‘a’ is transmitted continuously in interrupt mode.

Viva questions:

1. What is baud rate?
2. How do we know when data has been received?
3. How do we know when the complete data byte has been sent?
4. Explain SCON register.
5. What is Harvard architecture?

EXPERIMENT 6: Interfacing of Stepper motor

AIM: To interface stepper motor with 8051 parallel port and to vary speed of motor and direction of motor.

APPARATUS REQUIRED:

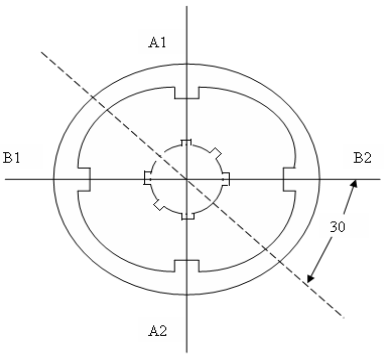
- 8051 trainer kit
- Stepper motor interface board

THEORY:

A motor in which the rotor is able to assume only discrete stationary angular position is a stepper motor. The motor motion occurs in a stepwise manner from one equilibrium position to next.

The motor under our consideration uses 2-phase scheme of operation .in this scheme, any two adjacent stator windings are energized. The switching condition for the above said scheme is shown in table.

STEPPER MOTOR



In order to vary the speed of the motor, the values stored in the registers r1, r2, r3 can be changed appropriately.

ALGORITHM:

1. Store the look up table address in DPTR.
2. Move the content value (04) to one of the register(R0).
3. Load the control word for motor rotation in accumulator.
4. Push the address in DPTR into stack.
5. Load FFC0 in to DPTR.
6. Call the delay program.
7. Send the control word for motor rotation to the external device.
8. POP up the values in stack and increment it.
9. Decrement the count in R0.if zero go to next step else proceed the step 3
10. Perform steps 1 to 9 repeatedly.

PROGRAM:

Address	Label	Mnemonics	Comments
4100	START	MOV DPTR,#4500	LOAD ADDRESS OF FIRST DATA IN DPTR
4103		MOV R0, #04	SAVE THE VALUE 04 IN R0
4105	AGAIN	MOVX A, @DPTR	LOAD THE DATA POINTER ADDRESS IN ACCUMULATOR
4106		PUSH DPH	PUSH DPH VALUE TO STACK
4107		PUSH DPL	PUSH PLD VALUE TO STACK
4108		MOV DPTR, #FFC0H	MOVE THE VALUE TO DATA POINTER
410B		MOV R2, #04H	MOVE VALUE 04 TO R2
410D	DLY 2	MOV R1, #04H	MOVE VALUE FF TO R1
410F	DLY 1	MOV R3, #01H	MOVE VALUE FF TO R3
4111	DLY	DJNZ R3, DLY	DECREMENTS THE BYTE
4113		DJNZ R1, DLY 1	DECREMENTS THE BYTE
4115		DJNZ R2, DLY 2	DECREMENTS THE BYTE
4117		MOVX @DPTR, A	MOVE THE ACCUMULATOR VALUE TO DATA POINTER
4118		POP DPL	POP DPL VALUE FROM STACK TO ACCUMULATOR
4119		POP DPH	POP DPH VALUE FROM STACK TO ACCUMULATOR
411A		INC DPTR	INCREMENT DPTR
411B		DJNZ R0, AGAIN	DECREMENTS THE BYTE, IF RESULTS ZERO BRANCHES TO AGAIN LOOP
411E		SJMP START	TRANSFER TO START ADDRESS

DATA:

4500: 09, 05, 06, 0A

RESULT:

Thus the speed and direction of motor controlled using 8051 trainer kit.

Viva questions:

1. To initialize any port as an output port what value is to be given to it?
2. Which addressing mode is used in pushing or popping any element on or from the stack?
3. Which pins of a microcontroller are directly connected with 8255?
4. What is the value of the control register when RESET button is set to zero?
5. How can we control the speed of a stepper motor?

EXPERIMENT 7: Interfacing of Temperature Sensor Relay control

AIM:

To interface temperature sensor and relay control with 8051 to reads the temperature from the LM35 sensor (connected to an ADC0804), converts the digital output, and controls a relay based on a threshold temperature.

APPARATUS REQUIRED:

- 8051 trainer kit
- Temperature sensor and Relay control board

Hardware Connections

- **LM35** → Outputs an analog voltage proportional to temperature (10mV per °C).
- **ADC0804** → Converts the analog voltage from LM35 into an 8-bit digital output.
- **8051 Microcontroller** → Reads the digital temperature from ADC0804 via Port 1.
- **Relay** → Connected to Port 2, activated based on temperature threshold.

PROGRAM:

```
ORG 0000H

; Define ports

TEMP_SENSOR EQU P1      ; ADC0804 digital output connected to Port 1

RELAY_CTRL EQU P2.0     ; Relay control connected to P2.0

THRESHOLD EQU 50        ; Temperature threshold (in °C)


MOV P1, #0FFH           ; Set Port 1 as input (ADC output)

MOV P2, #00H            ; Clear Port 2 (Relay OFF initially)


MAIN:

   MOV A, TEMP_SENSOR; Read ADC output (8-bit digital temperature)

   MOV B, #2          ; Multiply by 2 (since 1 unit = 2°C for 8-bit ADC)

   MUL AB             ; A = A * B (temperature in °C)


   MOV R0, A          ; Store temperature in R0

   MOV A, R0          ; Compare with threshold

   CJNE A, #THRESHOLD, CHECK_TEMP

   SJMP RELAY_ON      ; If temperature == threshold, turn ON relay
```

```
CHECK_TEMP:
   JB ACC.7, RELAY_OFF; If temperature < threshold, turn OFF relay
   SJMP RELAY_ON       ; Else, turn ON relay

RELAY_ON:
   SETB RELAY_CTRL     ; Turn ON relay
   SJMP MAIN           ; Repeat loop

RELAY_OFF:
   CLR RELAY_CTRL      ; Turn OFF relay
   SJMP MAIN           ; Repeat loop

END
```

Working of the Code

1. **Reads Temperature Data** from ADC0804 (8-bit digital output).
2. **Converts Digital Value** to °C using multiplication (assuming 2°C per bit).
3. **Compares with Threshold (50°C)**:
    - If **Temperature >= 50°C**, the relay **turns ON**.
    - If **Temperature < 50°C**, the relay **turns OFF**.
4. **Loop Continuously** to monitor temperature changes.

Assumptions

- ADC0804 is properly interfaced with LM35 and 8051.
- The ADC0804 outputs an 8-bit digital value (0-255).
- Relay is active-high (1 = ON, 0 = OFF).

Viva questions:

1. What steps have to be followed for interfacing a sensor to a microcontroller 8051?

2. Why Vref is set of ADC0848 to 2.56 V if analog input is connected to the LM35?

3. What is signal conditioning?

4. What is the difference between LM 34 and LM 35 sensors?

5. What is the temperature range of LM35?

EXPERIMENT 8: Using of more complex memory and branch type instructions such as LDMFD/STMFD, B AND BL

Aim: This program:

- **Uses LDMFD/STMFD** for stack operations.
- **Uses B and BL** for branching and function calls.
- **Reads temperature from an ADC (Analog-to-Digital Converter).**
- **Controls a relay** if the temperature exceeds **50°C**.

Program code:

```
.syntax unified
.cpu cortex-m4
.text
.global main

main:
    STMFD sp!, {lr}          @ Save Link Register (LR) on the stack
    BL read_temperature      @ Call function to read temperature
    CMP r0, #50              @ Compare temperature with threshold (50°C)
    BGE relay_on             @ If temperature >= 50°C, branch to relay_on
    B relay_off              @ Else, branch to relay_off

relay_on:
    MOV r1, #1               @ Set relay ON (assuming GPIO high = ON)
    STR r1, [r2]             @ Store value to relay control register
    B main                   @ Repeat process

relay_off:
    MOV r1, #0               @ Set relay OFF
    STR r1, [r2]             @ Store value to relay control register
    B main                   @ Repeat process
```

```
read_temperature:

    STMFD sp!, {lr}          @ Push LR to stack
    MOV r0, #45              @ Simulated temperature value (45°C for testing)
    LDMFD sp!, {lr}          @ Restore Link Register
    BX lr                    @ Return to main

    .end
```

Viva questions:

1. What is the purpose of the BL (Branch with Link) instruction?

2. What happens if the LR is modified before returning?

3. What does STMFD SP!, {R4-R7, LR} do?

4. What is the difference between STMFD and LDMFD?

5. What is the role of the ! in STMFD and LDMFD instructions?

EXPERIMENT 9: Basic reg/mem visiting and simple arithmetic/logic computing

Procedure-1:

ARM Assembly: Basic Register/Memory Access and Simple Arithmetic/Logic Operations

Aim: This ARM Assembly program demonstrates:

- **Register and Memory Access** (LDR, STR, MOV, ADR)
- **Basic Arithmetic Operations** (ADD, SUB, MUL, SDIV)
- **Logical Operations** (AND, ORR, EOR, MVN)

ARM Assembly Code:

```
    .syntax unified
    .cpu cortex-m4
    .text
    .global main
main:
    @ ===== Register and Memory Access =====
    LDR R0, =0x25       @ Load immediate value 25H into register R0
    LDR R1, =0x10       @ Load immediate value 10H into register R1
    ADR R2, memory      @ Load memory address into R2
    STR R0, [R2]        @ Store R0 (25H) into memory address

    @ ===== Arithmetic Operations =====
    ADD R3, R0, R1      @ R3 = R0 + R1 (25H + 10H)
    STR R3, [R2, #4]    @ Store result in memory (offset +4)

    SUB R4, R3, R1      @ R4 = R3 - R1 (35H - 10H)
    STR R4, [R2, #8]    @ Store result in memory (offset +8)

    MOV R5, #5          @ Load 5 into R5
    MOV R6, #3          @ Load 3 into R6
    MUL R7, R5, R6      @ R7 = R5 * R6 (5 * 3 = 15)
    STR R7, [R2, #12]   @ Store result
```

```
    MOV R5, #16         @ Load 10H (16) into R5
    MOV R6, #2          @ Load 2 into R6
    SDIV R8, R5, R6     @ R8 = R5 / R6 (16 / 2)
    STR R8, [R2, #16]   @ Store quotient

    @ ===== Logical Operations =====
    MOV R9, #0xF0       @ Load R9 with F0H (1111 0000)
    AND R10, R9, #0xF0  @ R10 = R9 AND F0H
    STR R10, [R2, #20]  @ Store result

    ORR R11, R9, #0x0F  @ R11 = R9 OR 0FH
    STR R11, [R2, #24]  @ Store result

    EOR R12, R9, #0xFF  @ R12 = R9 XOR FFH
    STR R12, [R2, #28]  @ Store result

    MVN R13, R9         @ R13 = NOT R9
    STR R13, [R2, #32]  @ Store result

    B main              @ Infinite loop (for embedded systems)

    .data
memory: .word 0         @ Reserve memory space
```

Explanation of Instructions

1. **Register and Memory Access**
    - LDR R0, =0x25 → Load **25H** into **R0**.
    - ADR R2, memory → Get the memory address.
    - STR R0, [R2] → Store **R0** into memory.
2. **Arithmetic Operations**
    - ADD R3, R0, R1 → Add **R0 + R1**.
    - SUB R4, R3, R1 → Subtract **R1** from **R3**.
    - MUL R7, R5, R6 → Multiply **R5 × R6**.
    - SDIV R8, R5, R6 → Divide **R5 / R6**.
    -

3. **Logical Operations**
    - AND R10, R9, #0xF0 → AND operation.
    - ORR R11, R9, #0x0F → OR operation.
    - EOR R12, R9, #0xFF → XOR operation.
    - MVN R13, R9 → NOT operation.

Expected Memory Output

Memory Address	Value (Hex)	Operation
memory	0x25	Initial value (R0)
memory+4	0x35	25H + 10H
memory+8	0x25	35H - 10H
memory+12	0x0F	5 × 3
memory+16	0x08	16 / 2 (Quotient)
memory+20	0xF0	AND Operation
memory+24	0xFF	OR Operation
memory+28	0x0F	XOR Operation
memory+32	0xFFFFFF0F	NOT Operation

Procedure-2:

Would you like to add **branching (B, BL) or loops (CMP, BNE, BEQ)?**

LPC2148 BASIC REG/MEM VISITING AND SIMPLE ARITHMETIC/LOGIC COMPUTING

LPC2148 ARM7 Assembly: Basic Register/Memory Access and Simple Arithmetic/Logic Operations

Aim: This assembly program demonstrates fundamental operations on the **LPC2148 (ARM7TDMI)** microcontroller:

✅**Register and Memory Access** (LDR, STR, MOV, ADR)

✅**Basic Arithmetic Operations** (ADD, SUB, MUL, UDIV)

✅**Logical Operations** (AND, ORR, EOR, MVN)

Assembly Code:

```
    .syntax unified
    .cpu arm7tdmi
    .text
    .global main
main:
    @ ===== Register and Memory Access =====
    LDR R0, =0x25       @ Load immediate value 25H into register R0
    LDR R1, =0x10       @ Load immediate value 10H into register R1
    LDR R2, =memory     @ Load memory address into R2
    STR R0, [R2]        @ Store R0 (25H) into memory

    @ ===== Arithmetic Operations =====
    ADD R3, R0, R1      @ R3 = R0 + R1 (25H + 10H)
    STR R3, [R2, #4]    @ Store result in memory (offset +4)

    SUB R4, R3, R1      @ R4 = R3 - R1 (35H - 10H)
    STR R4, [R2, #8]    @ Store result in memory (offset +8)
    MOV R5, #5          @ Load 5 into R5
    MOV R6, #3          @ Load 3 into R6
    MUL R7, R5, R6      @ R7 = R5 * R6 (5 * 3 = 15)
    STR R7, [R2, #12]   @ Store result
```

```
MOV R5, #16         @ Load 10H (16) into R5
MOV R6, #2          @ Load 2 into R6
UDIV R8, R5, R6     @ R8 = R5 / R6 (16 / 2) (Unsigned division)
STR R8, [R2, #16]   @ Store quotient

@ ===== Logical Operations =====
MOV R9, #0xF0       @ Load R9 with F0H (1111 0000)
AND R10, R9, #0xF0  @ R10 = R9 AND F0H
STR R10, [R2, #20]  @ Store result

ORR R11, R9, #0x0F  @ R11 = R9 OR 0FH
STR R11, [R2, #24]  @ Store result

EOR R12, R9, #0xFF  @ R12 = R9 XOR FFH
STR R12, [R2, #28]  @ Store result

MVN R13, R9         @ R13 = NOT R9
STR R13, [R2, #32]  @ Store result

B main              @ Infinite loop (for embedded systems)

.data

memory: .word 0         @ Reserve memory space
```

Explanation of Instructions

1. **Register and Memory Access**
    - LDR R0, =0x25 → Load **25H** into **R0**.
    - LDR R2, =memory → Load memory address into **R2**.
    - STR R0, [R2] → Store **R0** into memory.
2. **Arithmetic Operations**
    - ADD R3, R0, R1 → Add **R0 + R1**.
    - SUB R4, R3, R1 → Subtract **R1** from **R3**.
    - MUL R7, R5, R6 → Multiply **R5 × R6**.
    - UDIV R8, R5, R6 → Divide **R5 / R6**.

3. **Logical Operations**
    - AND R10, R9, #0xF0 → AND operation.
    - ORR R11, R9, #0x0F → OR operation.
    - EOR R12, R9, #0xFF → XOR operation.
    - MVN R13, R9 → NOT operation.

Expected Memory Output

Memory Address	Value (Hex)	Operation
memory	0x25	Initial value (R0)
memory+4	0x35	25H + 10H
memory+8	0x25	35H - 10H
memory+12	0x0F	5 × 3
memory+16	0x08	16 / 2 (Quotient)
memory+20	0xF0	AND Operation
memory+24	0xFF	OR Operation
memory+28	0x0F	XOR Operation
memory+32	0xFFFFFF0F	NOT Operation

Viva questions:

1. How many general-purpose registers does ARM have?

2. How do you access memory in ARM LPC2148? What instructions are used?

3. Explain how to perform an increment or decrement in ARM.

4. What flags are affected by arithmetic operations?

5. What are logical instructions available in ARM?

INTRODUCTION TO ARM BOARD (LPC2148)

INTRODUCTION TO ARM BOARD (LPC2148)

This section of the document introduces LPC2148 microcontroller board based on a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation and embedded trace support, that combine microcontrollers with embedded high-speed flash memory ranging from 32 kB to 512 kB. A 128-bit wide memory interface and unique accelerator architecture enable 32-bit code execution at the maximum clock rate. For critical code size applications, the alternative 16-bit Thumb mode reduces code by more than 30% with minimal performance penalty. The meaning of LPC is Low Power Low Cost microcontroller. This is 32 bit microcontroller manufactured by Philips semiconductors (NXP). Due to their tiny size and low power consumption, LPC2148 is ideal for applications where miniaturization is a key requirement, such as access control and point-of-sale.

Features of ARM Microcontroller:

- 16-bit/32-bit ARM7TDMI-S microcontroller in a tiny LQFP64 package.
- 8 kB to 40 kB of on-chip static RAM and 32 kB to 512 kB of on-chip flash memory; 128-bit wide interface/accelerator enables high-speed 60 MHz operation.
- In-System Programming/In-Application Programming (ISP/IAP) via on-chip boot loader software, single flash sector or full chip erase in 400 ms and programming of 256 Bytes in 1 ms Embedded ICE RT and Embedded Trace interfaces offer real-time debugging with the on-chip Real Monitor software and high-speed tracing of instruction execution.
- USB 2.0 Full-speed compliant device controller with 2kB of endpoint RAM. In addition, the LPC2148 provides 8 kB of on-chip RAM accessible to USB by DMA.
- One or two (LPC2141/42 vs, LPC2144/46/48) 10-bit ADCs provide a total of 6/14 analog inputs, with conversion times as low as 2.44 ms per channel.
- Single 10-bit DAC provides variable analog output (LPC2148 only)
- Two 32-bit timers/external event counters (with four capture and four compare channels each), PWM unit (six outputs) and watchdog.
- Low power Real-Time Clock (RTC) with independent power and 32 kHz clock input.
- Multiple serial interfaces including two UARTs, two Fast I2C-bus (400 kbit/s), SPI and SSP with buffering and variable data length capabilities.

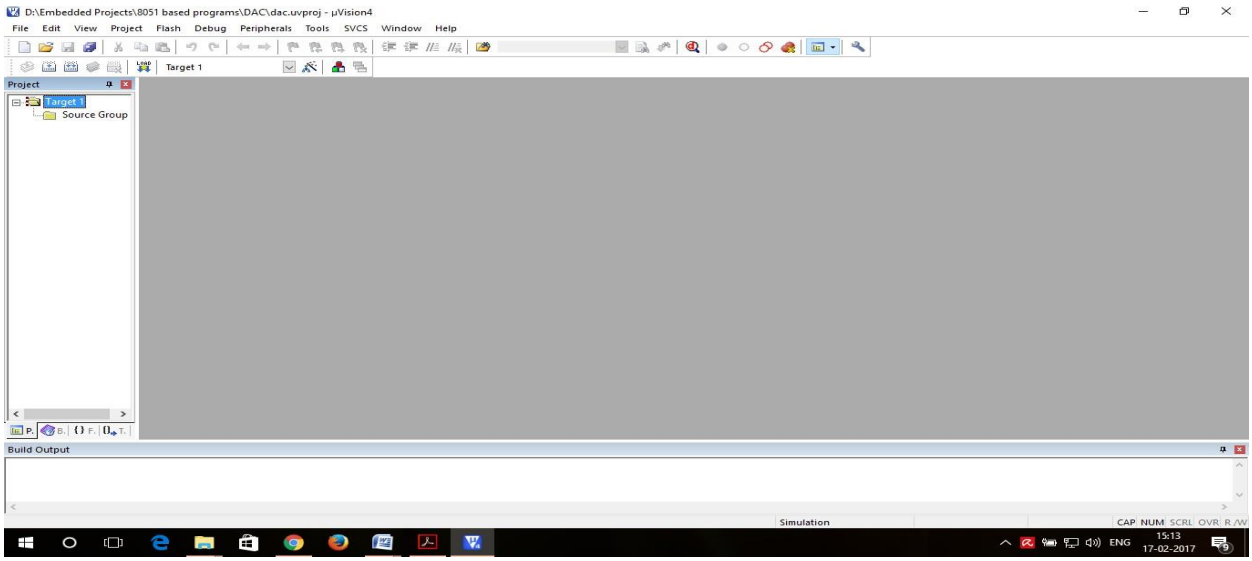
- Vectored Interrupt Controller (VIC) with configurable priorities and vector addresses.
- Up to 45 of 5 V tolerant fast general purpose I/O pins in a tiny LQFP64 package.
- Up to nine edge or level sensitive external interrupt pins available.
- 60 MHz maximum CPU clock available from programmable on-chip PLL with settling time of 100 ms.
- Power saving modes include Idle and Power-down
- Individual enable/disable of peripheral functions as well as peripheral clock scaling for additional power optimization.
- Processor wake-up from Power-down mode via external interrupt or BOD.
- Single power supply chip with POR and BOD circuits:
    - CPU operating voltage range of 3.0 V to 3.6 V (3.3 V ± 10 %) with 5 V tolerant I/O.

INTRODUCTION TO KEIL µ VISION 4 FOR LPC2148

The LPC2148 microcontroller is supported by various commercially available IDEs for compiling and debugging of the code. Keil being one of them is the widely used IDE for LPC family of microcontrollers. The µVision4 IDE is Windows-based software development platforms that combines a robust editor, project manager, and make facility. µVision4 integrates all tools including the C compiler, macro assembler, linker/locator, and HEX file generator. The evaluation version of Keil µVision4 IDE is used for demonstrating the codes. The open source community has been doing a lot in the development of open source tools for ARM architecture based Microcontroller. The open source tools are available at zero cost and are being improved with time. Eclipse being one of them and is most commonly used IDE due to its unique features like auto complete, project tree, etc. It requires GCC tool chain for code compilation.

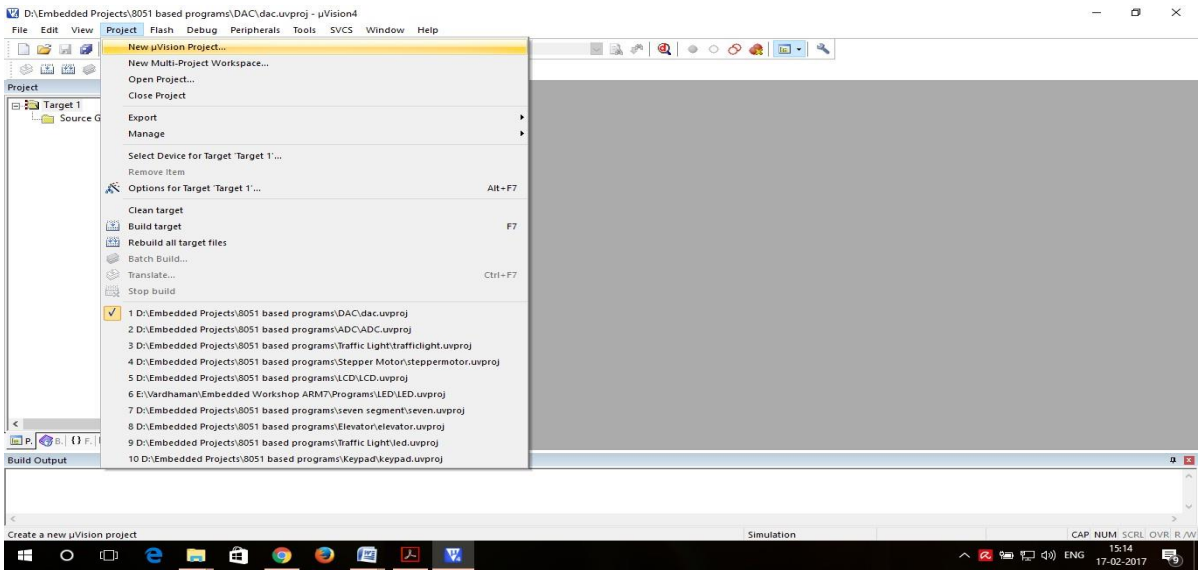
Create a Project in Keil for LPC2148 development board:

Step1: Click for KEIL IVISION4 Icon . This is appearing after Installing Keil µ vision 4. Give a double click on µvision4 icon on the desktop; it will generate a window as shown

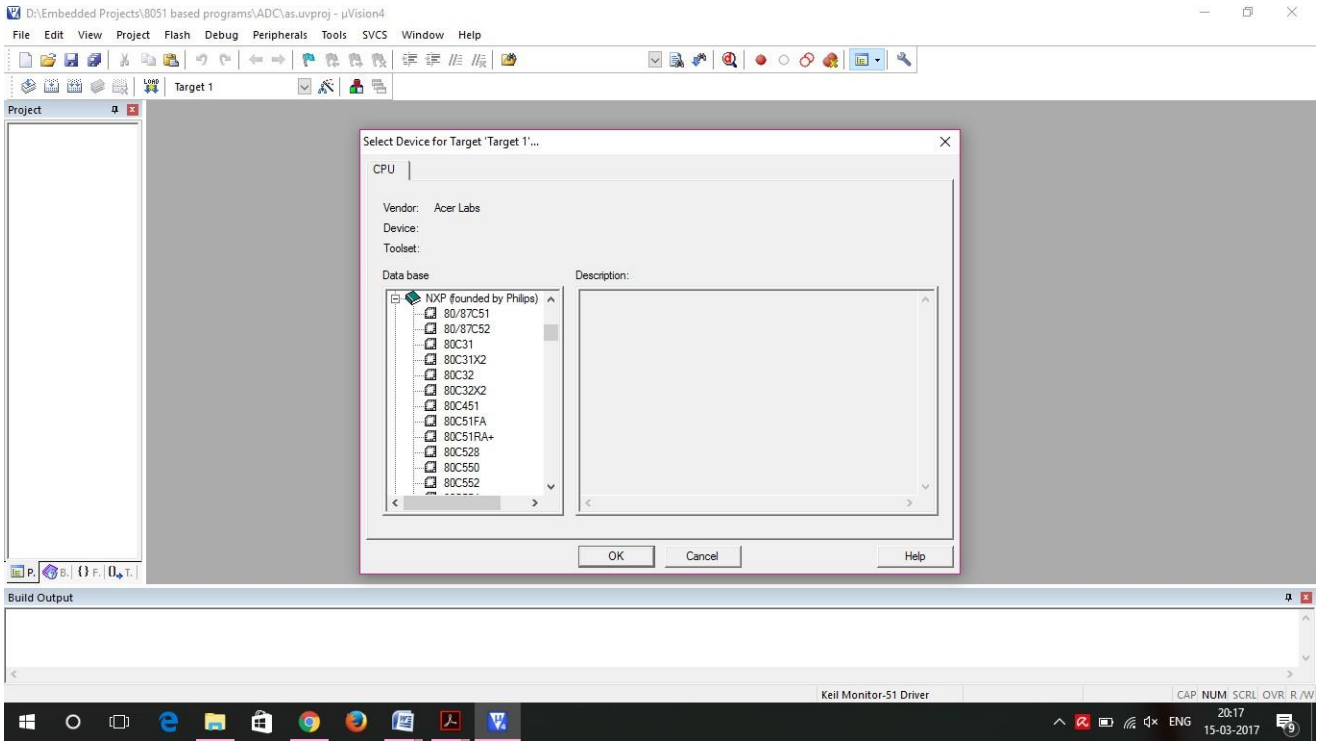


below:

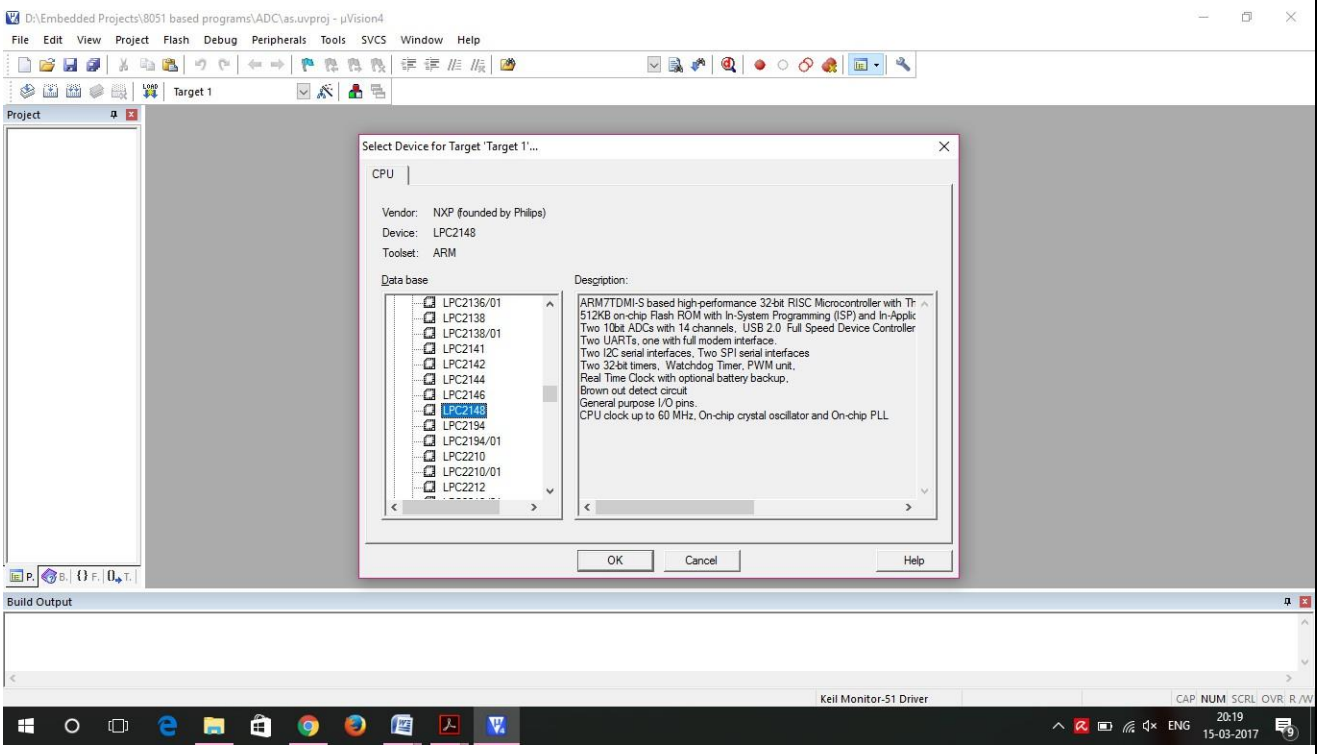
Step 2: To create new project, go to project, select new µvision project.



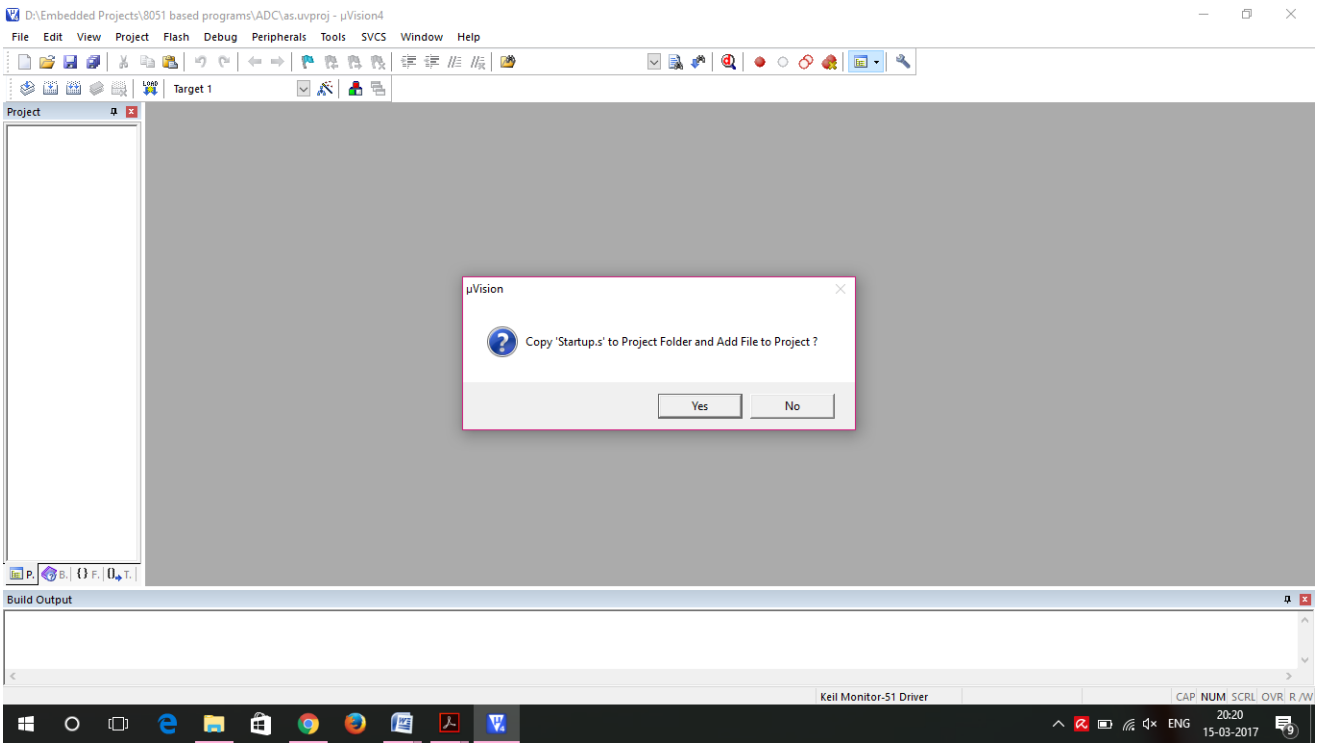
Step 3: Select a drive where you would like to create your project.
Step 4: Create a new folder and name it with your project name.
Step 5: Open that project folder and give a name of your project executable file and save it.
Step 6: After saving, it will show the below window there select your microcontroller company i.e. NXP.



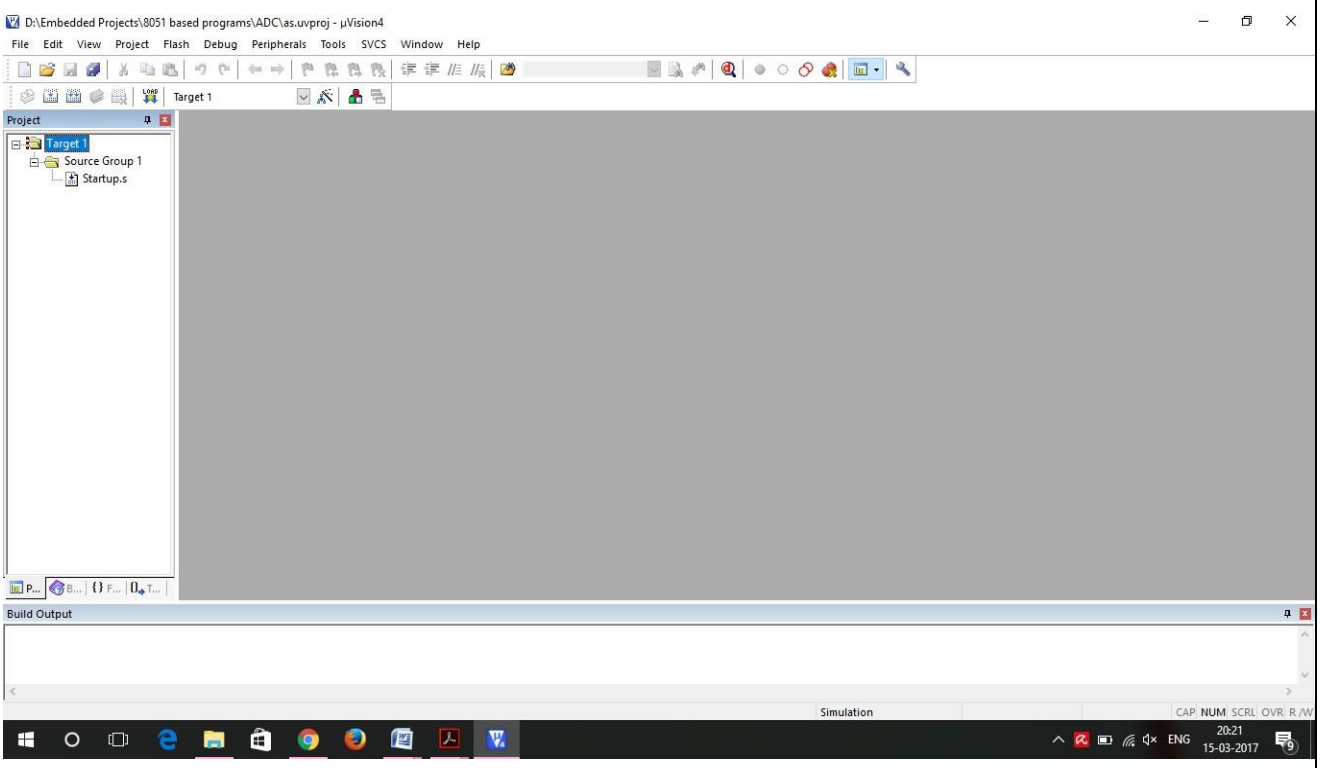
Step 7: Now select target device as LPC2148 and click OK to continue.



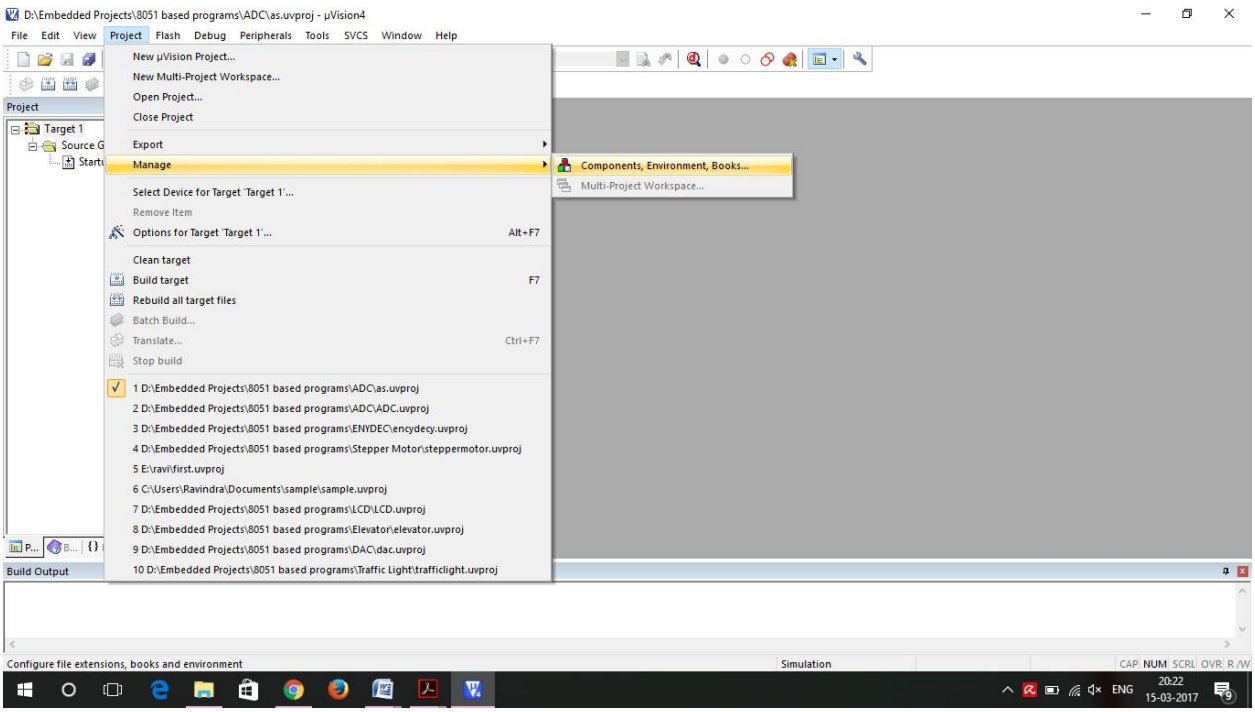
Step 8: Click Yes to copy Startup's file to project folder. This file configures stack, PLL and maps memory as per the configurations in the wizard. It is discussed in the later sections.



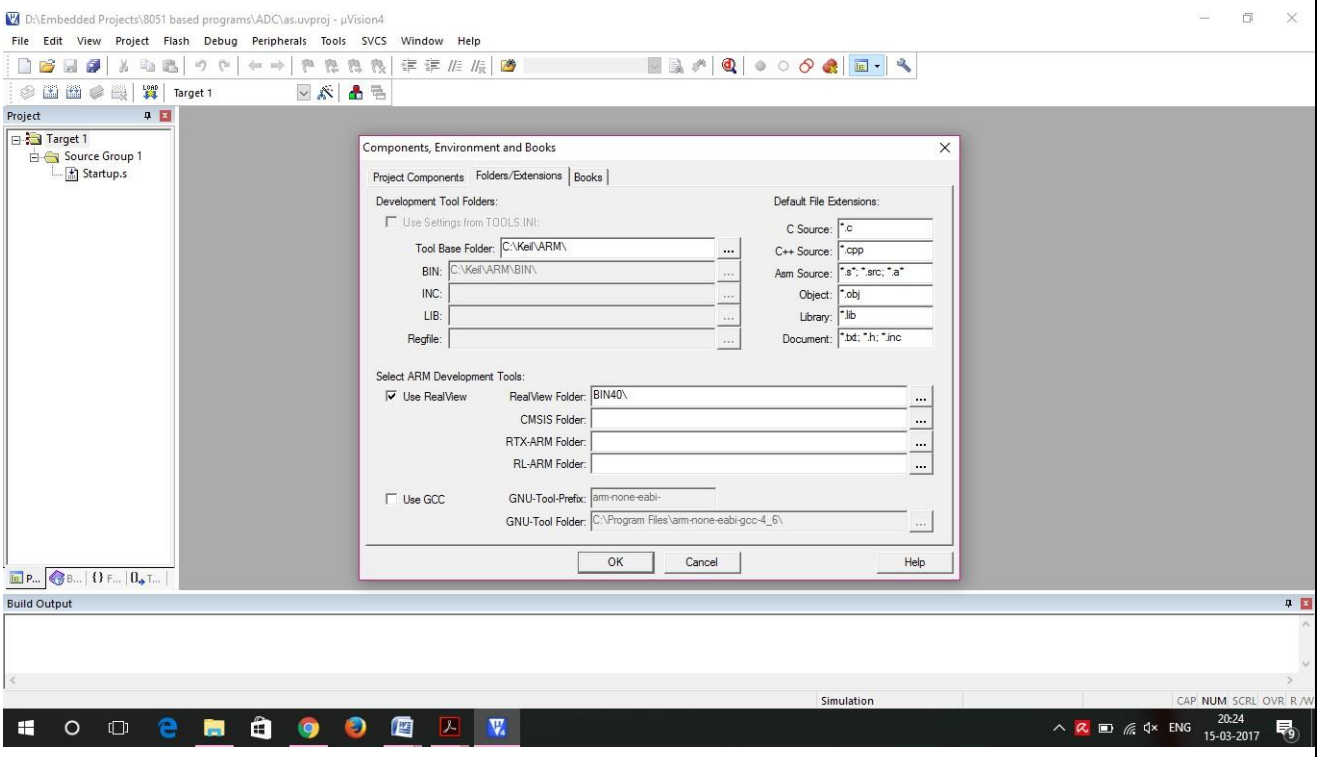
Step 9: Observe the project explorer area in the main window on left side.



Step 10: Now click Project>Manage>Components, Environments, Books from the main menu to ensure compiler settings.

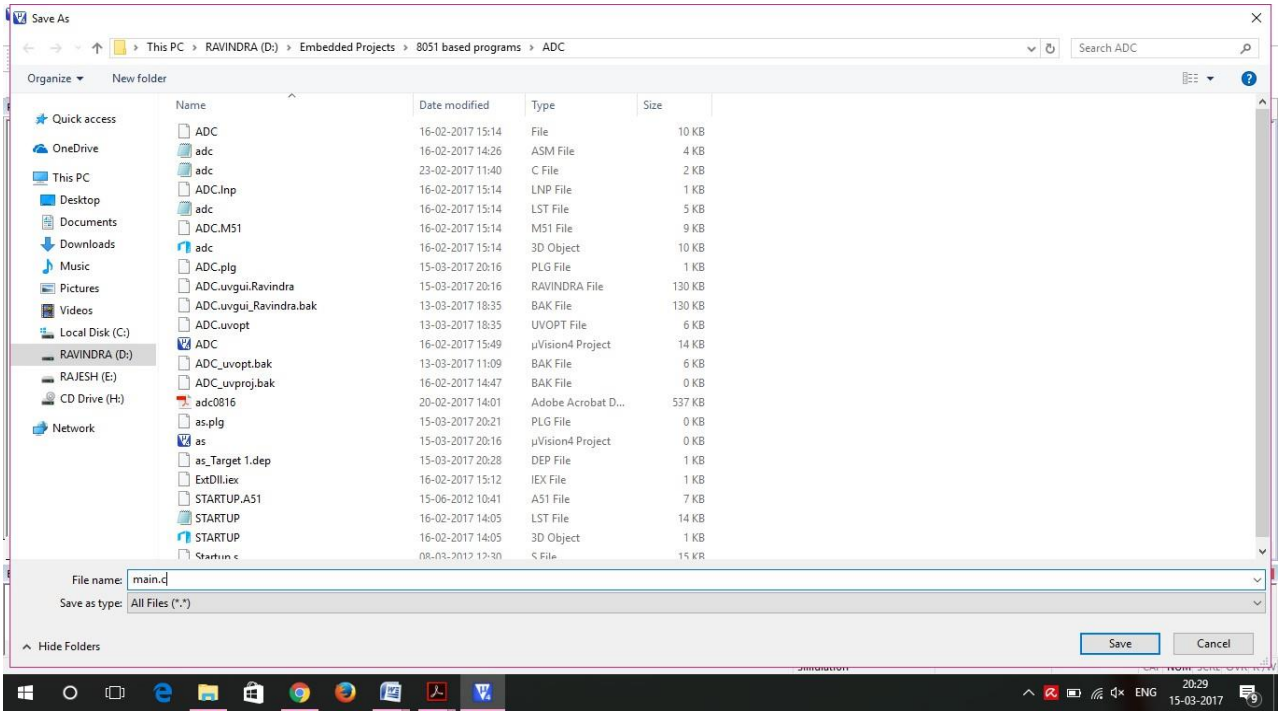


Step 11: In the Folders/Extensions tab ensure the compiler settings are as shown in the figure below. If you have installed Keil software at a different location then change Tool Base Folder location. Click OK to continue.

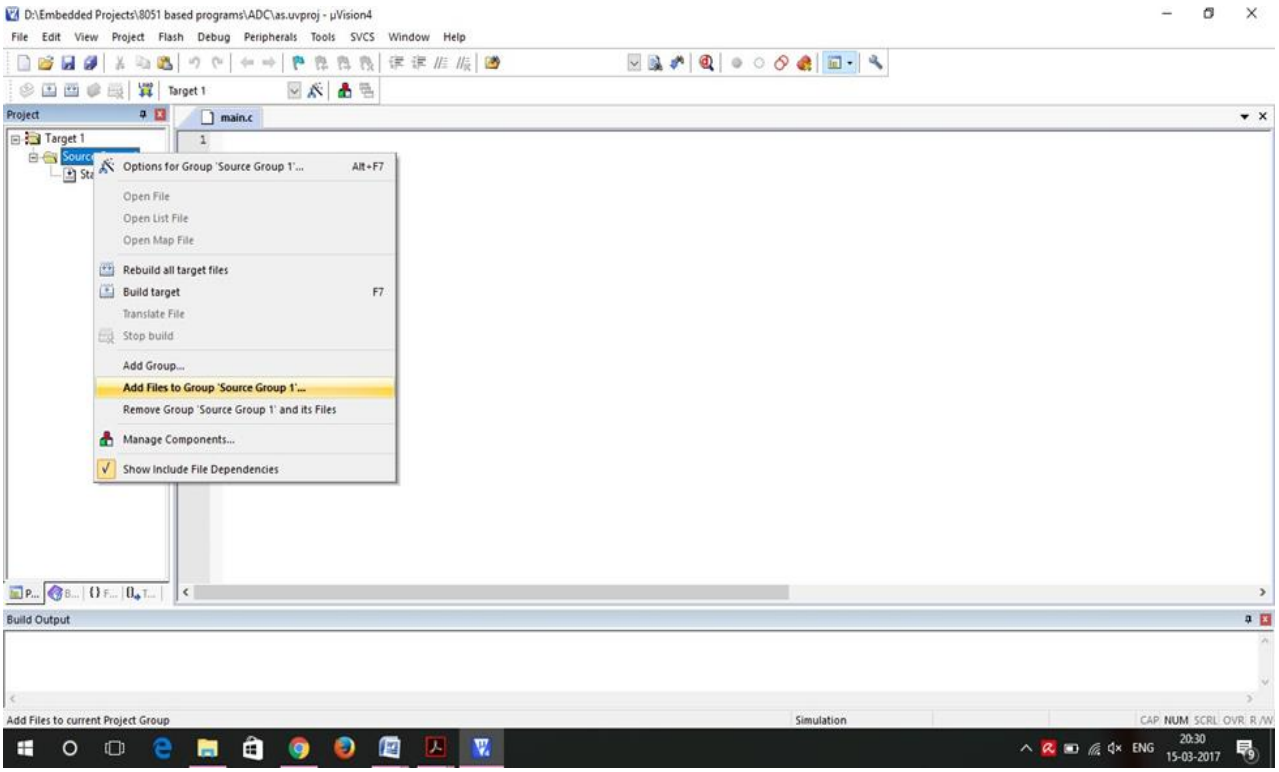


Step 12: Now click File>New to create a new file.

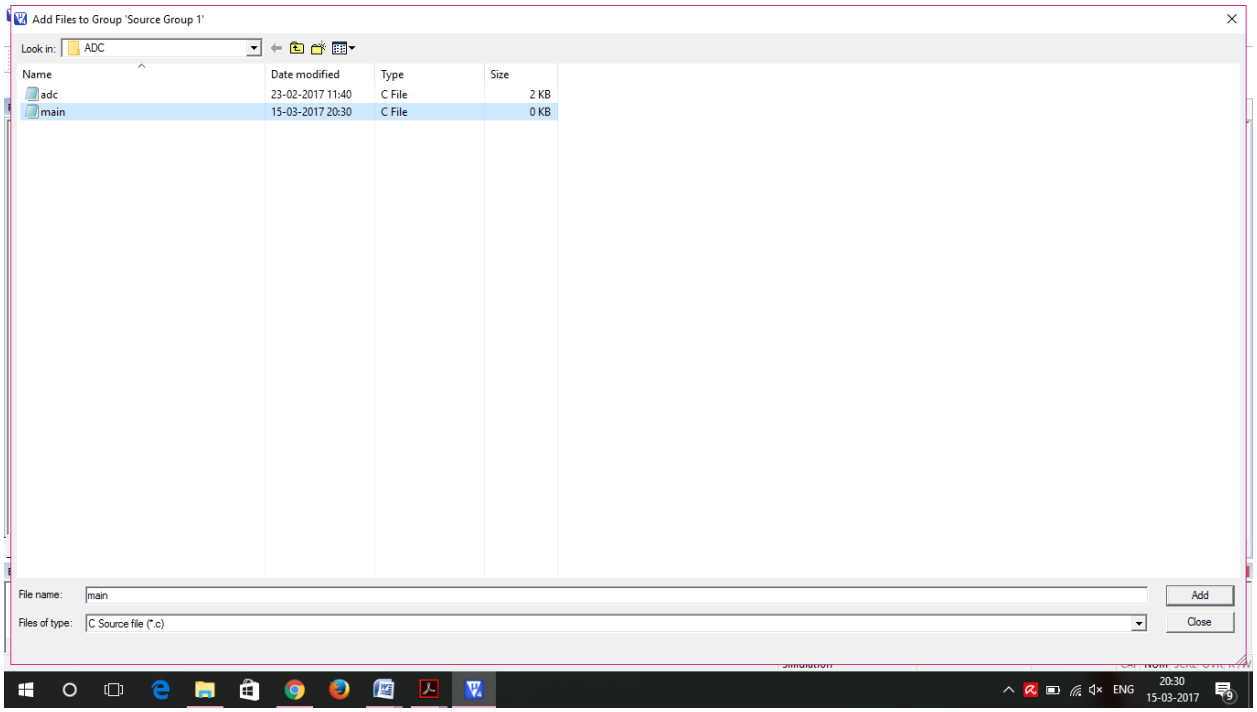
Step 13: Save the new file in to the same folder that was created earlier and name it as main.c and click Save to continue



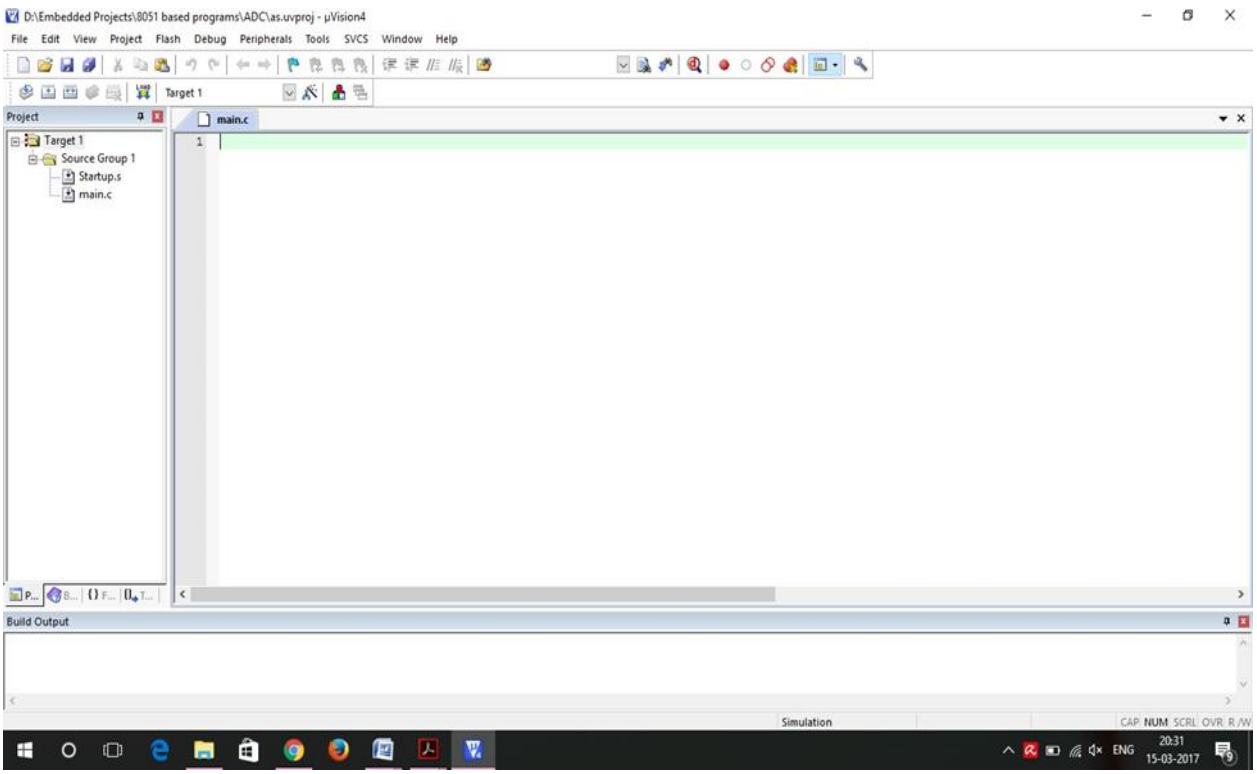
Step 14: Now add “main.c” to the source group by right clicking on the Source Group 1 From the project explorer and select the highlighted option as shown in the fig. below.



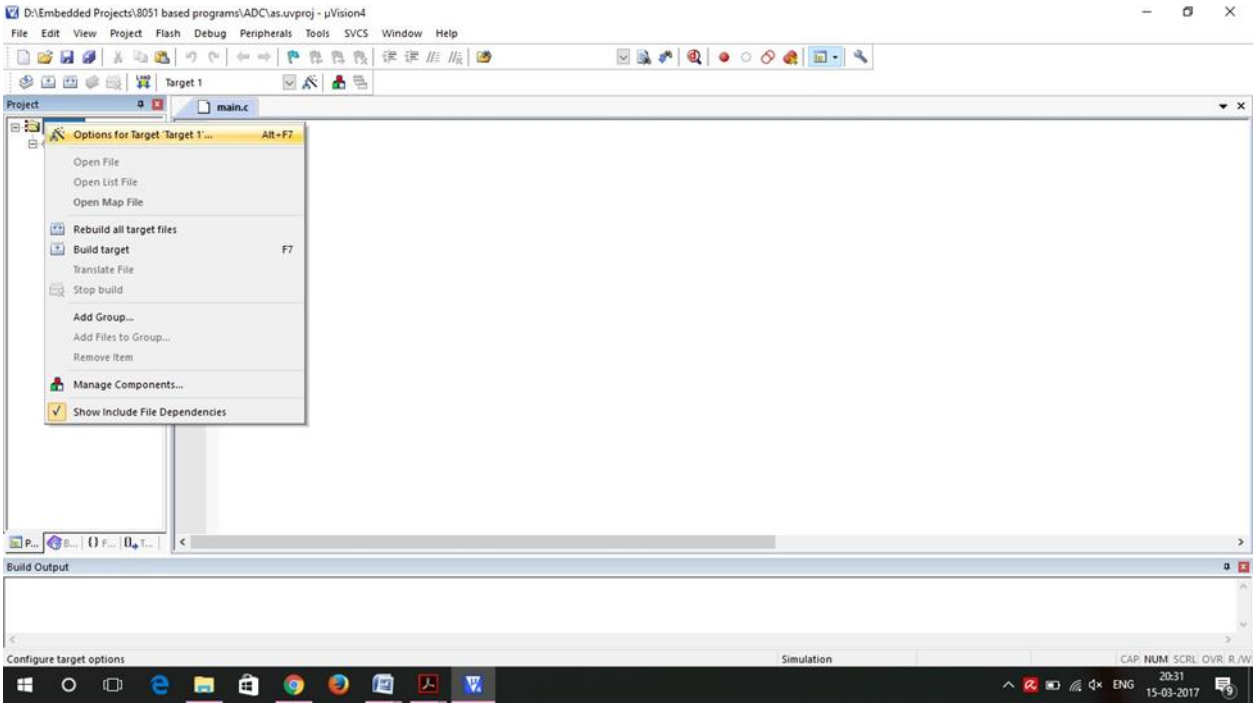
Step 15: Select “main.c” file to be added and click ADD to continue.



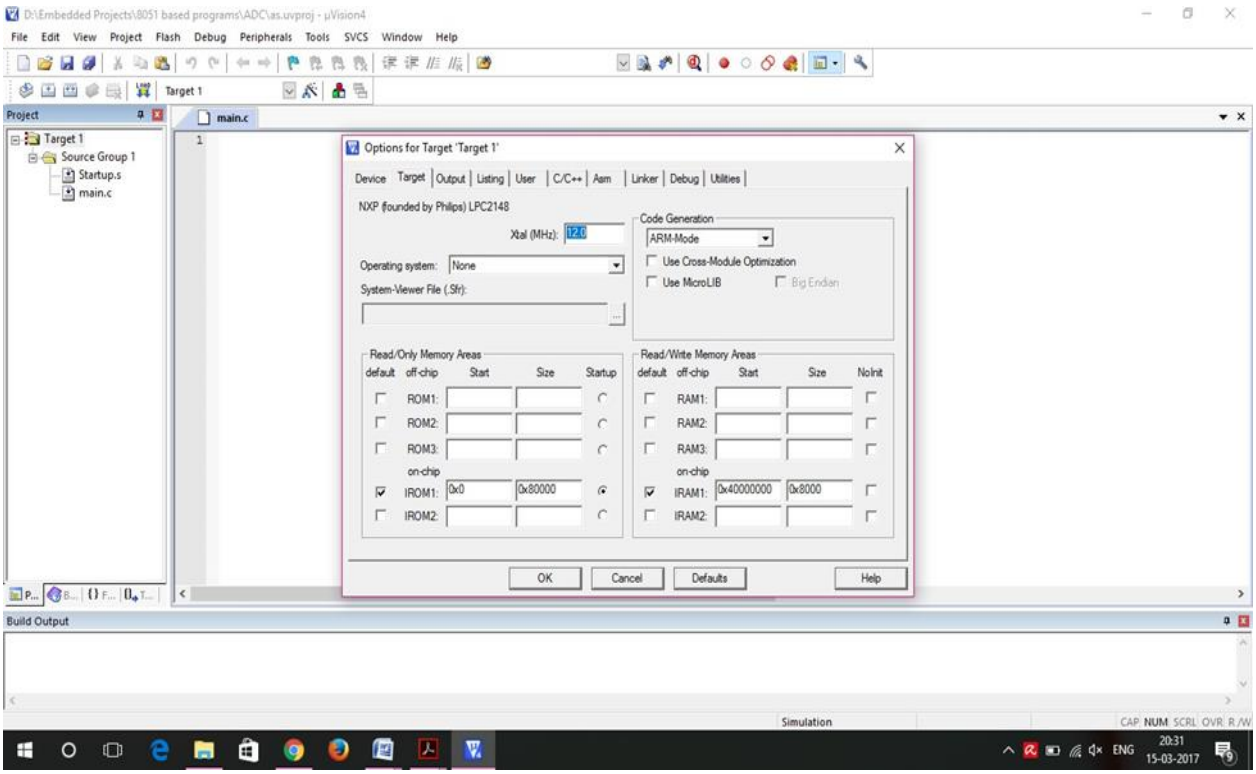
Step 16: Observe that "main.c" file is added to the source group in the project explorer window



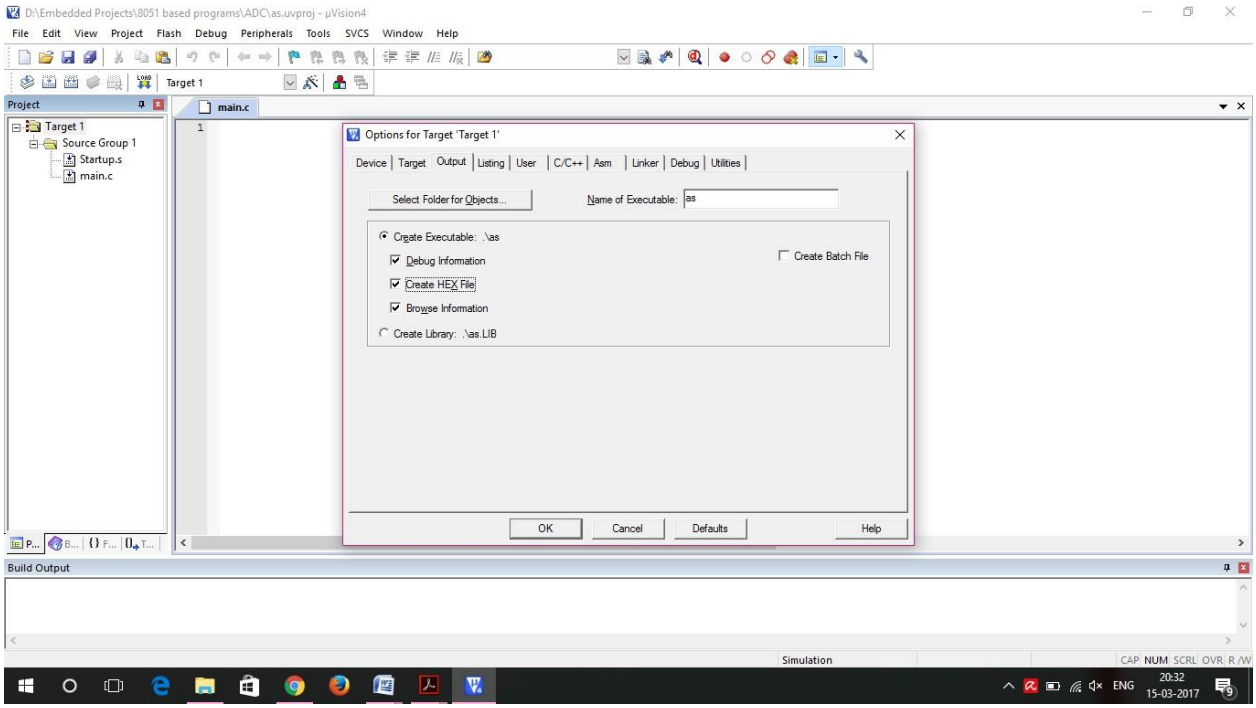
Step 17: Right click Target1 in the project explorer window and select the highlighted option as shown in the fig. below.



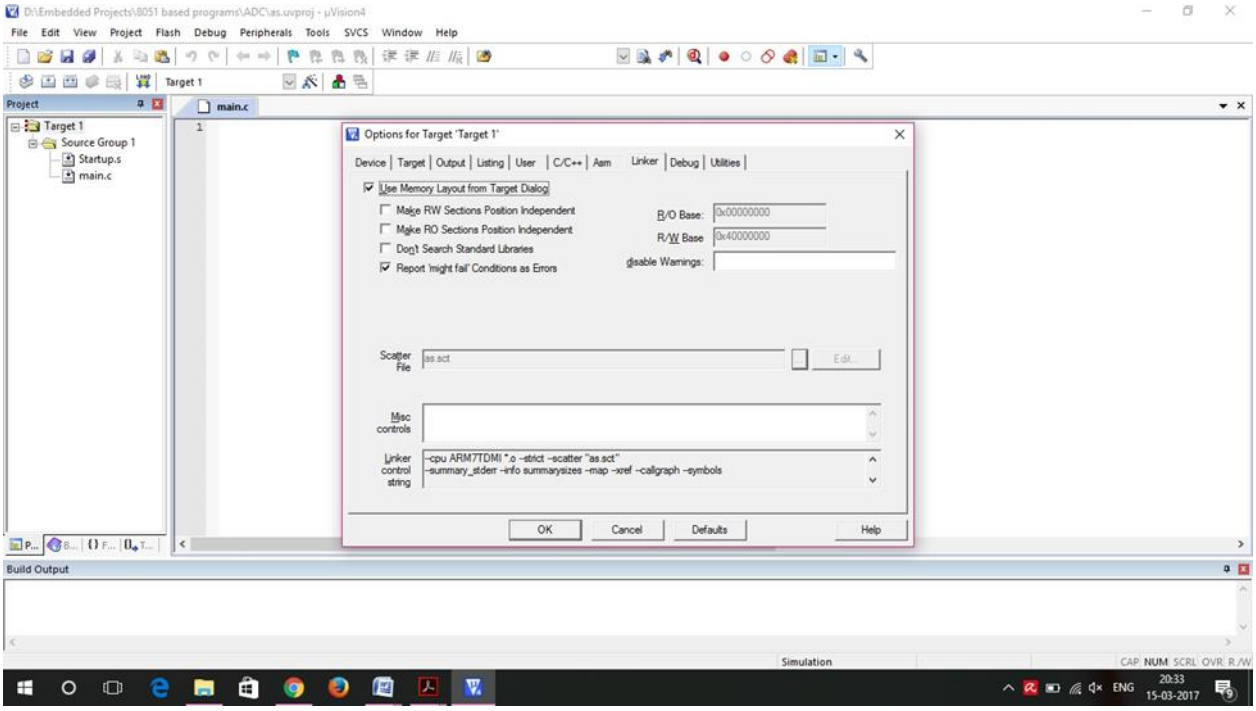
Step 18: In the appearing window select Target tab and set Xtal. frequency as 12MHz.



Step 19: In the Output tab ensure that Create HEX File option is selected.

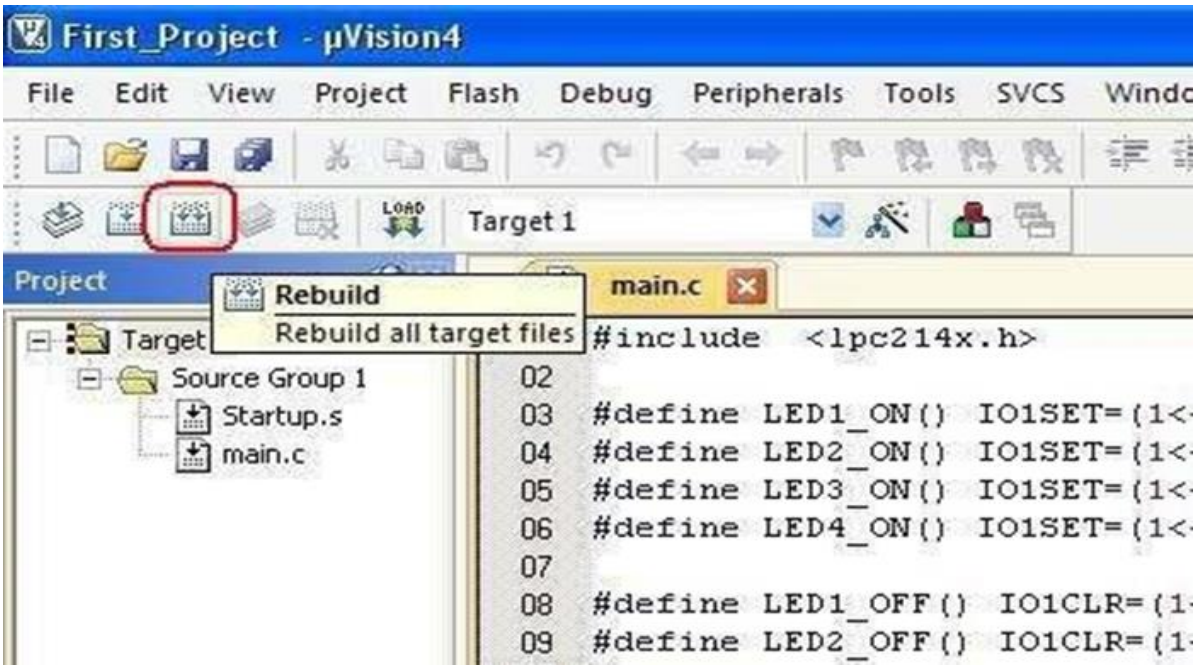


Step 20: In the Linker tab ensure that the highlighted option is selected and click OK to Continue.

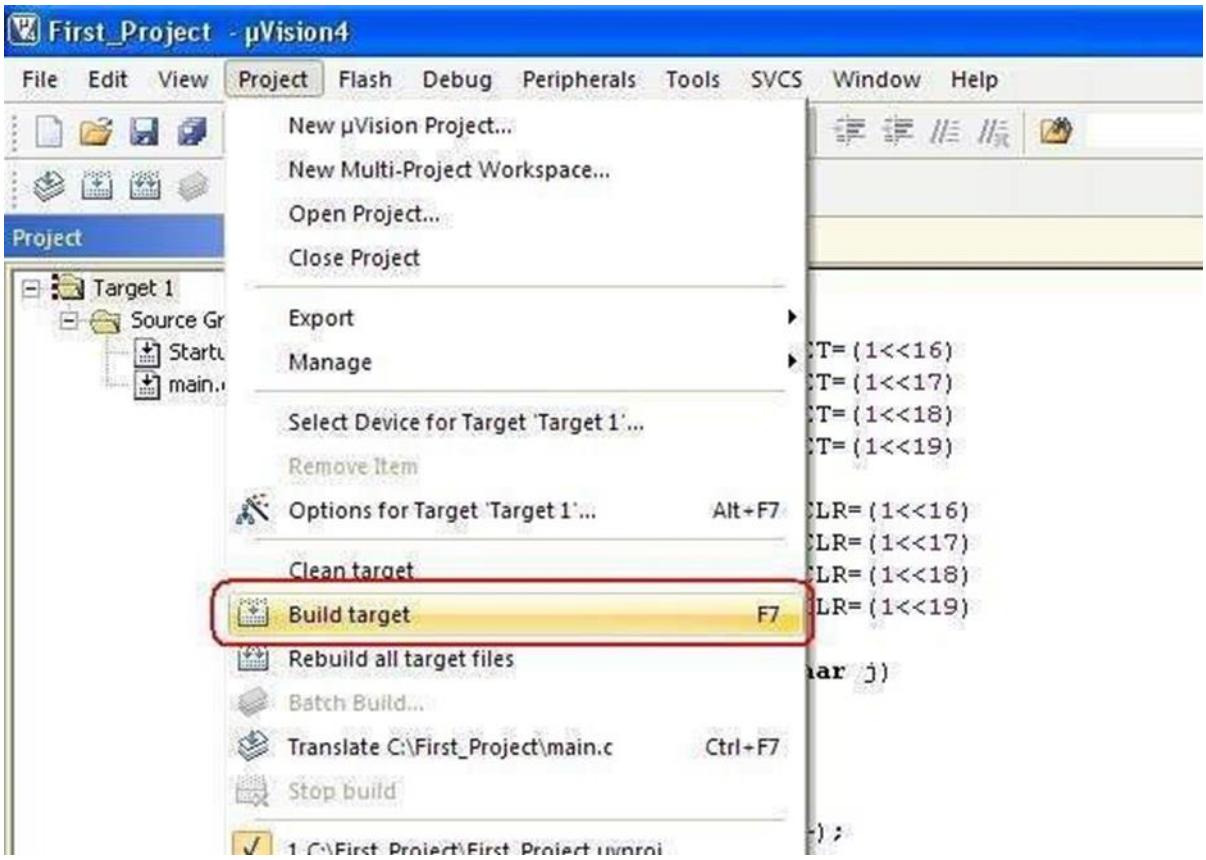


Step 21: Now since the project is almost setup we can start writing code in the "main.c" file that was created earlier. For demonstration purpose you can copy the following code and paste it in the "main.c" file.

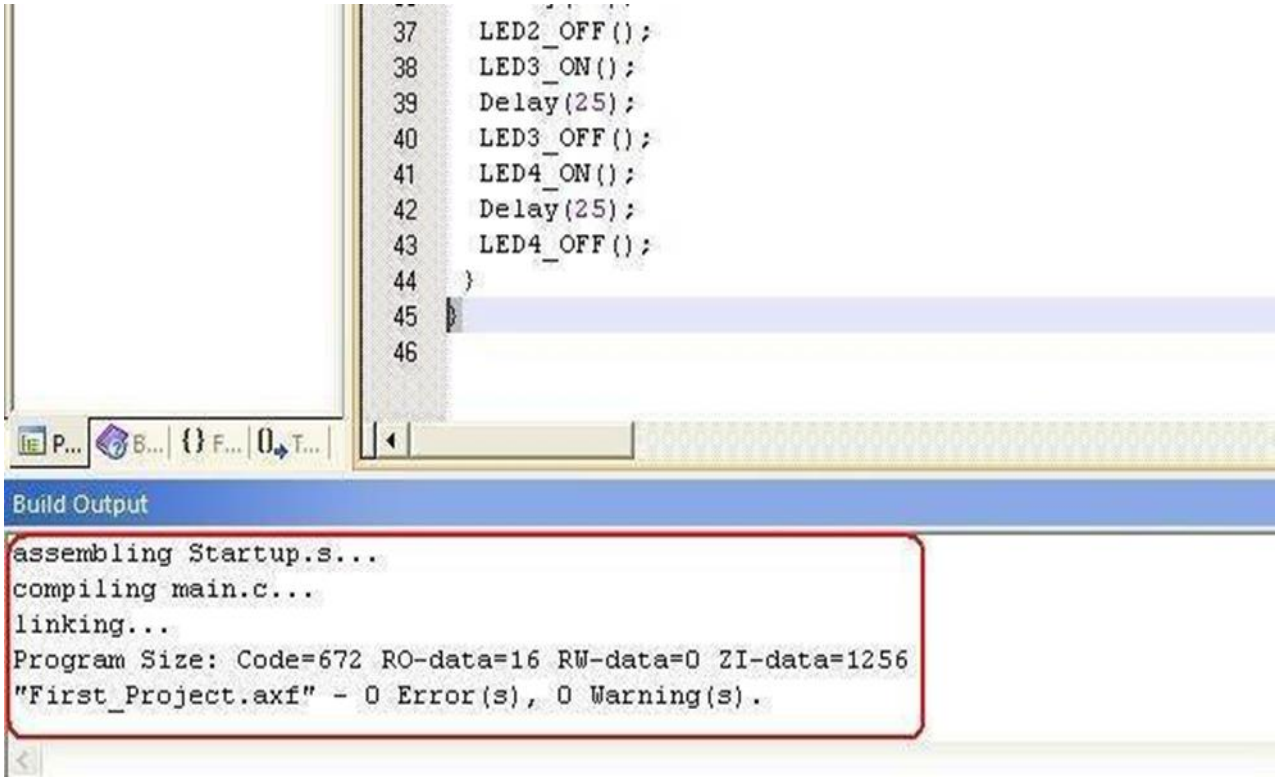
Step 22: Now build the project by clicking on Rebuild button on the main toolbar.



Step 23: You can alternatively build project by clicking on Project>Build Target from the main menu.



Step 24: You can observe the build process in the output window. If any errors, rectify it by double clicking on it and you will be pointed to the erroneous line.


37 LED2_OFF();
38 LED3_ON();
39 Delay(25);
40 LED3_OFF();
41 LED4_ON();
42 Delay(25);
43 LED4_OFF();
44 }
45 }
46
P...
B...
{} F...
{}, T...
Build Output
assembling Startup.s...
compiling main.c...
linking...
Program Size: Code=672 RO-data=16 RW-data=0 ZI-data=1256
"First_Project.axf" - 0 Error(s), 0 Warning(s).

EXPERIMENT 10: Changing ARM state mode by using MRS/MMSR instruction and specify a start address of the text segment by using command line.

Aim: Write a program to change the mode of operation of ARM using MRS/MSR instructions.

Apparatus:

1. Computer
2. Keil µ vision 4

Hints to write Program:

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

Example: The MSR first copies the *CPSR* into register *r1*. The BIC instruction clears bit 7 of *r1*. Register *r1* is then copied back into the *CPSR*, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the *CPSR* and only modifies the*I* bit in the control field.

PRE cpsr =

nzcvqIFt_SVC MRS r1,

cpsr

BIC r1, r1, #0x80; 0b01000000

MSR cpsr_c, r1

POST cpsr = nzcvqiFt_SVC

Viva questions:

1. What are the different modes of operation in ARM?
2. Which mode is entered when a hardware interrupt occurs?
3. What is the default mode after reset in ARM?
4. Can a program running in User Mode switch to Supervisor Mode directly?
5. What happens to the PC and CPSR during an exception or mode switch?

EXPERIMENT 11: ARM programming in C language using KEIL IDE.

Aim: Write a program to generate a delay using timers.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

```
void delay(void) //delay
{
int i;
for(i=0;i<=10;i
++)
{
_nop_();
}
}
void delay10(void)
{
int i;
for(i=0;i<=10;i
++)
{
delay();
}
}

__main()
{
int i=5;
for(;;)
{
delay10();
}
}
```

Viva questions:

1. Write a C program to read a switch and turn ON/OFF an LED on ARM.
2. How is a delay function written in C for ARM microcontrollers?
3. How do you enable interrupts using C in ARM-based microcontrollers?
4. How do you use PWM in ARM through C code?
5. How do you configure GPIO pins in C for LPC2148?

EXPERIMENT 12: Write a random number generation function using assembly language. Call this function from a C program to produce a series of random numbers and save them in the memory.

Aim: Write a program to generate random numbers.

Apparatus:

1. Computer
2. Keil µ vision 4

Hints to write Program:

To generate truly random numbers requires special hardware to act as a source of random noise. However, for many computer applications, such as games and modeling, speed of generation is more important than statistical purity. These applications usually use *pseudorandom* numbers.

Pseudorandom numbers aren't actually random at all; a repeating sequence generates the numbers. However, the sequence is so long, and so scattered, that the numbers appear to be random. Typically we obtain *Rk* , the *k*th element of a pseudorandom sequence, by iterating a simple function of *Rk*−1:

$$Rk = f\,(Rk-1)$$

For a fast pseudorandom number generator we need to pick a function *f* (*x*) that is easy to compute and gives random-looking output. The sequence must also be very long before it repeats. For a sequence of 32-bit numbers the longest length achievable is clearly 2^{32}.

A linear congruence generator uses a function of the following form.

$$f(x) = (a*x+c) \% m;$$

. For fast computation, we would like to take $m = 2^{32}$. The theory in Knuth assures us that if a % 8 = 5 and *c* = a, then the sequence generated has maximum length of 2^{32} and is likely to appear random. For example, suppose that a=0x91e6d6a5.

Then the following iteration generates a pseudorandom sequence:

$$\text{MLA r, a, r, a ; r\_k} = (a*r\_(k-1) + a) \bmod 2\Lambda32$$

Since *m* is a power of two, the low-order bits of the sequence are not very random. Use the high-order bits to derive pseudorandom numbers of a smaller range. For example, set *s* = *r* _ 28 to generate a four-bit random number *s* in the range 0–15. More generally, the following code generates a pseudorandom number between 0 and *n*:

```
; r is the current random seed
; a is the multiplier (eg 0x91E6D6A5)
; n is the random number range (0...n-1)
; t is a scratch register
MLA r, a, r, a ; iterate random number
generator UMULL t, s, r, n ; s = (r*n)/2^32
; r is the new random seed
; s is the random result in range 0 ... n-1
```

Viva questions:

1. What’s the difference between **true random** and **pseudo-random** numbers?
2. Which function is used in C to generate random numbers?
3. Why can't you use time(NULL) on basic ARM microcontrollers like LPC2148?
4. What is the range of values returned by rand()?
5. What is a seed in random number generation? Why is it important?

EXPERIMENT 13: Configure and read/write the memory space. Use assembly and C language to read/write words, half-words, bytes, half bytes from/to RAM.

Aim: Write a program to read/write words, half-words, bytes, half bytes from/to RAM.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

```
LDR     R1, [R0]              /* load R1 from the address in R0 */
LDR     R8, [R3,#4]           /* load R8 from the address in R3+4 */
LDR     R12, [R13, #-4]       /* load R12 from R13 – 4 */
STR     R2, [R1,#0X100]       /* store R2 to the address in R1 + 0x100 */
LDRB    R5, [R9]              /* load byte into R5 from R9 */        /* (zero top 3 bytes) */
LDRB    R3, [R8, #3]          /* load byte to R3 from R8 + 3 */              /* (zero top 3 bytes) */
STRB    R4, [R10, #0X200]     /* store byte from R4 to R10  + 0x200 */
LDR     R11, [R1, R2]         /* load R11 from the address in R1 + R2 */
STRB    R10, [R7, -R4]        /* store byte from R10 to address in R7 – R4 */
LDR     R11, [R3, R5, LSL     /* load R11 from R3 + (R5*4) */
        #2]
LDR     R1, [R0, #4]!         /* load R1 from R0 +4, then R0 = R0 + 4 */
STRB    R7, [R6, #-1]!        /* store byte from R7 to R6 – 1, then R6 = R6 – 1 */
LDR     R3, [R9], #4          /* load R3 from R9, then R9 = R9 + 4*/
STR     R2, [R5], #8          /* store R2 to R5, then R5= R5 + 8 */
LDR     R0, [PC, #40]         /* load R0 from PC + 0x40 ( = address of the LDR instruction + 8
                              +0x40)*/
LDR     R0, [R1], R2          /* load R0 from R1, then R1 = R1 + R2 */
LDRH    R1, [R0]              /* load halfword to R1 from R0 ( Zero top 2 bytes) */
LDRH    R8, [R3, #2]          /* load halfword to R8 from R3 + 2*/
LDRH    R12, [R13, #-6]       /* load halfword to R12 from R13 – 6 */
STRH    R2, [R1, #0X80]       /* store halfword from R2 to R1 + 0x80 */
LDRS    R5, [R9]              /* load signed halfword to R5 from R9 */
H
LDRS    R3, [R8, #3]          /* load signed byte to R3 from R8 + 3  */
B
LDRS    R4, [R10, #0XC1]      /* load signed byte to R4 from R10 + 0xc1  */
B
LDRH    R11, [R1, R2]         /* load halfword into R11 from address in R1+ R2 */
STRH    R10, [R7, -R4]        /* store halfword from R0 to R7 – R4 */
LDRS    R1, [R0, #2]!         /* load signed halfword R1 from R0 + 2, then R0 = R0 + 2 */
H
LDRSB R7, [R6, #-1]!          /* load signed byte to R7 from R6 – 1, then R6 = R6 - 1
```

/ LDRH R3, [R9], #2 / load halfword into R3 from R9 then R9 = R9 + 2*/

STRH R2, [R5], #8 /* store halfword from R2 to R5 then R5 = R5 + 8*/

Viva questions:

1. What type of RAM is used in LPC2148?
2. How do you write data to a specific RAM location in C on ARM
3. What does volatile mean in this context? Why is it used?
4. What's the difference between stack and heap memory in ARM-based systems?
5. How do you simulate reading/writing RAM on Keil uVision?

EXPERIMENT 14: Implement the lighting and winking LEDs of the ARM I/O port via programming.

Aim: Write a program to program blinks the LEDs continuously with a small delay.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

```
#include<lpc2148.H>                    //LPC2148
 Header void delay()
{
for(int i=0x00;i<=0xff;i++)
for(int j=0x00;j<=0xFf;j++)  ;      // Delay program
}
void main()
{
PINSEL2 = 0X00000000;          //  Set   P1.24 TO P1.31 as GPIO
IO1DIR = 0XFF000000;           //Port pins  P1.24 to P 1.31 Configured as Output port.

while(1)                       //Infinite loop
{
IO1SET=0XFF000000;               // Pins P1.24 to P1.31 goes to high
state delay();
IO1CLR=0XFF000000;               // Pins P1.24 to   P1.31 goes to low
state delay()  ;
}
}
```

Viva questions:

1. Why do we use C language for programming ARM microcontrollers?
2. How do you access ARM registers in C?
3. What is the use of const in embedded programming?
4. How do you configure a timer using C in LPC2148?
5. Write a simple LED blink code in C for ARM?

EXPERIMENT 15: ISR (Interrupt Service Routine) programming in ARM based systems with I/O port.

Aim: Write a program to program Interrupt in LPC2148 ARM7 Microcontroller.

Apparatus:

1. Computer
2. Keil µ vision 4

Procedure-1:

Program:

```
#include <lpc214x.h>
void initClocks(void);
void initTimer0(void);
__irq void timer0ISR(void);
int main(void)
{
initClocks(); // Initialize PLL to setup clocks
initTimer0(); // Initialize Timer0
IO0DIR = (1<<10); // Configure pin P0.10 as Output
IO0PIN = (1<<10);
T0TCR = (1<<0); // Enable timer
while(1); // Infinite Idle Loop
}
void initTimer0(void)
{
T0CTCR = 0x0; //Set Timer Mode
T0PR = 60000-1; //Increment T0TC at every 60000 clock cycles
//60000 clock cycles @60Mhz = 1 mS
T0MR0 = 500-1; //Zero Indexed Count-hence subtracting 1
T0MCR = (1<<0) | (1<<1);//Set bit0 & bit1 to Interrupt & Reset TC on MR0
VICVectAddr4 = (unsigned )timer0ISR; //Pointer Interrupt Function (ISR)
VICVectCntl4 = (1<<5) | 4; //(bit 5 = 1)->to enable Vectored IRQ slot //bit[4:0]) -> this the source
number
VICIntEnable = (1<<4); // Enable timer0 interrupt
T0TCR = (1<<1); // Reset Timer
}
__irq void timer0ISR(void)
{
```

```
long int readVal;
readVal = T0IR; // Read current IR value
IO0PIN ^= (1<<10); // Toggle LED at Pin P0.10
T0IR = readVal; // Write back to IR to clear Interrupt Flag
VICVectAddr = 0x0; // End of interrupt execution
}
void initClocks(void)
{
PLL0CON = 0x01; //Enable PLL
PLL0CFG = 0x24; //Multiplier and divider setup
PLL0FEED = 0xAA; //Feed sequence
PLL0FEED = 0x55;
while(!(PLL0STAT & 0x00000400)); //is locked?
PLL0CON = 0x03; //Connect PLL after PLL is locked
PLL0FEED = 0xAA; //Feed sequence
PLL0FEED = 0x55;
VPBDIV = 0x01; //PCLK is same as CCLK i.e.60 MHz
}
```

Procedure-2:

Overview: ISR with I/O Port on LPC2148

1. Configure a GPIO pin as **input** (e.g., button).
2. Enable an **External Interrupt (EINT)**.
3. Write the **ISR** (e.g., toggle an LED).
4. Clear the interrupt flag inside the ISR.

Step-by-Step LPC2148 Code (Keil C / Embedded C)

1. **Pin Configuration**

```
#include <lpc214x.h>

void GPIO_Init(void) {

    // Set P0.10 as output (LED)

    IO0DIR |= (1 << 10);



    // Set P0.14 as EINT1

    PINSEL0 &= ~(3 << 28); // Clear bits

    PINSEL0 |=  (2 << 28); // Select EINT1 function for P0.14

}
```

2. External Interrupt Configuration (EINT1)

```
void EINT1_Init(void) {

    // Configure EINT1 to trigger on falling edge

    EXTMODE |= (1 << 1);    // Edge sensitive

    EXTPOLAR &= ~(1 << 1);  // Falling edge

    // Enable EINT1 interrupt

    VICIntSelect &= ~(1 << 15); // EINT1 as IRQ

    VICIntEnable |=  (1 << 15); // Enable EINT1 interrupt

    VICVectCntl1 = 0x20 | 15;   // Use slot 1, enable, source #15 (EINT1)

    VICVectAddr1 = (unsigned)EINT1_ISR; // ISR address

}
```

3. ISR for EINT1

```
void EINT1_ISR(void) __irq {
    // Toggle LED on P0.10
    IO0PIN ^= (1 << 10);

    EXTINT |= (1 << 1);     // Clear EINT1 interrupt flag
    VICVectAddr = 0;        // Signal end of ISR to VIC
}
```

4. Main Function

```
int main(void) {
    GPIO_Init();
    EINT1_Init();

    while (1) {
        // Main loop does nothing, interrupt handles event
    }
}
```

Viva questions:

1. What are interrupts and exceptions?
2. Define Software interrupt vector?
3. What is a reset vector?
4. Define an ISR (Interrupt Service Routine)?
5. Describe how an ISR is used in an ARM-based system like the LPC2148, particularly when handling interrupts from I/O ports?

CONTENT BEYOND SYLLABUS

Additional Programs

Exercise : 1: GCD and LCM

Aim: Write a program to find the GCM and LCM of a given two bytes in a memory locations 50h,51h and store the GCM in 52h and LCM in 53h locations.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

ORG 0000H

; TO FIND THE GCD

MOV A, 50H ; TWO NUMBERS IN 50H AND 51H LOCATIONS MOV B, 51H

BACK: MOV R1,B DIV AB

MOV A, B

CJNE A, #00H, L1 L1: JZ LAST

MOV A, R1 SJMP BACK

LAST: MOV A, R1

MOV 52H, A ; GCD IN 52H LOCATION

; TO FIND THE LCM

MOV A, 50H MOV B, 51H MUL AB MOV B, 52H DIV AB

MOV 53H, A ; LCM IN 53H LOCATION END

Exercise : 2: BCD to SEVEN SEGMENT CODE

Aim: Write a program to convert the given BCD number into its equivalent seven segment value.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

```
        ORG 0000H
        MOV P1,
        #0FFH MOV
        P2, #00H

        MOV DPTR,
#0300H AGAIN: MOV
A, P1

        ANL A, #0FH
        MOVC
        A,@A+DPTR
        MOV P2, A

        SJMP
AGAIN ORG 0300H

        DB
        3FH,06H,5BH,4FH,66H,6DH,7DH,07H,7FH,6
        FH END
```

Exercise : 3: DELAY using Timers

Aim: Write a program to demonstrate the operation of timers in 8051 microcontroller to generate the delays using interrupts and without interrupts.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

1. Toggling of p1.5 with the delay of 5ms without interrupts using timer 1

```
ORG 0000H
MOV TMOD, #10h AGAIN: MOV TL1, #0FFH
MOV TH1, #0EDH CPL P1.5
SETB TR1 BACK: JNB TF1, BACK
CLR TR1 CLR TF1
SJMP AGAIN END
```

2. Toggling of p1.2 with the delay of 5ms with interrupts using timer 0

```
ORG 0 LJMP MAIN
ORG 000BH; ISR for Timer 0 CPL P1.2
MOV TL0, #0FFH MOV TH0, #0EDH RETI
ORG 30H
MAIN: MOV TMOD, #01h; Timer 0, Mode 1 MOV TL0, #00
MOV TH0, #0DCH
MOV IE, #82H; enable Timer 0 interrupt SETB TR0
HERE: SJMP HERE END
```

Exercise : 4: Counters with/out Interrupts

Aim: Write a program to demonstrate the operation of counters in 8051 microcontroller to count the no. of pulses as input to pin P3.4 using interrupts and without interrupts.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

1. Count no. 0f pulses without interrupts using counter 0

ORG 0000H

MOV TMOD, #05h SETB P3.4

MOV TL0, #00H MOV TH0, #00H SETB TR0

BACK: MOV A, TL0

MOV P1, A MOV A, TH0 MOV P2, A SJMP BACK END

2. Count no. 0f pulses without interrupts using counter 1

ORG 0000H LJMP MAIN ORG 001BH MOV P1, #55H

MOV TL1, #0FAH MOV TH1, #0FFH RETI

ORG 30H

MAIN: MOV TMOD, #50h SETB P3.5

MOV TL1, #0FAH MOV TH1, #0FFH MOV IE, #88H SETB TR1

HERE: MOV P1, #0AAH

SJMP HERE END

Exercise : 5: Serial Communication

Aim: Write a program to demonstrate the operation of UART in 8051 microcontroller by sending and receiving the characters serially.

Apparatus:

1. Computer
2. Keil µ vision 4

Program:

1. Program to transfer data serially with baud rate of 9600

MOV TMOD, #20H MOV TH1, #-3 MOV SCON, #50H SETB TR1

AGAIN: MOV SBUF, #"A" HERE: JNB TI, HERE

CLR TI

SJMP AGAIN END

2. Program to receive data serially with baud rate of 9600

ORG 0000h

MOV TMOD, #20H MOV TH1, #-3 MOV SCON, #50H SETB TR1

HERE: JNB RI, HERE

MOV A, SBUF MOV P1, A CLR RI

SJMP HERE END

Exercise : 6: DEMONSTRATION OF LOAD/STORE AND BRANCH INSTRUCTIONS

Aim: Write a program to load/store and branch instructions of any data to/from registers.

Apparatus:

1. Computer
2. Keil µ vision 4

Hints to write Program:

A: *This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.*

```
B forward
ADD r1, r2,
#4 ADD r0,
r6, #2 ADD
r3, r7, #4
forward
SUB r1, r2, #4
backward
ADD r1, r2,
#4 SUB r1,
r2, #4 ADD
r4, r6, r7
B backward
```

B: *Examples of LDR instructions using different addressing modes.*

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

C: *Variations of STRH instructions.*

	Instruction	Result	*r1* +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Viva questions:

1. How is a subroutine call implemented in ARM?
2. How is the return address managed in ARM branch instructions?
3. How can you perform a jump table using load and branch instructions?
4. How do you write a while or for loop equivalent using ARM branching instructions?
5. What is the purpose of BX instruction? How is it used for switching modes (ARM/Thumb)?

Exercise : 7: DEMONSTRATION OF LOAD/STORE AND BRANCH INSTRUCTIONS

Aim: Write a program to load/store and branch instructions of any data to/from registers.

Apparatus:

3. Computer
4. Keil µ vision 4

Hints to write Program:

A: *This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.*

```
    B forward
    ADD r1, r2,
    #4 ADD r0,
    r6, #2 ADD
    r3, r7, #4
    forward
    SUB r1, r2, #4
    backward
    ADD r1, r2,
    #4 SUB r1,
    r2, #4 ADD
    r4, r6, r7
    B backward
```

B: *Examples of LDR instructions using different addressing modes.*

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

C: *Variations of STRH instructions.*

	Instruction	Result	*r1* +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Viva questions:

1. How is a subroutine call implemented in ARM?
2. How is the return address managed in ARM branch instructions?
3. How can you perform a jump table using load and branch instructions?
4. How do you write a while or for loop equivalent using ARM branching instructions?
5. What is the purpose of BX instruction? How is it used for switching modes (ARM/Thumb)?

Exercise : 8: PROGRAMS USING ARITHMETIC AND LOGICAL INSTRUCTIONS

Aim: Write a program to perform the arithmetic and logical operations for the given data.

Apparatus:

1. Computer
2. Keil µ vision 4

Hints to write Program:

A: Arithmetic instructions

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

Example 1: This simple subtract instruction subtracts a value stored in register r2 from a value stored in register r1. The result is stored in register r0.

PRE r0 =

0x00000000 r1 =

0x00000002

r2 =

0x00000001

SUB r0, r1, r2

POST r0 = 0x00000001

Example 2: This reverse subtract instruction (RSB) subtracts r1 from the constant value #0, writing the result to r0. You can use this instruction to negate numbers.

PRE r0 =

0x00000000 r1 =

0x00000077

RSB r0, r1, #0 ; Rd = 0x0 - r1

POST r0 = -r1 = 0xffffff89

B: Logical instructions

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example : This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

PRE r1 =

0b1111 r2 =

0b0101

BIC r0, r1, r2

POST r0 = **0b1010**

This is equivalent

to Rd = Rn **AND**

NOT (N)

Viva questions:

1. What's the difference between ADD and ADDS?
2. What is the use of CMP and how is it different from SUB?
3. How does the ORR instruction work?
4. What's the difference between MOV and MVN?
5. Which logical instructions update the CPSR flags?