ch3. Instruction Set.

- Addressing modes of 8051.

- Register   MOV A, R5
- Direct   MOV A, 42H $A \Leftarrow mem(42H)$
- Immediate   MOV A, #42
- Relative   SJMP 42H

  $PC \Leftarrow PC + 2 + 42H$

- Indirect

  Internal : MOV A, @ R0 → note that only R0 & R1 can be used.

  $A \Leftarrow mem(R0)$

  External : MOVX A, @DPTR

  $A \Leftarrow XMEM(DPTR)$

- Long   LJMP 14A0H → 16 bit address

  $PC \Leftarrow 14A0H$.

- Indexed : go to an address plus an index

  JMP @A + DPTR

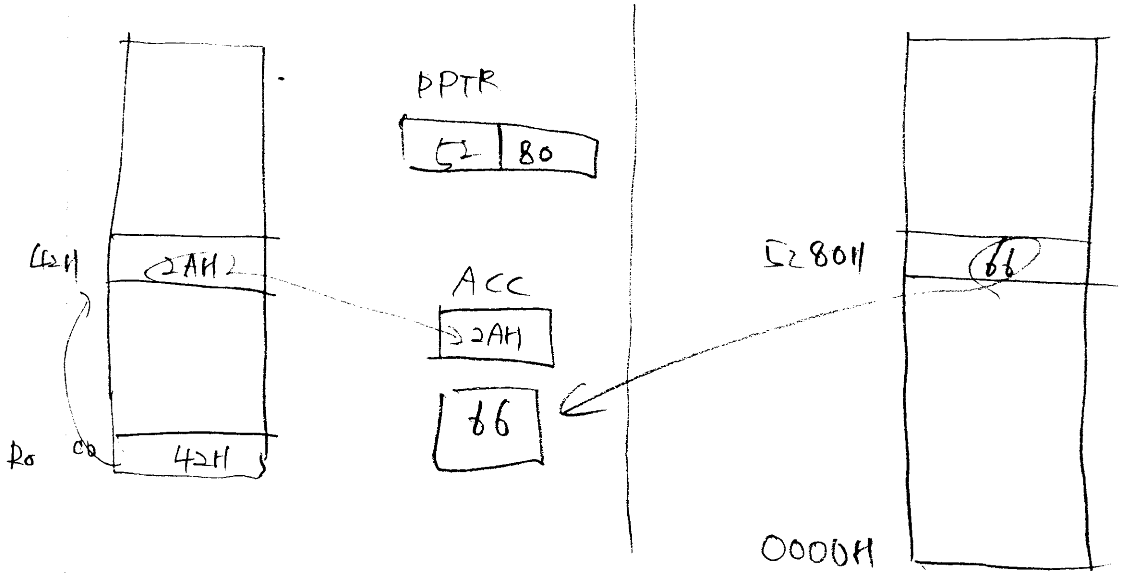
  A: 8-bit index, DPTR: 16bit base addr

  $PC \Leftarrow A + DPTR$

ex) Indirect addressing

MOV A, @R0

MOVX A, @DPTR

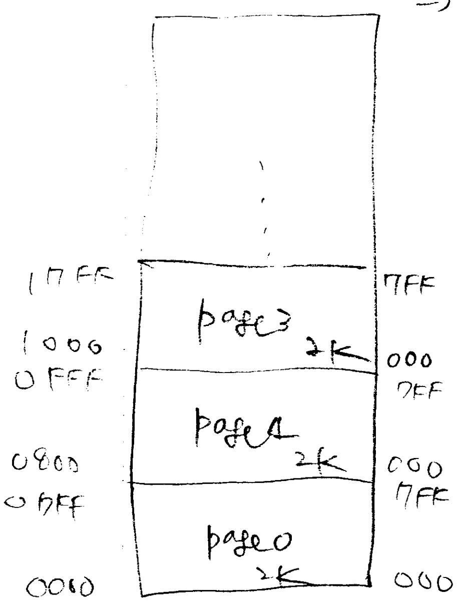


- Absolute addressing

64K code mem has 32 pages of 2K mem

=> upper 5 addr bits => page

lower 11 " => address within the page.



See page 250 4251

AJMP 042H, → 11 bit address.

Encoding: 000 0 0001 0100 0010

AJMP

11 bit addr

PC	15 ... 11	10 ... 0
	unchanged	000 0100 0010

indicates page ex) 00000 page0.

00001 page1

⋮

HW #3

3, 10, 12, 18, 19, 26, 34

3. SETB 28H (from fig 2-6, PPT)

10,
- a) 31H, 32H, 35H (26H = 0010 0110)
- b) 31H, 33H, 34 ~ 36H (7AH = 0111 1010)
- c) E0H, E1H, E4H, D0H (13H = 0001 0011) → $\boxed{\text{P bit set}}$
- d) ~~78H ~ 7FH~~ 30H is not bit addressible!
- e) 91H
- f) B2H, B3H (0CH = 0000 1100)

12.

```
MOV  A, #0ABH
MOV  DPTR, #9A00H
MOV  @DPTR, A
```

18.

		RS1 RS0
	a) 3	1111 1101
	b) 3	0001 1000
	c) 1	0000 1000

19,

	a) 1	1100 1000
	b) 2	0101 0000
	c) 2	0001 0000

26. $\overline{PSEN}$ selects ext EPROM.
$\overline{WR}$ and $\overline{RD}$ select external RAMS.

34. a) A = 0000 0000 (P = 0)

b) A = 0000 0011 (P = 0)

c) A = 1010 1011 (P = 1)

#

✪ 8051 instructions

① Quick ref. chart for 8051 insts is shown in Appendix A.

- Inst written in assembly language is called "Mnemonic"

ex) $\overbrace{\underbrace{\text{ADD}}_{\text{opcode}} \quad \underbrace{\text{A, }R_0}_{\text{Operands.}}}^{\text{Mnemonic}}$

✪ Inst code summary is shown Appendix B.

- Opcodes
- # bytes required for inst
- # cycles " inst.

ex) what inst has opcode 24H?

⇒ ADD A, #data.

→ 2 bytes required. $\underbrace{0010 \quad 0100}_{\text{opcode}} \quad \underbrace{dddd \quad dddd}_{\text{immediate data. (operand).}}$

→ 1 cycle required to execute.
1 = 6 states.
1 state = 2 clock cycles
⇒ 1 machine cycle = 12 clock cycles)

✪ Inst definitions are shown in Appendix C ex) what is opcode for

CJNE R5, #42H, 2AH (page 255)

10111 rrr dddd dddd eeee eeee

101 0100 0010 0010 1010

ex) ADD A, #42H (page 249)

0010 0101 0100 0010

☆ 8051 inst types

1. Arithmetic
2. logical
3. Data transfer
4. Boolean variable
5. Program branching

see appendix A for quick ref. chart.

☆ Arithmatic insts.

ex) Illustrate an inst sequence to subtract the content of R6 from R7 and leave the result in R7.

```
MOV   A, R7
CLR   C
SUBB  A, R6     → subtract R6 from A with borrow. (Carrybit can be borrowed).
MOV   R7, A
```

ex) decrement DPTR by 1

⇒ Since DPTR is 16-bit reg, DPL & DPH must be individually considered. ⇒ DPH is decremented only if DPL underflows from 00H to FFH.

```
      DEC   DPL              ; decrement low-byte by one.
      MOV   R7, DPL
      CJNE  R7, #0FFH, SKIP  ; if underflow
      DEC   DPH              ; decrement high-byte by one
SKIP: (continue).
```

① if DPTR = 0100H

→ DPL becomes FFH
DPH " 00H) $\boxed{\text{00FFH}}$

② if DPTR = 0001H

→ 0000H.

⊗ MUL AB ; multiply A and B., then store the 16 bit product into B (high byte) and A (low byte).

ex) A = 55H , B = 22H.

MUL AB ?

solution: A = 4AH , B = 0BH

$85_{10} \times 34_{10} = 2890_{10} = 0B4AH.$

\# BCD arithmatic : ADD & ADDC must be followed by a DA A (decimal adjust)

A = 59H (BCD value 59)

ADD A, #1 ⇐ A = 5AH

DA A ⇐ A = 60H

ex) Illustrate how to add two 4-digit BCD #s.
The first #s in internal mem location 40H & 41H,
" second " 42H & 43H.
The most significant digits are in locs 40H & 42H.
Place the BCD result in locs 40H & 41H.

40H	05H	→ 11H
41H	35H	→ 80H
42H	06H	
43H	45H	

```
MOV   A, 43H      A = 45H
ADD   A, 44H      A = 7AH
DA    A           A = 80H
MOV   41H, A
MOV   A, 42H      A = 06H
ADDC  A, 40H      A = 0BH
DA    A           A = 11H
MOV   40H, A
```

⊛ Logic insts. (AND, OR, XOR, NOT ···)

ex) A = 0011 0101

```
ANL  A, # 0101 0011B
```

```
      0011 0101
and   0101 0011
---------------
A =   0001 0001
```

ex) XRL P1, #0FFH.

⇒ Quick & easy way to invert port bits

```
P1 =  1010   0101
 XOR  1111   1111     } inverted.
-------------------
P1 =  0101   1010
```

- Rotate insts (RL A, RR A)
  - Shift the Acc one bit to the left/right.

    For a left rotation, MSB rolls into LSB.

    " right " , LSB " MSB

ex) A = 1001 0010

RL A

A = 0 1001 001

- 9-bit rotation insts (RLC A, RRC A)
  - Carry flag is considered as the 9th bit.

ex) C = 1 A = 00H

RRC A

C = Ø A = 80H

- SWAP A - exchanges the high & low nibbles in Acc. ⇒ useful in BCD manipulations

ex) Acc has a bin number that is known to be less than 100_{10}. It is quickly converted to BCD as follows.

	MOV B, #10	$B = 89_{10}$
(A) ← Quotient of (A)/(B) (B) ← Remainder of (A)/(B)	DIV AB	A = 08H, B = 09H,
	SWAP A	A = 80H
	ADD A, B	A = 89H.
		$8\ 9_{10}$ in BCD.

★ Data transfer instrs.

MOV <dest> , <src>

direct
indiref
Reg

direct indirect immediate , reg
ex) 2AH , @R0 , #2AH , R6.

XCH A , <byte - variable> : exchage data.
dirct, indirect , reg.

ex) A = 01H R0 = 02H
XCH A , R0
⇒ A = 02H R0 = 01H

XCHD exchanges... low - order nibbles...

★ Internal stack.

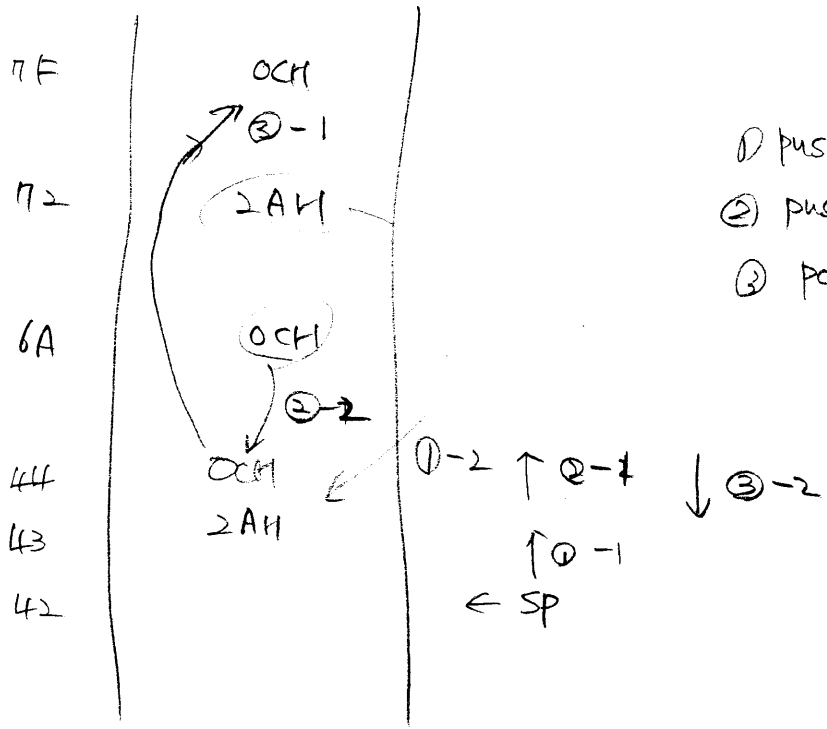
① First - in , last - out data structure.

② SP (stack pointer) is used to indicate the stack top.
→ direct address 81H

③ PUSH <direct - addr> inst increment SP by 1 , then pushes Content of mem location <direct - addr> onto stack

④ POP <direct - addr> pop & store the data to mem location <direct - addr> & increment SP by 1.

ex) Mem



① push 72H
② push 6AH
③ pop 7FH

★ External mem.

MOVX <dest> , <src>
  " A , @Ri
  " A , @DPTR
  " @Ri , A
  " @DPTR , A.

MOVC A , @A + DPTR A ← code mem (A + DPTR)
  " @A + PC A ← A + PC

⇒ used to load constants stored in ROM.

~~* Boolean insts (see Appendix A, C)~~

~~ANL ex) ANL A, #01H~~

~~ORL ANL C, /P1.0~~

~~XRL~~

* Branching

- Unconditional

  JMP @A + DPTR ; jump indirect.

```
ex)       MOV A, #04H
          MOV DPTR, # JMP_TBL
          JMP @A + DPTR
JMP_TBL:  <inst 0>
          <inst 1>
          <inst 2>     <- Jumps to <inst 2>
          <inst 3>
```

suppose 2 byte insts.

- Function calls

  ACALL addr11 — call function at addr.

  LCALL addr16.

  a) increment PC by 2 (ACALL) or 3 (LCALL)

  b) push PCL ⇒ will be used as return address.

     push PCH

  c) jump addr.

RET return from function

a) pop PCH

   pop PCL

b) jump PC (return addr)

- Conditional branch
  - JZ rel — jump if A = 0
  - JNZ rel — 〃 A ≠ 0
  - CJNE <byte1>, <byte2>, rel — 〃 byte1 ≠ byte2.
    - <byte1>: A, Rn, @Ri
    - <byte2>: Direct, Immediate
  - DJNZ <byte>, rel — decrement <byte> and if not zero, jump.
    - <byte>: Rn, direct

ex) DJNZ R6, 22H

R6	PC
42H	5280H
↓	
41H	5280+2+22 = 52A4H
not zero	→ jump rel

☆ logical insts.

ANL <dest>, <src> — bitwise logical AND.

ex)
MOV A, #42H
ANL A, #65H

$$\begin{array}{r} 0100\ 0010 \\ 0110\ 0101 \\ \hline 0100\ 0000 \end{array}$$

A = 40H.

ORL — bitwise OR

XRL — 〃 XOR

CLR A — clear A

CPL A — complement A.

ex)
MOV A, #42H — A = 0100 0010
CPL A — A = 1011 1101.

- Bit insts.

Recall: 00 ~ 7F bit mem = 20 ~ 2F bytemem.

SFRs ending w/ 0 or 8 ⇒ bit addressible.

ANL C, bit $\quad (C) \leftarrow (C) \cdot (bit)$

direct.

" /bit $\quad$ " $\cdot (\overline{bit})$

ORL C, bit

C, /bit

MOV C, bit or bit, C

CLR bit

SETB bit

CPL C $\quad (C) \leftarrow (\bar{C})$

Quiz #3 (30pts) 3-6-03.

Implement 8051 assembly pgm to reverse the bits in the Acc.

(hint: (use a loop rotation insts.)

ex) Reverse the bits in the Acc.

```
1          MOV    R7, #8
2   LOOP:  RLC    A
3          XCH    A, 0F0H      → any G.P. Register will do.
4          RRC    A
5          XCH    A, 0F0H
6          DJNZ   R7, LOOP     → decreament & Jump.
7          XCH    A, 0F0H
```

→ B register is a direct addr 0F0H.

~~Skip~~ Quiz solution

C	A	R7	B	
0	1011 0101	8_{10}	0000 0000	1
1	0110 1010			2
	0000 0000		0110 1010	3
0	1000 0000			4
	0110 1010		1000 0000	5
		7_{10}		6
0	1101 0100			2
	1000 0000		1101 0100	3
0	0100 0000			4
	1101 0100		0100 0000	
	⋮			
			1010 1101	5
				6
	1010 1101			7

Ch 7. 8051 ASM language programming

- Machine instructions you've been learning show basic capabilities of 8051.

- Assembly is language built around these basic instructions with a few "extras" to make the programmer's life easier.

- Note that particular version of ASM we will be using is slightly different from books.

☆. General format for each line.

[label:] mnemonic [operand] [, operand] [...] [; comment]

mnemonic → mandatory

optional

☆ Assembler directives

☆ translates ASM code → machine code

→ insts to the assembler program.

→ are not assembly language insts executable by the target μC or μP.

Segment selection directives

Def: a defined block of memory

Two types

1. Absolute segments – each absolute segment is located at specific memory location

	directives	operand (absolute addr)
Code	CSEG	[AT address]
direct internal data	DSEG	"
indirect internal data	ISEG	"
bit-addressible mem	BSEG	"
external data mem	XSEG	"

Ex) CSEG AT 0000H

⇒ Code segment starts at memory location 0000H.

2. Relocatable segments – assembler decides location.

⇒ generally BEST choice!

RSEG segment-name

the name of a relocatable segment previously defined with the SEGMENT directive

<u>syntex</u>

symbal SEGMENT segment-type

CODE - code
XDATA - ext data
Const - constants stored in code mem + DATA - direct internal data
IDATA - indirect "
BIT - bit addressible mem space.

name of the segment

ex) mydata SEGMENT DATA → direct internal data segment

RSEG mydata ; start relocatable segment.

☆ Labels (including variables)

- Variables declared in segment

ex)

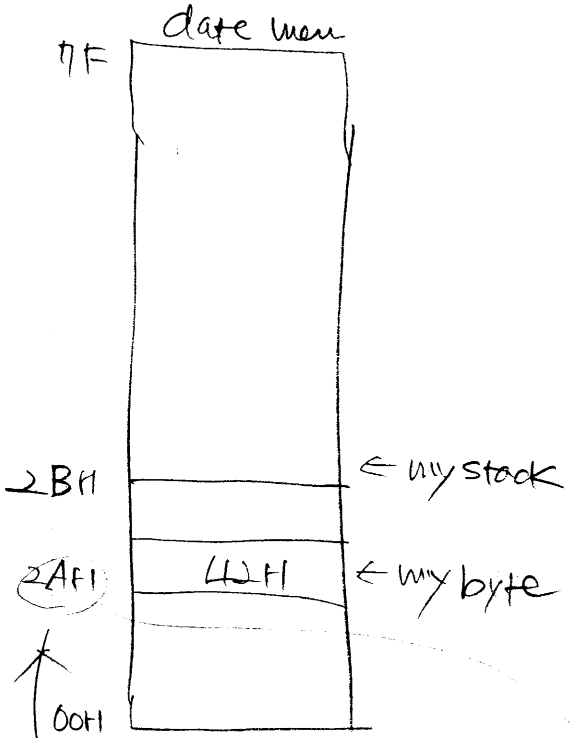
; segment name, type, start.

mybyte: DS 1.

name (label) — define storage (byte) — # of bytes.

In program....

MOV mybyte, #42H



whatever assembler places.

→ Assembler codes

MOV mybyte, #42H

as

MOV 2AH, #42H.

What about.

① MOV A, my Stack+1 → 2BH+1 = 2CH

→ MOV A, 2CH

② MOV R0, # my byte → address of mydata

R0 = 2AH.

MOV @R0, #42H.

* Variables declared outside of segments.

⇒ use DATA, IDATA, CODE, XDATA and BIT directly!

ex) another (name of variable) DATA (seg) 7FH (address location)

- Constants (read-only data in code mem)

CSEG AT 0100H.

ex) lookup: DB 1,2,3,4,5,42

name

define byte

Values in mem.

addr	Contents	
0100	1	lookup
0101	2	lookup+1
0102	3	" +2
0103	4	" +3
0104	5	" +4
0105	42	" +5

DW - define word. (2 bytes)

ex) CSEG AT 200H

DW $, 'A', 1234H, 2, 'BC'

current mem location

0200	02	word.
0201	00	
0202	00	ascii code for 'A'
0203	41	
0204	12	
0205	34	

0206 00
0207 02
0208 42 → 'B'
0209 43 'C'

exam

* More assembler directives
  ↳ commends to assembler

- EXTERN ~ declare variables from other modules (files)
- PUBLIC ~ declare variable to be used elsewhere (in other module)
- END ~ the last statement in the source file

ex) Main.src

```
Extern    Code (HELLO, GOOD-BYE)
...
CALL      HELLO
...
CALL      GOOD-BYE
...
END
```

Messages.src

```
          public    HELLO, GOOD-BYE
HELLO:    ...
          RET
GOOD_BYE: ...
          RET
          ...
          END
```

- EQU ~ Create "assembler" constant
  (like #define in C)

ex) TheAssembler EQU #42H

⋮

MOV A, TheAssembler ⇒ Assembler replaces TheAssembler with 42H.

⇒ MOV A, #42H

★ immediate data

MOV A, #2AH → Hex #.
MOV A, #42D → dec #
MOV A, #0010 1010B → Bin #.
MOV A, #42 → dec # is default.
MOV A, #41 +1

same!

★ SFR.

SETB D7H → Bit addr.
SETB PSW.7 → reg name, 7th bit.
SETB D0H.7 → Byte addr, 7th bit
SETB CY → bit register name.

same! (see page 25).

⇒ SFRs can generally be referenced by name.

⊛ General program layout

A. data segment
  a) declare segs
  b) declare variables

B Code segment
  a) declare segcs)
  b) write constants

⊛ Go through test.a51 using μvision2

① Create a project.
Select device (generic 8051)
② Create an assembler source file & add it to the project.
③ Set tool options (Create hex file)
→ for PROM programmer
④ Build the project & Create a hex file.
⑤ Simulate the application with the debugger.
open P1 & P3   F11 to step.   → breakpoint   F5 BP.
View /disassembly.   double click on a line → F5
insert MOV C,acc.5 , MOV P3.0, C , CLR..

```
; example

        cseg at 0
        ljmp start

table:  db 0,1,4,9,16,25,36,49


        cseg at 100h
start:  mov p1,#0e0h                    ; make bits 5 to 7 inputs

loop:   mov a,p1            ; read p1

        mov r2,#5           ; shift right 5x (acc>>5)
rotlp:  rr a
        djnz r2,rotlp

        anl a,#7            ; mask input bits
        mov dptr,#table                 ; get table address
        movc a,@a+dptr                  ; get ith entry in table

        mov c,acc.5
        mov p3.0,c
        orl a,#0e0h                     ; make high order bits = 1
        mov p1,a            ; output to p1
        sjmp loop           ; repeat forever

        end
```

CpE 213
Example ISM78 – Assembly language programming with µVision2

Purpose

This is a brief overview of how to use the Keil µVision2 software to write and debug simple assembly language programs. Only short absolute assembly programs are discussed. The full capability of the A51 macro assembler is not used.

Description

This example will make use of a small 8051 assembly language program that uses table lookup to translate between a three bit code input through the 8051's P1 and a five bit code to be output on the remaining 5 bits of P1. Specifically, the program specification is to read a three bit code on P1(7 downto 5) and output the square of the code on P1(4 downto 0).

We can read the code with a MOV A,P1 instruction and then shift A right five times to put the code in the least significant bits of A. The instruction ANL A,#7 will set the unused 5 bits to zero. Then we can use the MOVC A,@A+DPTR instruction to do the table lookup. Output the code from the table with the instruction MOV P1,A, then jump back to the beginning to start over. We can use the assembler's DB psuedo-op to construct the table.

The process for using µvision2 to create an application for the 8051 is outlined on page 47 of the Getting Started Guide (gs51.pdf). This process includes:

a) create a project file and select a cpu,
b) create an assembler source file and add it to the project,
c) set tool options for the target hardware,
d) build the project and create a hex file,
e) simulate the application with the debugger.

You can find gs51.pdf in the keil/c51/hlp directory or by clicking on the books tab in the project window of µvision2. The project window is the middle left window and the books tab is the rightmost bottom tab. Chapter 3 p39, chapter 4 p47-53, and chapter 5 of gs51 should be read before you try doing much with µvision2. This little handout will help you get started but won't tell you all there is to know about µvision2!

Create a project

Start µvision2 from the Start menu. When µvision starts, close any active project with the Project/Close menu item and create a new project with the Project/New menu item. A dialog box like that shown in Figure 1 will be displayed. Use the directory list box and navigation icons to switch to a directory where you have write access. The list in figure 1 shows a subdirectory (cpe213) with several projects (*.uv2) listed. You are advised to organize your subdirectories systematically and not put all files into a single large directory although the getting started guide's recomendation of a folder per project seems a bit extreme. Use a meaningful name for the project since the default name of the executable is the name of the project file.

After you save the project file, you'll be presented with a dialog box to select the target processor like that shown in Figure 2. The example shows a generic 8051 selected. When you select a target device, a short synopsis of the device's features is displayed. It is instructive to browse through a few of the processors in the µvision database. See if you can find one with two DPTR's.

Create an assembler source file and add it to the project

new file icon

Next, click on the new file icon . This will bring up a text editor window where you can enter your assembler program source code.

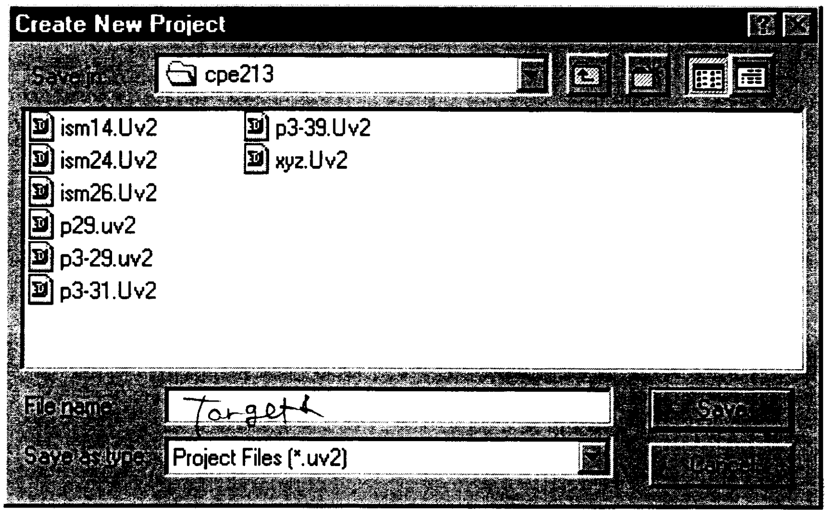


Figure 1 New Project dialog box

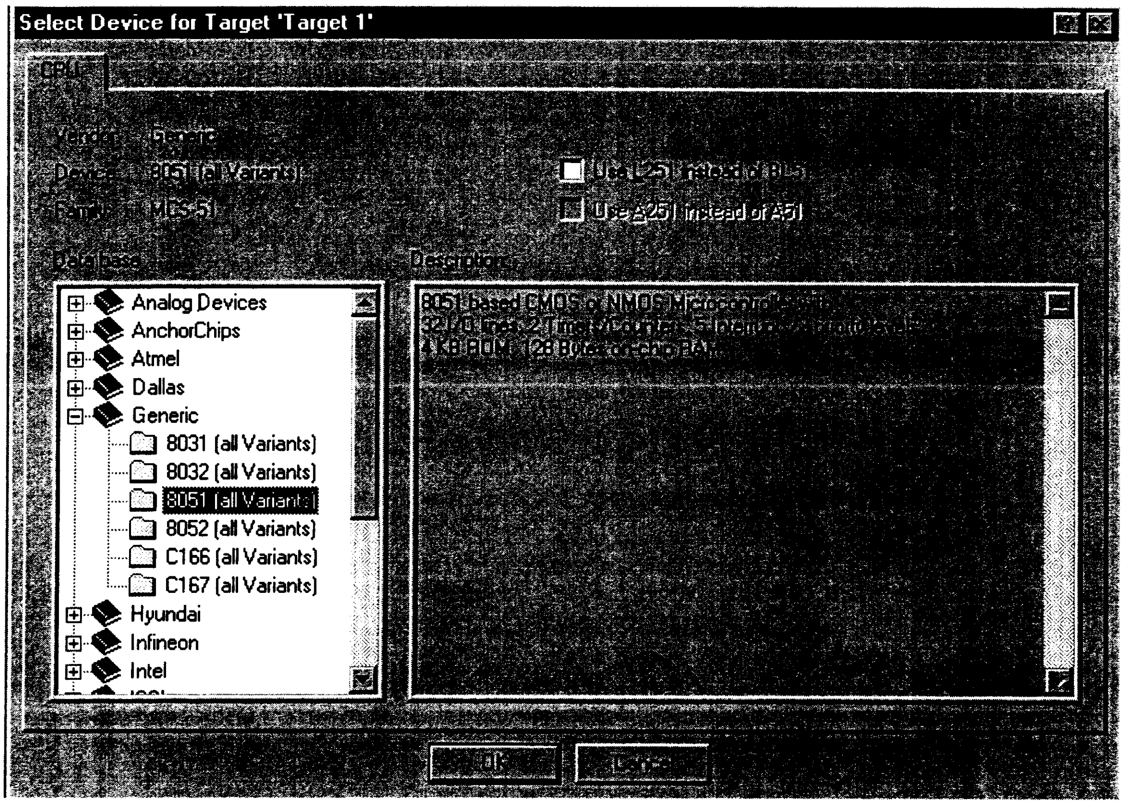


Figure 2 Select device dialog box

After you enter your source code, save it using the File/Save As menu item. Save it with a meaningful filename and an 'a51' extension (eg p39.a51) in the same directory as your project file. After you save the file, you can add it to your project. Expand target 1 in the project window by clicking on the '-' box to get a source group 1 entry. Right click on source group 1 and select the 'add files' popup menu item. Select your new a51 source file, click on 'add', then close. Your source file should look something like that in Figure 3.

The first line in this file is a comment started with a semicolon. Line 2 is a cseg psuedo-op that defines a code segment starting at location 0. This is something like the debugger's 'asm 0' command. The first instruction is a long jump (ljmp) to location 'start'. Since all 8051's start up at location 0 after power on, this will take the processor to the first instruction of our program. The next line is a table created with the DB psuedo-op. The table starts at symbolic location 'table' and consists of a string of 8 bytes which are the squares of the first 8 integers (0 through 7). These three lines will result in 11 bytes in locations 0 through 10 of the 8051's code memory. The first three bytes will be 20h, 01h, and 00h which are the machine code for the ljmp instruction. Note that the table MUST be located in code memory. Data memory is sram whose contents are lost after power is turned off. Code memory is non-volatile and is the only place where we can keep initialized data. If we insist on putting initialized data in sram, then we will need to add code to initialize it from code memory. We will never change 'table' so it's best to just keep it in code memory.

S:\CPE213\p3-39.a51*

```
; example for a p39-like problem

        cseg at 0
        ljmp start

table:  db 0,1,4,9,16,25,36,49

        cseg at 100h
start:  mov P1,#0e0h     ; make bits 5 to 7 inputs

loop:   mov a,p1         ; read p1

        mov r2,#5        ; shift right 5x (Acc>>5)
rotlp:  rr a
        djnz r2,rotlp

        anl a,#7         ; mask input bits
        mov dptr,#table  ; get table address
        movc a,@a+dptr   ; get ith entry in table

        orl a,#0e0h      ; make high order bits = 1
        mov p1,a         ; output to p1
        sjmp loop        ; repeat forever

        end
```

Possible. Exam Question

From A drive. test.a51

Figure 3 Example assembler source file

The second cseg is there to move the location counter above the interupt vectors which occupy lower code space. We don't really need it this time but it's a good habit to get into. The symbol 'start' defines the location of the mov instruction which is the target of the initial ljmp. We could have simply said ljmp 100h but that isn't good practice. It's better to use symbols than bare constants.

The mov P1,#0e0h instruction makes the upper three bits of port 1 inputs. Recall that writing a '0' to an 8051 port bit turns on its pulldown and writing a '1' turns the pulldown off. We want the external device (a switch pulldown and resistor pullup combination) to determine whether the port bit is a '1' or a '0' so this instruction is there just to make sure the port bit is acting as an input and not an output. The other 5 bits are set to 0's. This is arbitrary in this example. In a particular application we may want to set them to 1's as well.

The next instruction (labeled 'loop') is the start of a large loop that reads the 3 bit input, shifts it to the least significant bits of the accumulator, does a table lookup to calculate the square of the 3 bit number, and outputs the resulting 5 bit number on the other 5 bits of port 1. The next three instructions shift the accumulator right 5 bits. Notice that this takes 16 cycles to execute and requires 5 bytes of code. Five rr instructions would also require five bytes of code and run in only 5 cycles. This program has room for improvement!

Once the 3 bit input number is in the least significant bits, we clear the other 5 bits to 0's with the anl, put the address of the lookup table into dptr, and then use an indexed movc instruction to load the square of the 3 bit number into the accumulator. Finally we set the 3 most significant bits to 1's so we don't turn our input port into an output port, output the square to P1 (keeping the upper three bits as inputs), and repeat the whole process.

Select tool options

Before building the project, right click on the target 1 box and select 'select tool options' from the popup menu. You'll get a dialog box like that in Figure 4. Select the 'debug' tab and make sure the defaults are set like that in figure 4. We will be using the simulator and we want the application loaded when the simulator starts. Under the 'output' tab you can select 'create hex file'. It's not required but will give you a hex file to look at. Microvision's simulator uses the linker output. The hardware simulator we use in CpE 214 uses the hex file. PROM programmers also use the hex file format.

Build the project application

Build icon

Click on the build target icon to assemble the source file and create an object file. If there are any errors, fix them and rebuild the target. If there aren't any errors (there shouldn't be), it's instructive to create one just to see what happens. Try taking the last 't' off one of the 'start' symbols and rebuild.

Debug the application

debug icon

Click on the debug icon to start up the debugger with your application. The project window will switch to the register tab and display the 8051's registers and the text window with your code will display a yellow arrow that points to the next instruction to be executed (the ljmp). Open up the P1 window by selecting the Peripherals/IO Ports/Port 1 menu item. Press F11 twice and notice what happens: the location arrow moves to 'loop' and the two instructions that are executed get marked with a green mark. These 'code coverage' marks show how much of your program has been executed and are an aid in testing complex programs. When the mov instruction is executed, the Port 1 window reveals that P1 bits 5, 6, and 7 are set and the remaining bits are reset. Notice that there is a row of 'pins' bits and a row for P1. The P1 bits correspond to the port's flip flops that are set/reset by the mov instruction, while the 'pins' bits reflect the actual state of the pins. Try clicking on pin 0 and see what happens. If you set a real processor's P1.0 to 0 and then tried to force the P1.0 pin high you wouldn't get a warning message. Most likely what would happen is that the port bit driver would quietly die and you'd be left with a damaged microcontroller. You can click on P1.0 (the top row) to set it and this will have exactly the same effect as a setb P1.0 instruction. You can do this to any of the registers and memory locations if you need to during debugging. Now press the reset icon to reposition the location arrow back at location 0.

reset icon

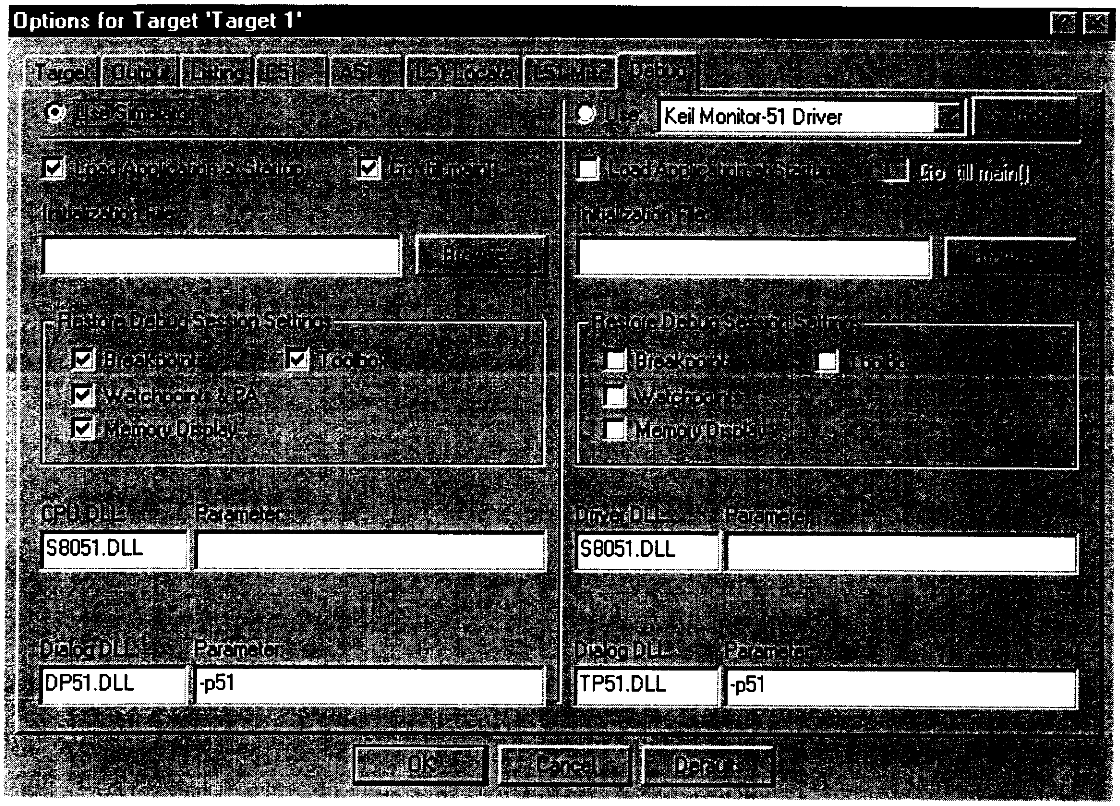


Figure 4 Target options dialog box.

Now select the View/Disassembly window to bring up a window that shows the assembly code along with the machine code. Notice that location c:0 contains a 02h, c:1 contains 01h, c:2 contains 00h and so on. Notice also that the 8 bytes in 'table' have also been dutifully disassembled as if they were instructions. The disassembler doesn't know any better. Adjust the register window so that the accumulator is visible. Press F11 until the mov instruction at location 103 is executed. Notice that the A register changes to an E0h and is colored green to highlight the fact that it has changed. Registers that change as a result of instruction execution get highlighted this way.

We could just keep hitting F11 and execute one instruction at a time but that is a hard way to debug a million lines of code. Breakpoints are better. Double click on the sjmp instruction in the disassembly window and notice that a red dot appears in the left margin. This is a breakpoint. If we now run the program, it will stop when it hits the breakpoint. Go ahead and hit F5 (run) and notice that the yellow location arrow stops at the breakpoint. Notice also that a is now equal to F1. Is this correct? Unless you changed P1 you should have read an input of 7 (upper 3 bits are one by default), read a 49 from the table (31h), and tried to output it. Unfortunately 31h is one bit too large for 5 bits. We need 6 bits. We now have a dilemma. We need to read 3 bits and output 6 but our ports are only 8 bits. There are many ways to solve this and which way is 'best' depends a lot on other factors that haven't been specified. For example, let's assume that we've already fabricated printed circuit boards and reassigning the 3 input bits to a different port is out of the question. If a spare port bit isn't available we are really in hot water and probably should start looking for another job. Let's assume that P3.0 is free and use it to output the missing 6^{th} bit in our output code. We can do that by moving ACC.5 to P3.0 before the ORL is executed.

Click the debug icon to go back to the editor mode and add the two instructions: mov c,acc.5 and mov P3.0,c to your source code just before the ORL instruction. Build the application as before and click debug to get back into debug mode. Now open up both P1 and P3 port windows.

stop icon

Instead of stepping through using F11 or F5 and breakpoints, this time we'll just run the program. Select the View/Periodic Window Update item from the menu. This will periodically update the port windows as the program runs. With both P1 and P3 visible, hit the F5 key to run continuously. Now try entering different bit patterns into P1 bits 5, 6, and 7. Be sure to click on the pins check boxes and not the port boxes. Clearing the check box forces the pin low and will result in the mov a,p1 instruction reading a 0 in that bit position. Checking the box causes a '1' to be read. As you enter each number, make sure that the 5 low order bits of P1 and P3.0 display the square of the 3 bit number in the P1 input field. If you got that far, click on the stop icon to stop execution. Click on the reset icon to reset the location pointer back to 0.

Click on the debug icon to leave the debugger and get back to the editor window. Click on the open file icon (or select File/Open from the menu) and open the LST file for your source code. You will need to change the 'files of type' list box to display LST files first. Notice that the listing file is date stamped and displays the A51 command line used to invoke the assembler. In this case you should see the name of your source file, the 'small memory model' option, and some other options. To see what these options mean, click on the 'books' tab in the project window, and double click on the A51 User's Guide which will open the user's guide PDF in Adobe Acroread. Go to Appendix C in that manual for a list of controls.

The source code with line numbers and machine code added is displayed next, followed by the symbol table. This is a list of the symbols in your program along with their values. For example, ACC is a data type address with value E0 absolute. The complete format of the listing file is documented in Appendix F of the user's guide. Chapter 1 of this manual gives a nice overview of assembly language programming.

Summary

This example has shown:

- How to use µvsion2 to create an assembly language application
- How to use the µvision2 debugger to test an assembly language application

This has been a brief introduction to assembly language application development with Keil's µvsion2. Those who wish to dig deeper are encouraged to read the A51 user's guide.

Available at BB website, too!

CpE213 Homework Assignment #5 – Keil uVision2
150 points
Due: Thursday, Apr 3, 03

Download p78.pdf (assembly language programming with uVision2) and test.a51 (example assembly program used in p78.pdf) from the blackboard website (under Course Documents). Obtain gs51.pdf (uVision2 getting started guide), if you need a detailed reference for uVision2 IDE (Integrated Design Environment). Carefully go over step-by-step instructions shown in p78.pdf and be familiar with uVision2 IDE, and then do the followings.

1. Get a hardcopy of .lst file of the corrected test.a51 file and submit it.

2. Modify the program to read p1.7, p1.6 and p1.5 to get an integer value i, ranged from 0 (bit pattern 000) to 7 (bit pattern 111), then outputs 1 to p3.i. For example, if the user inputs a bit pattern 010 (e.g., an integer value 2) to p1.7, p1.6 and p1.5. p3 becomes 0000 0100. If the input pattern is 001, p3 becomes 0000 0010, etc…

3. Get a hardcopy of .lst file of the program and submit it.

4. Get eight different screen captures for every possible I/O combinations using the debugger. Note that both p1 and p3 must be shown. Use 'CTRL-PRINT SCREEN' to get a capture of the entire screen and 'ALT-PRINT SCREEN' to get a capture of the active window.

```
; HW5 part 2 solution


        cseg at 0
        ljmp start

table:  db 1,2,4,8,16,32,64,128


        cseg at 100h
start:  mov p1,#0e0h            ; make bits 5 to 7 inputs

loop:   mov a,p1                ; read p1

        mov r2,#5               ; shift right 5x (acc>>5)
rotlp:  rr a
        djnz r2,rotlp

        anl a,#7                ; mask input bits
        mov dptr,#table         ; get table address
        movc a,@a+dptr          ; get ith entry in table


        mov p3,a
        sjmp start              ; repeat forever

        end
```