# 8051 Microcontroller – Assembly Programming

EE4380 Spring 03

Class 3

*Pari vallal Kannan*

Center for Integrated Circuits and Systems

University of Texas at Dallas

# Topics

- Machine code
- 8051 Addressing Modes
- Jump, Loop and Call instructions
- Subroutines
- Simple delay loops

# 8051 Object Code

- Assembler converts assembly code to machine /object code
- Unique translation from assembly instruction to object code
  - Data sheet for 8051 lists the table of conversion
  - Manual assembly is cool !
- Object code is a (long) sequence of Machine instructions
- Each m/c instr. Can be 1 or more bytes long
- The m/c instr is a binary value and is written in Hex notation

# 8051 Object code (contd.)

- Assemblers produce a .lst file during assembly.
- Instruction to m/c code translation on a line by line basis is listed

| Instruction | Hex code | #bytes |
|---|---|---|
| nop | 00 | 1 |
| inc R0 | 08 | 1 |
| mov A, #55H | 74 55 | 2 |
| mov R0, #0AAH | 78 AA | 2 |
| mov R0, 0AAH | A8 AA | 2 |
| mov DPTR, #55AAH | 90 55 AA | 3 |

# 8051 Object code (contd.)

| Hex | Assembly |
|---|---|
| 0000: | .equ cout, 0x0030 |
| 0000: | .equ cin, 0x0032 |
| 0000: | .equ esc, 0x004E |
| 8000: | .org 0x8000 |
| 8000: 79 61 | mov r1, #'a' |
| 8002: 78 1A | mov r0, #26 |
| | next_char: |
| 8004: E9 | mov A, r1 |
| 8005: 12 00 30 | lcall cout |
| 8008: 09 | inc r1 |
| 8009: D8 F9 | djnz r0, next_char |
| 800B: 12 00 32 | lcall cin |
| 800E: 02 00 00 | ljmp 0x0000 |

| Address | Value |
|---|---|
| 8000 | 79 |
| 8001 | 61 |
| 8002 | 78 |
| 8003 | 1A |
| 8004 | E9 |
| 8005 | 12 |
| 8006 | 00 |
| 8007 | 30 |
| 8008 | 09 |
| 8009 | D8 |
| 800A | F9 |
| 800B | 12 |
| 800C | 00 |
| 800D | 32 |
| 800E | 02 |
| 800F | 00 |
| 8010 | 00 |

# 8051 Addressing Modes

- CPU can access data in various ways
  - Specify data directly in the instruction
  - Use different Addressing modes for data stored in code memory and data memory
- Five Addressing Modes
  - Immediate
  - Register
  - Direct
  - Register Indirect
  - Indexed

# 1. Immediate Addressing Mode

- Operand (data) directly specified in the instruction (opcode)
- Operand is a constant, known during assemble time
- Immediate data has to be preceded by “#” sign
- Eg.

```
mov A, #25H          ;A ← #0x25
mov DPTR, #1FFFH     ;DPTR ← #0x1FFF
temp EQU 40          ;assembler directive
mov R1, #temp        ;R1 ← 28H (40 decimal)
```

# 2. Register Addressing Mode

- Involves the use of registers to hold data
- Put the operand in a register and manipulate it by referring to the register (by name) in the instruction

```
mov A, R0        ;A ← contents (R0)
mov R2, A        ;R2 ← contents (A)
ADD A, R1        ;A ← contents (A) + contents (R1)
```

- Source and destination registers must match in size
- There may not be instructions for moving any register to any
  - mov R4, R7 ; invalid
  - Check with the instruction list before using
  - Assembly will fail in these cases

# 3. Direct Addressing Mode

- For data stored in RAM and Registers
  - All memory locations accessible by addresses
  - Same with all registers, ports, peripherals (SFRs) in 8051
- Use the address of the operand directly in the instruction
  - mov A, 40H ; A ← mem[40H] (note no # sign before 40H)
- Register addressing as Direct addressing
  - mov A, 4H ; 4H is the address for R4
  - mov A, R4 ; same as above. Both do the same
    ; but may have different op codes
- All registers and SFRs have addresses
- Stack in 8051 uses only direct addressing modes

# 4. Register Indirect Mode

- A register is used as a pointer
  - Register stores the address of the data
- Only R0, R1 and DPTR can be used for this purpose in 8051
- R0 and R1 can be used for internal memory (256 bytes incl. SFRs) or from 00H to FFH of external memory
  - mov A, @R0 ;A ← internal_mem[R0]
  - mov @R1, A ;A ← internal_mem[R1]
  - movx A, @R0 ; A ← external_mem[R0]
- DPTR can be used for external memory, for entire memory of 64K
  - movx A, @DPTR ;A ← external_mem[DPTR]
  - movx @DPTR, A ;vice versa

# 5. Indexed Addressing Mode

- Use a register for storing the pointer and another register for an offset
- Effective address is the sum = base+offset
  - Move code byte relative to DPTR to A. Effective address is DPTR + A
    - movc A, @A+DPTR ;A ← ext_code_mem [(A + DPTR)]
  - Move code byte relative to PC to A. Effective address is PC + A
    - movc A, @A+PC ;A ← ext_code_mem [(A + PC)]
- Widely used for implementing look-up tables, data arrays, character generators etc in code memory (ROM)

# Indexed Addressing Mode - Example

- Program to read a value x from P1 and send $x^2$ to P2

```
            ORG 0                   ; assembler directive
            mov DPTR, #LUT          ; 300H is the LUT address
            mov A, #0FFH
            mov P1, A               ; program the port P1 to input data
Again:      mov A, P1               ; read x
            movc A, @A+DPTR         ; get x^2 from LUT
            mov P2, A               ; output x^2 to P2
            sjmp again              ; for (1) loop

            ORG 300H                ;Look-up Table starts at 0x0300
LUT:        DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

# Program Control Instructions

- Unconditional Branch
  - ajmp addr11 ; absolute jump
  - ljmp addr16 ; long jump
  - sjmp rel ; short jump to relative address
  - jmp @A+DPTR ; jump indirect
- Conditional branch
  - jz, jnz rel ; short conditional jump to rel. addr
  - djnz rel ; decrement and jump if not zero
  - cjne rel ; compare and jump if not equal
- Subroutine Call
  - acall addr11 ; absolute subroutine call
  - lcall addr16 ; long subroutine call
  - ret ; return from subroutine call
  - reti ; return from ISV

# Pgm Branches – Target Address

- Target address can be,
  - absolute: A complete physical address
    - addr16: 16 bit address, anywhere in the 64k
    - addr11: 11 bit address, anywhere within 2k
  - rel: relative (forward or backward) -128 bytes to +127 bytes from the current code location
- Target address calculation for relative jumps
  - PC of next instruction + rel address
  - For jump backwards, drop the carry
    - PC = 15H, SJMP 0FEH
    - Address is 15H + FEH = 13H
    - Basically jump to next instruction minus two (current instruction)

# Conditional Jumps

- jz, jnz : Conditional on A==0
  - Checks to see if A is zero
  - jz jumps if A is zero and jnz jumps if A not zero
  - No arithmetic op need be performed (as opposed to 8086)
- djnz : dec a byte and jump if  not equal to zero
  - djnz Rn, rel
  - djnz direct, rel
- jnc : Conditional on carry CY flag
  - jc  rel
  - jnc rel
- Cjne : compare and jump if not equal
  - cjne A, direct, rel
  - cjne ARn, #data, rel
  - cjne @Rn, #data, rel

# Loop using djnz

- Add 3 to A ten times

```
            mov     A, #0           ; clear A
            mov     R2, #10         ; R2 ← 10, can also say 0AH
AGAIN:      add     A, #03          ; add 3 to A
            djnz    R2, AGAIN       ; repeat until R2==0
            mov     R5, A           ; save the result in R5
```

- Loop within loop using djnz

```
            mov     R3, #100
loop1:      mov     R2, #10         ; trying for 1000 loop iterations
loop2:      nop                     ; no operation
            djnz    R2, loop2       ; repeat loop2 until R2==0
            djnz    R3, loop1       ; repeat loop1 until R3==0
```

# Unconditional Jumps

- LJMP addr16
  - Long jump. Jump to a 2byte target address
  - 3 byte instruction
- SJMP rel
  - Jump to a relative address from PC+127 to PC-128
  - Jump to PC + 127 (00H – 7FH)
  - Jump to PC – 128 (80H – FFH)

# Call Instructions

- Subroutines:
  - Reusable code snippets
- LCALL addr16
  - Long call. 3 byte instr. Call any subroutine in entire 64k code space

    push PC

    jmp address
- ACALL addr11
  - 2 byte instruction. Call any subroutine within 2k of code space
  - Saves code ROM for devices with less than 64K ROM
- RET
  - Return from a subroutine call,

    pop PC

# Machine Cycle

- Number of clock cycles used to perform one instruction
- Varies with instruction. Usually the lowest is quoted as the machine cycle
- For 8051, 12 clock cycles are minimum needed per instruction
- Time per machine cycle
  - $T_{mc}$ = Clocks per machine cycle / Clock frequency
  - For 8051 clocked at 11.0592MHz,
    - $T_{mc}$ = 12 / 11.0592M = 1.085 micro seconds
- Time spent executing an instruction
  - $T_{instr}$ = machine cycles for the instruction * $T_{mc}$
  - For the nop instruction, machine cycles = 1. So
    - $T_{instr}$ = 1 * 1.085 = 1.085 micro seconds

# Simple delay loops

- Find the time delay for the subroutine

```
DELAY:    mov R3, #200      ; 1 machine cycle
HERE:     djnz R3, HERE     ; 2 machine cycles
          RET               ; 1 machine cycle
```

- Calculation
  - Total machine cycles = 200*2 + 1 + 1 = 402
  - Time = 402 * 1.085us (assuming 11.0592 MHz clk)
    = 436.17us
- Similarly any delay can be obtained by loop within loop technique
- For much longer delays, use timers

# Class –3 : Review

- How does 8051 machine code look ?
- What are 8051 addressing modes ?
- What are program control instructions ?
- Conditional Vs Unconditional branches
- Subroutines
- What is Machine cycle ?
- How to calculate exact time spent in executing a program ? Or how to write exact time delay loops ?

# Thanks