The 8051 Microcontroller and Embedded Systems

Muhammad Ali Mazidi
Janice Gillispie Mazidi

SOFTWARE ENCLOSED
Book not returnable if software has been removed.
PRENTICE-HALL, INC.

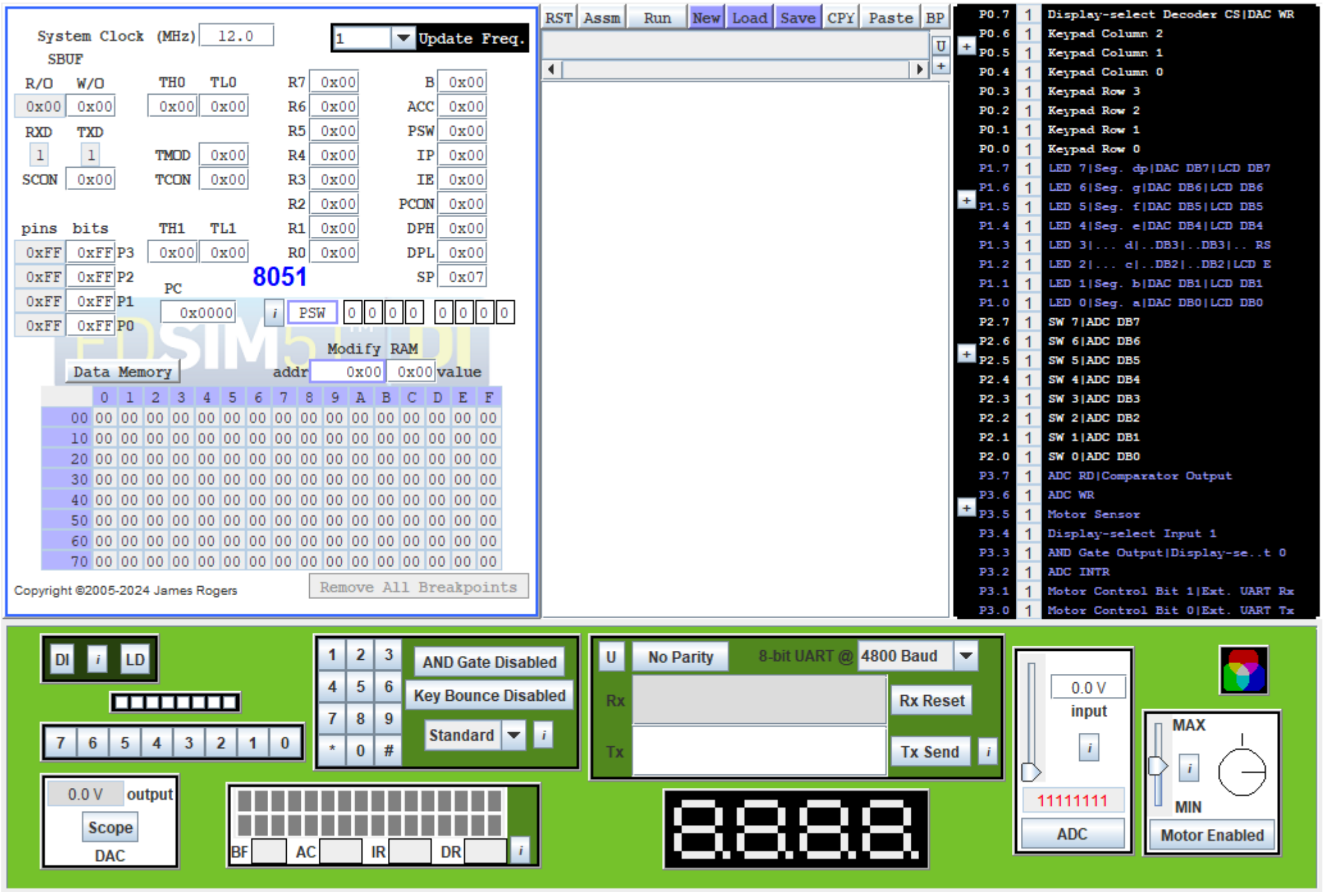
The 8051 Microcontroller and Embedded Systems
Using Assembly and C

Second Edition

Muhammad Ali Mazidi
Janice Gillispie Mazidi
Rolin D. McKinlay

CONTENTS

- ***Introduction to Computing***
- ***The 8051 Microcontrollers***
- ***8051 Assembly Language Programming***
- ***Branch Instructions***
- ***I/O Port Programming***
- ***8051 Addressing Modes***
- ***Arithmetic & Logic Instructions And Programs***
- ***8051 Programming in C***
- ***8051 Hardware Connection and Hex File***
- ***8051 Timer/Counter Programming in Assembly and C***
- ***8051 Serial Port Programming in Assembly and C***
- ***Interrupts Programming in Assembly and C***
- ***8051 Interfacing to External Memory***
- ***8051 Real World Interfacing I: LCD,ADC AND SENSORS***
- ***LCD and Keyboard Interfacing***
- ***8051 Interfacing with 8255***


System Clock (MHz) 12.0
1 Update Freq.
SBUF
R/O W/O
0x00 0x00
TH0 TL0
0x00 0x00
R7 0x00
R6 0x00
R5 0x00
R4 0x00
R3 0x00
R2 0x00
R1 0x00
R0 0x00
B 0x00
ACC 0x00
PSW 0x00
IP 0x00
IE 0x00
PCON 0x00
DPH 0x00
DPL 0x00
SP 0x07
RXD TXD
1 1
SCON 0x00
TMOD 0x00
TCON 0x00
pins bits
0xFF 0xFF P3
0xFF 0xFF P2
0xFF 0xFF P1
0xFF 0xFF P0
TH1 TL1
0x00 0x00
8051
PC
0x0000
PSW 0 0 0 0 0 0 0 0
Modify RAM
Data Memory
addr 0x00 0x00 value
0 1 2 3 4 5 6 7 8 9 A B C D E F
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Copyright ©2005-2024 James Rogers
Remove All Breakpoints
RST Assm Run New Load Save CPY Paste BP
P0.7 1 Display-select Decoder CS|DAC WR
P0.6 1 Keypad Column 2
P0.5 1 Keypad Column 1
P0.4 1 Keypad Column 0
P0.3 1 Keypad Row 3
P0.2 1 Keypad Row 2
P0.1 1 Keypad Row 1
P0.0 1 Keypad Row 0
P1.7 1 LED 7|Seg. dp|DAC DB7|LCD DB7
P1.6 1 LED 6|Seg. g|DAC DB6|LCD DB6
P1.5 1 LED 5|Seg. f|DAC DB5|LCD DB5
P1.4 1 LED 4|Seg. e|DAC DB4|LCD DB4
P1.3 1 LED 3|... d|..DB3|..DB3|.. RS
P1.2 1 LED 2|... c|..DB2|..DB2|LCD E
P1.1 1 LED 1|Seg. b|DAC DB1|LCD DB1
P1.0 1 LED 0|Seg. a|DAC DB0|LCD DB0
P2.7 1 SW 7|ADC DB7
P2.6 1 SW 6|ADC DB6
P2.5 1 SW 5|ADC DB5
P2.4 1 SW 4|ADC DB4
P2.3 1 SW 3|ADC DB3
P2.2 1 SW 2|ADC DB2
P2.1 1 SW 1|ADC DB1
P2.0 1 SW 0|ADC DB0
P3.7 1 ADC RD|Comparator Output
P3.6 1 ADC WR
P3.5 1 Motor Sensor
P3.4 1 Display-select Input 1
P3.3 1 AND Gate Output|Display-se..t 0
P3.2 1 ADC INTR
P3.1 1 Motor Control Bit 1|Ext. UART Rx
P3.0 1 Motor Control Bit 0|Ext. UART Tx
DI LD
7 6 5 4 3 2 1 0
1 2 3 4 5 6 7 8 9 * 0 #
AND Gate Disabled
Key Bounce Disabled
Standard
U No Parity 8-bit UART @ 4800 Baud
Rx
Tx
Rx Reset
Tx Send
0.0 V input
11111111
ADC
MAX
MIN
Motor Enabled
0.0 V output
Scope
DAC
BF AC IR DR

8051 MICROCONTROLLERS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

Mazidi, Mazidi and McKinlay

OUTLINES

- Microcontrollers and embedded processors
- Overview of the 8051 family

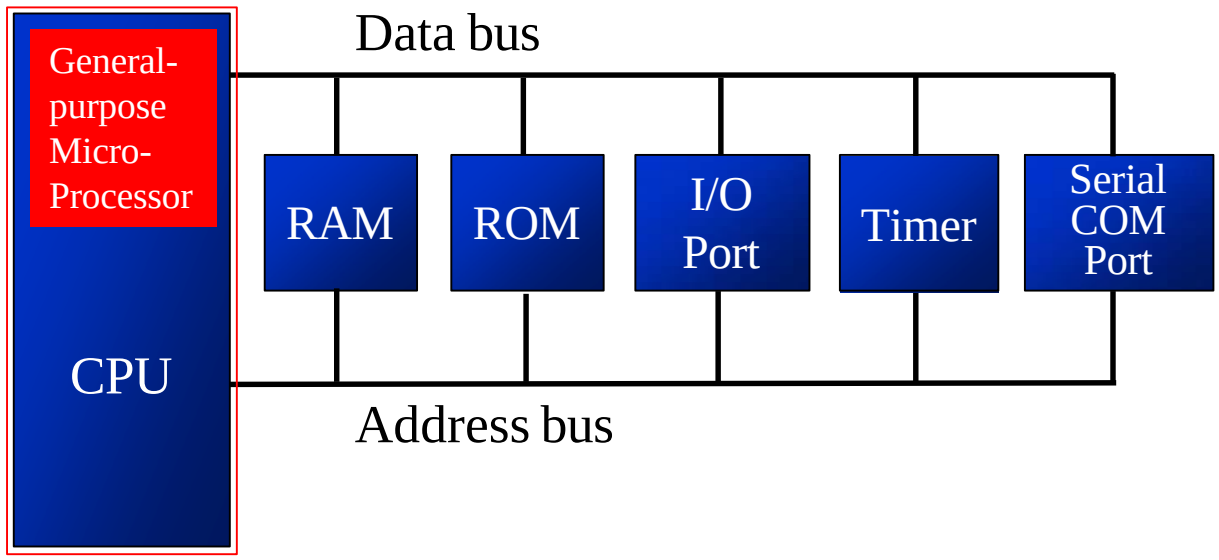
MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

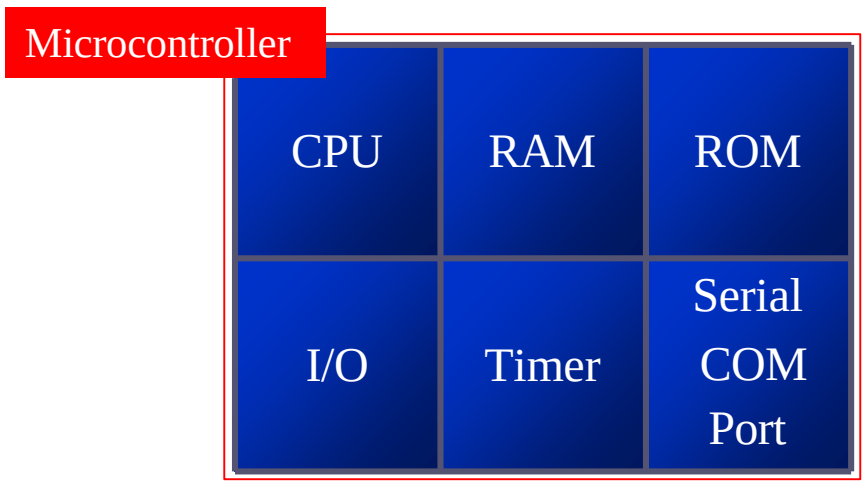
Microcontroller vs. General-Purpose Microprocessor

- General-purpose microprocessors contains
  - No RAM
  - No ROM
  - No I/O ports
- Microcontroller has
  - CPU (microprocessor)
  - RAM
  - ROM
  - I/O ports
  - Timer
  - ADC and other peripherals

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Microcontroller vs. General-Purpose Microprocessor (cont')





MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Microcontroller vs. General-Purpose Microprocessor (cont')

- General-purpose microprocessors
  - Must add RAM, ROM, I/O ports, and timers externally to make them functional
  - Make the system bulkier and much more expensive
  - Have the advantage of versatility on the amount of RAM, ROM, and I/O ports
- Microcontroller
  - The fixed amount of on-chip ROM, RAM, and number of I/O ports makes them ideal for many applications in which cost and space are critical
  - In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Microcontrollers for Embedded Systems

- An embedded product uses a microprocessor (or microcontroller) to do one task and one task only
  - There is only one application software that is typically burned into ROM
- A PC, in contrast with the embedded system, can be used for any number of applications
  - It has RAM memory and an operating system that loads a variety of applications into RAM and lets the CPU run them
  - A PC contains or is connected to various embedded products
    - Each one peripheral has a microcontroller inside it that performs only one task

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Microcontrollers for Embedded Systems (cont')

- Home
  - Appliances, intercom, telephones, security systems, garage door openers, answering machines, fax machines, home computers, TVs, cable TV tuner, VCR, camcorder, remote controls, video games, cellular phones, musical instruments, sewing machines, lighting control, paging, camera, pinball machines, toys, exercise equipment
- Office
  - Telephones, computers, security systems, fax machines, microwave, copier, laser printer, color printer, paging
- Auto
  - Trip computer, engine control, air bag, ABS, instrumentation, security system, transmission control, entertainment, climate control, cellular phone, keyless entry

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

x86 PC Embedded Applications

- Many manufactures of general-purpose microprocessors have targeted their microprocessor for the high end of the embedded market
  - There are times that a microcontroller is inadequate for the task
- When a company targets a general-purpose microprocessor for the embedded market, it optimizes the processor used for embedded systems
- Very often the terms embedded processor and microcontroller are used interchangeably

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

x86 PC Embedded Applications (cont')

- One of the most critical needs of an embedded system is to decrease power consumption and space
- In high-performance embedded processors, the trend is to integrate more functions on the CPU chip and let designer decide which features he/she wants to use
- In many cases using x86 PCs for the high-end embedded applications
  - Saves money and shortens development time
    - A vast library of software already written
    - Windows is a widely used and well understood platform

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Choosing a Microcontroller

- 8-bit microcontrollers
  - Motorola's 6811
  - Intel's 8051
  - Zilog's Z8
  - Microchip's PIC
- There are also 16-bit and 32-bit microcontrollers made by various chip makers

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Criteria for Choosing a Microcontroller

- Meeting the computing needs of the task at hand efficiently and cost effectively
  - Speed
  - Packaging
  - Power consumption
  - The amount of RAM and ROM on chip
  - The number of I/O pins and the timer on chip
  - How easy to upgrade to higher-performance or lower power-consumption versions
  - Cost per unit

MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

Criteria for Choosing a Microcontroller (cont')

- Availability of software development tools, such as compilers, assemblers, and debuggers
- Wide availability and reliable sources of the microcontroller
  - The 8051 family has the largest number of diversified (multiple source) suppliers
    - Intel (original)
    - Atmel
    - Philips/Signetics
    - AMD
    - Infineon (formerly Siemens)
    - Matra
    - Dallas Semiconductor/Maxim

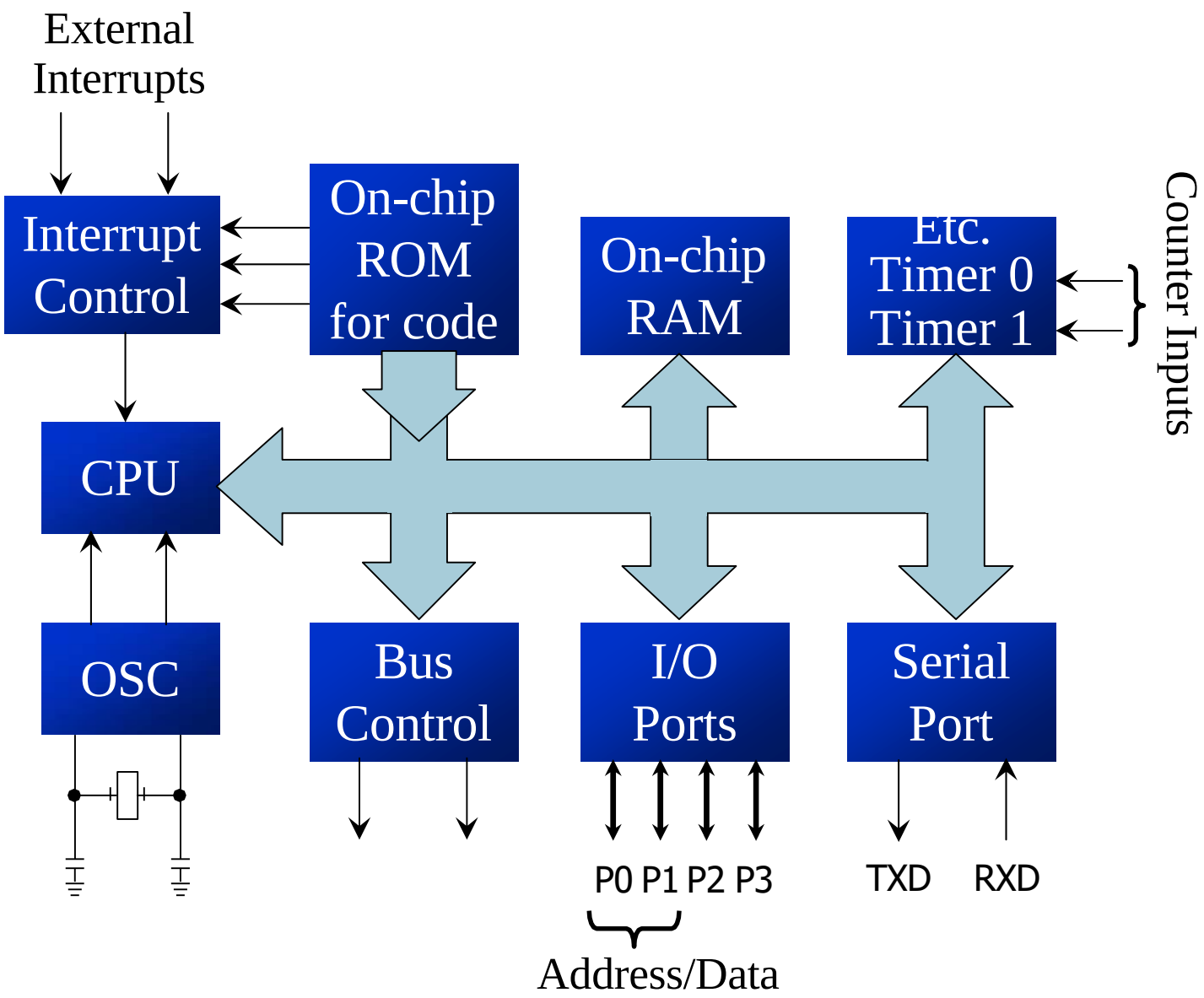
OVERVIEW OF 8051 FAMILY

8051 Microcontroller

- Intel introduced 8051, referred as MCS-51, in 1981
  - The 8051 is an 8-bit processor
    - The CPU can work on only 8 bits of data at a time
  - The 8051 had
    - 128 bytes of RAM
    - 4K bytes of on-chip ROM
    - Two timers
    - One serial port
    - Four I/O ports, each 8 bits wide
    - 6 interrupt sources
- The 8051 became widely popular after allowing other manufactures to make and market any flavor of the 8051, but remaining code-compatible

OVERVIEW OF 8051 FAMILY

8051 Microcontroller (cont')



OVERVIEW OF 8051 FAMILY

8051 Family

- The 8051 is a subset of the 8052
- The 8031 is a ROM-less 8051
  - Add external ROM to it
  - You lose two ports, and leave only 2 ports for I/O operations

Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

OVERVIEW OF 8051 FAMILY

Various 8051 Microcontrollers

- 8751 microcontroller
  - UV-EPROM
    - PROM burner
    - UV-EPROM eraser takes 20 min to erase
- AT89C51 from Atmel Corporation
  - Flash (erase before write)
    - ROM burner that supports flash
    - A separate eraser is not needed
- DS89C4x0 from Dallas Semiconductor, now part of Maxim Corp.
  - Flash
    - Comes with on-chip loader, loading program to on-chip flash via PC COM port

OVERVIEW OF 8051 FAMILY

Various 8051 Microcontrollers (cont')

- DS5000 from Dallas Semiconductor
  - NV-RAM (changed one byte at a time), RTC (real-time clock)
    - Also comes with on-chip loader
- OTP (one-time-programmable) version of 8051
- 8051 family from Philips
  - ADC, DAC, extended I/O, and both OTP and flash

8051 ASSEMBLY LANGUAGE PROGRAMMING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

Mazidi, Mazidi and McKinlay

INSIDE THE 8051

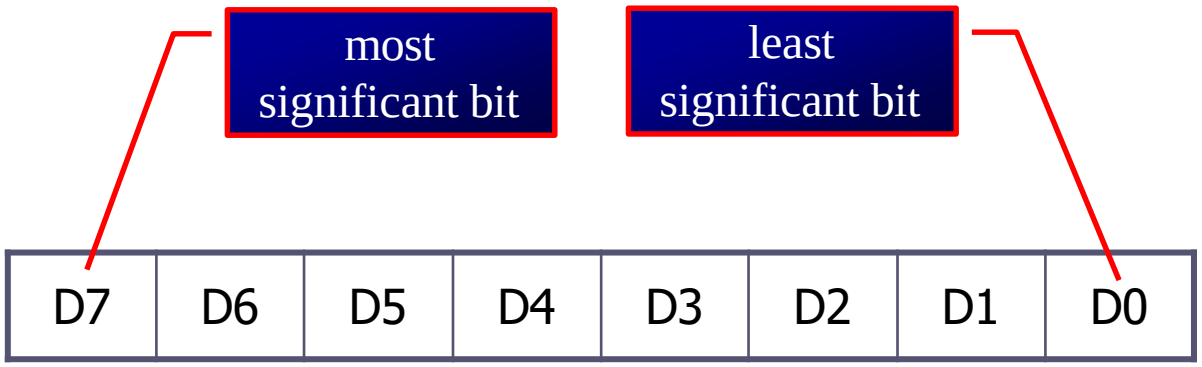
Registers

- Register are used to store information temporarily, while the information could be
  - a byte of data to be processed, or
  - an address pointing to the data to be fetched
- The vast majority of 8051 register are 8-bit registers
  - There is only one data type, 8 bits

INSIDE THE 8051

Registers (cont')

- The 8 bits of a register are shown from MSB D7 to the LSB D0
  - With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed

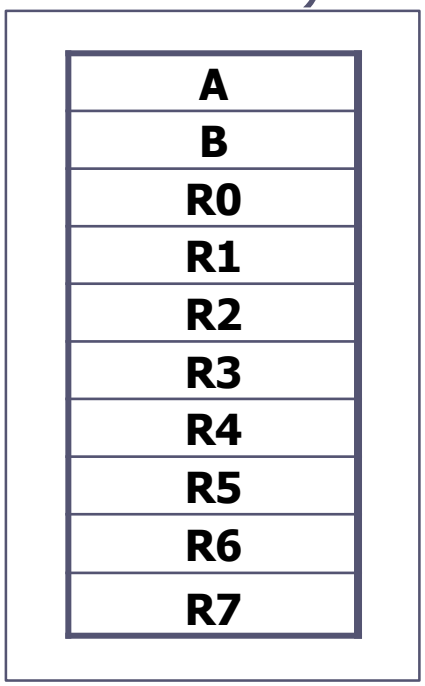


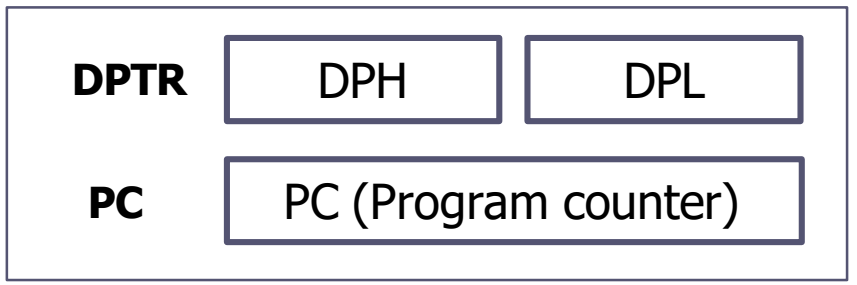
8 bit Registers

INSIDE THE 8051

Registers (cont')

- The most widely used registers
  - A (Accumulator)
    - For all arithmetic and logic instructions
  - B, R0, R1, R2, R3, R4, R5, R6, R7
  - DPTR (data pointer), and PC (program counter)





INSIDE THE 8051

MOV Instruction

MOV destination, source ;copy source to dest.

- The instruction tells the CPU to move (in reality, COPY) the source operand to the destination operand

"#" signifies that it is a value

```
MOV   A,#55H      ;load value 55H into reg. A
MOV   R0,A        ;copy contents of A into R0
                  ;(now A=R0=55H)
MOV   R1,A        ;copy contents of A into R1
                  ;(now A=R0=R1=55H)
MOV   R2,A        ;copy contents of A into R2
                  ;(now A=R0=R1=R2=55H)
MOV   R3,#95H     ;load value 95H into R3
                  ;(now R3=95H)
MOV   A,R3        ;copy contents of R3 into A
                  ;now A=R3=95H
```

INSIDE THE 8051

MOV Instruction (cont')

- Notes on programming
  - Value (proceeded with #) can be loaded directly to registers A, B, or R0 – R7
    - `MOV A, #23H`
    - `MOV R5, #0F9H`

Add a 0 to indicate that F is a hex number and not a letter

If it's not preceded with #, it means to load from a memory location

  - If values 0 to F moved into an 8-bit register, the rest of the bits are assumed all zeros
    - "`MOV A, #5`", the result will be A=05; i.e., A = 00000101 in binary
  - Moving a value that is too large into a register will cause an error
    - `MOV A, #7F2H` ; ILLEGAL: 7F2H>8 bits (FFH)

INSIDE THE 8051

ADD Instruction

```
ADD A, source     ;ADD the source operand
                  ;to the  accumulator
```

- The `ADD` instruction tells the CPU to add the source byte to register A and put the result in register A
- Source operand can be either a register or immediate data, but the destination must always be register A
  - "`ADD R4, A`" and "`ADD R2, #12H`" are invalid since A must be the destination of any arithmetic operation

```
MOV A, #25H       ;load 25H into A
MOV R2, #34H      ;load 34H into R2
ADD A, R2 ;add R2 to Accumulator
                  ;(A = A + R2)
```

There are always many ways to write the same program, depending on the registers used

```
MOV A, #25H       ;load one operand
          ;into A (A=25H)
ADD A, #34H       ;add the second
                  ;operand 34H to A
```

8051 ASSEMBLY PROGRAMMING

Structure of Assembly Language

- In the early days of the computer, programmers coded in machine language, consisting of 0s and 1s
  - Tedious, slow and prone to error
- Assembly languages, which provided mnemonics for the machine code instructions, plus other features, were developed
  - An Assembly language program consist of a series of lines of Assembly language instructions
- Assembly language is referred to as a low-level language
  - It deals directly with the internal structure of the CPU

8051 ASSEMBLY PROGRAMMING

Structure of Assembly Language

- Assembly language instruction includes
  - a mnemonic (abbreviation easy to remember)
    - the commands to the CPU, telling it what those to do with those items
  - optionally followed by one or two operands
    - the data items being manipulated
- A given Assembly language program is a series of statements, or lines
  - Assembly language instructions
    - Tell the CPU what to do
  - Directives (or pseudo-instructions)
    - Give directions to the assembler

8051 ASSEMBLY PROGRAMMING

Structure of Assembly Language

- An Assembly language instruction consists of four fields:

[label:] Mnemonic [operands] [;comment]

```
   ORG   0H              ;start(origin) at location
   0
   MOV   R5, #25H         ;load 25H into R5
   MOV   R7, #34H         ;load 34H in
   MOV   A, #0            ;load 0 into
   ADD   A, R5            ;add content
                          ;now A = A +
   ADD   A, R7            ;add contents of R7 to A
                          ;now A = A + R7
   ADD   A, #12H          ;add to A value 12H
                          ;now A = A + 12H
HERE: SJMP HERE           ;stay in this loop
   END                    ;en
```

Mnemonics produce opcodes

Directives do not generate any machine code and are used only by the assembler

The label field allows the program to refer to a line of code by name

Comments may be at the end of a line or on a line by themselves
The assembler ignores comments

ASSEMBLING AND RUNNING AN 8051 PROGRAM

- The step of Assembly language program are outlines as follows:

1) First we use an editor to type a program, many excellent editors or word processors are available that can be used to create and/or edit the program
   - Notice that the editor must be able to produce an ASCII file
   - For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension “asm“ or “src”, depending on which assembly you are using

ASSEMBLING AND RUNNING AN 8051 PROGRAM (cont')

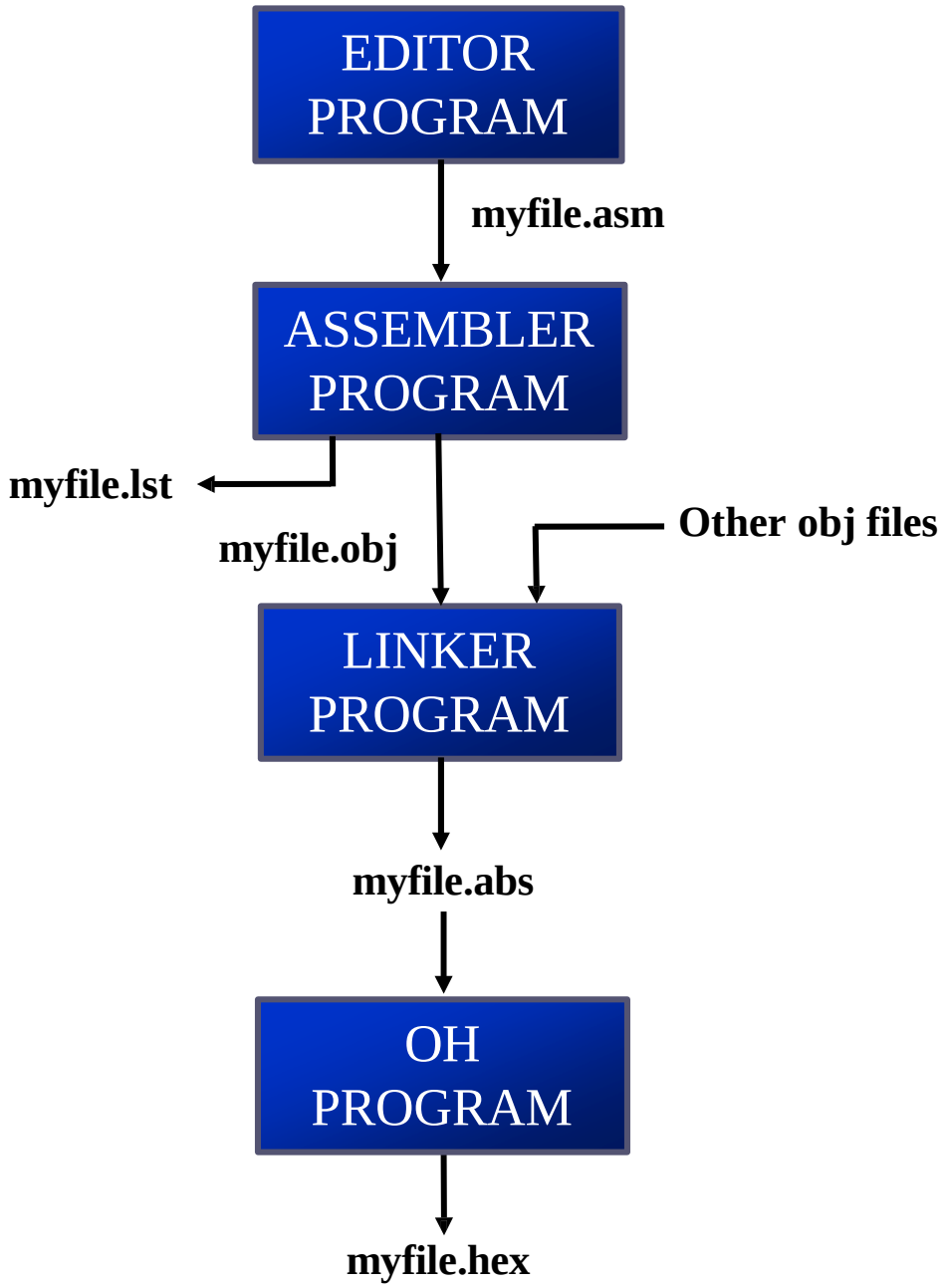
2) The "asm" source file containing the program code created in step 1 is fed to an 8051 assembler
   - The assembler converts the instructions into machine code
   - The assembler will produce an object file and a list file
   - The extension for the object file is "obj" while the extension for the list file is "lst"
3) Assembler require a third step called linking
   - The linker program takes one or more object code files and produce an absolute object file with the extension "abs"
   - This abs file is used by 8051 trainers that have a monitor program

ASSEMBLING AND RUNNING AN 8051 PROGRAM (cont')

4) Next the "abs" file is fed into a program called "OH" (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM
   - This program comes with all 8051 assemblers
   - Recent Windows-based assemblers combine step 2 through 4 into one step

ASSEMBLING AND RUNNING AN 8051 PROGRAM

Steps to Create a Program



ASSEMBLING AND RUNNING AN 8051 PROGRAM

lst File

- The lst (list) file, which is optional, is very useful to the programmer
  - It lists all the opcodes and addresses as well as errors that the assembler detected
  - The programmer uses the lst file to find the syntax errors or debug

```
1 0000          ORG 0H          ;start (origin) at 0
2 0000  7D25    MOV R5,#25H     ;load 25H into R5
3 0002  7F34    MOV R7,#34H     ;load 34H into R7
4 0004  7400    MOV A,#0        ;load 0 into A
5 0006  2D      ADD A,R5        ;add contents of R5 to A
                                ;now A = A + R5
6 0007  2F      ADD A,R7        ;add contents of R7 to A
                                ;now A = A + R7
7 0008  2412    ADD A,#12H      ;add to A value
                    12H
8 000A  80EF HERE:              ;now A = A + 12H
9 000C          END SJMP HERE;stay in this loop
                                ;end of asm source file
```

address

PROGRAM COUNTER AND ROM SPACE

Program Counter

- The program counter points to the address of the next instruction to be executed
  - As the CPU fetches the opcode from the program ROM, the program counter is increasing to point to the next instruction
- The program counter is 16 bits wide
  - This means that it can access program addresses 0000 to FFFFH, a total of 64K bytes of code

PROGRAM COUNTER AND ROM SPACE

Power up

- All 8051 members start at memory address 0000 when they're powered up
  - Program Counter has the value of 0000
  - The first opcode is burned into ROM address 0000H, since this is where the 8051 looks for the first instruction when it is booted
  - We achieve this by the `ORG` statement in the source program

PROGRAM COUNTER AND ROM SPACE

Placing Code in ROM

- Examine the list file and how the code is placed in ROM

```
1 0000            ORG 0H              ;start (origin) at 0
2 0000  7D25      MOV R5,#25H         ;load 25H into R5
3 0002  7F34      MOV R7,#34H         ;load 34H into R7
4 0004  7400      MOV A,#0            ;load 0 into A
5 0006  2D        ADD A,R5            ;add contents of R5 to A
                                      ;now A = A + R5
6 0007  2F        ADD A,R7            ;add contents of R7 to A
                                      ;now A = A + R7
7 0008  2412      ADD A,#12H          ;add to A value 12H
                                      ;now A = A + 12H
8 000A  80EF      HERE: SJMP HERE     ;stay in this loop
9 000C            END                 ;end of asm source file
```

ROM Address	Machine Language	Assembly Language
0000	7D25	MOV R5, #25H
0002	7F34	MOV R7, #34H
0004	7400	MOV A, #0
0006	2D	ADD A, R5
0007	2F	ADD A, R7
0008	2412	ADD A, #12H
000A	80EF	HERE: SJMP HERE

PROGRAM COUNTER AND ROM SPACE

Placing Code in ROM (cont')

- After the program is burned into ROM, the opcode and operand are placed in ROM memory location starting at 0000

ROM contents

Address	Code
0000	7D
0001	25
0002	7F
0003	34
0004	74
0005	00
0006	2D
0007	2F
0008	24
0009	12
000A	80
000B	FE

PROGRAM COUNTER AND ROM SPACE

Executing Program

- A step-by-step description of the action of the 8051 upon applying power on it
  1. When 8051 is powered up, the PC has 0000 and starts to fetch the first opcode from location 0000 of program ROM
     - Upon executing the opcode 7D, the CPU fetches the value 25 and places it in R5
     - Now one instruction is finished, and then the PC is incremented to point to 0002, containing opcode 7F
  2. Upon executing the opcode 7F, the value 34H is moved into R7
     - The PC is incremented to 0004

PROGRAM COUNTER AND ROM SPACE

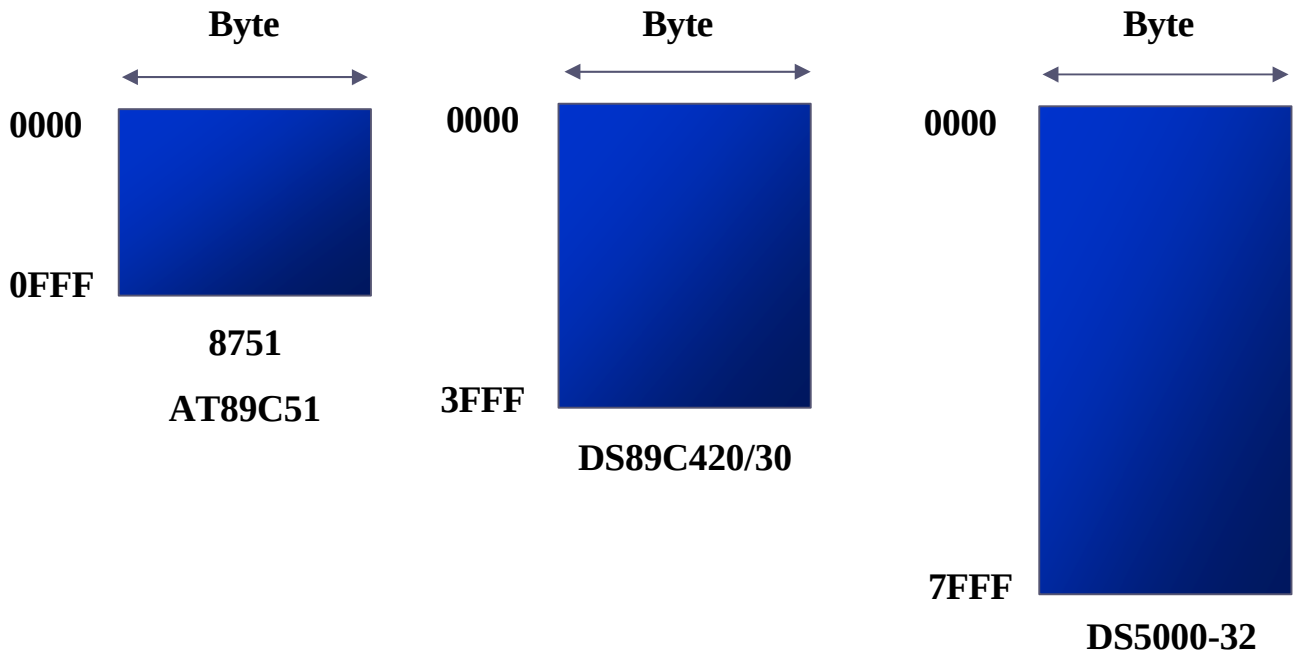
Executing Program (cont')

- (cont')
  3. The instruction at location 0004 is executed and now PC = 0006
  4. After the execution of the 1-byte instruction at location 0006, PC = 0007
  5. Upon execution of this 1-byte instruction at 0007, PC is incremented to 0008
     - This process goes on until all the instructions are fetched and executed
     - The fact that program counter points at the next instruction to be executed explains some microprocessors call it the instruction pointer

PROGRAM COUNTER AND ROM SPACE

ROM Memory Map in 8051 Family

- No member of 8051 family can access more than 64K bytes of opcode
  - The program counter is a 16-bit register



8051 DATA TYPES AND DIRECTIVES

Data Type

- 8051 microcontroller has only one data type - 8 bits
  - The size of each register is also 8 bits
  - It is the job of the programmer to break down data larger than 8 bits (00 to FFH, or 0 to 255 in decimal)
  - The data types can be positive or negative

8051 DATA TYPES AND DIRECTIVES

Assembler Directives

- The DB directive is the most widely used data directive in the assembler
  - It is used to define the 8-bit data
  - When DB is used to define data, the numbers can be in decimal, binary, hex, ASCII formats

```
        ORG     500H
DATA1:  DB      28              ;DECIMAL (1C in Hex)
DATA2:  DB      00110101B       ;BINARY (35 in Hex)
DATA3:  DB      39H             ;HEX
        ORG     510H
DATA4:  DB      "2591"
        ORG     518H
DATA6:  DB      "My name is Joe"
                                ;ASCII CHARACTERS
```

The "D" after the decimal number is optional, but using "B" (binary) and "H" (hexadecimal) for the others is required

The Assembler will convert the numbers into hex

Place ASCII in quotation marks The Assembler will assign ASCII code for the numbers or characters

Define ASCII strings larger than two characters

8051 DATA TYPES AND DIRECTIVES

Assembler Directives (cont')

- ORG (origin)
  - The ORG directive is used to indicate the beginning of the address
  - The number that comes after ORG can be either in hex and decimal
    - If the number is not followed by H, it is decimal and the assembler will convert it to hex
- END
  - This indicates to the assembler the end of the source (asm) file
  - The END directive is the last line of an 8051 program
    - Mean that in the code anything after the END directive is ignored by the assembler

8051 DATA TYPES AND DIRECTIVES

Assembler directives (cont')

- `EQU` (equate)
  - This is used to define a constant without occupying a memory location
  - The `EQU` directive does not set aside storage for a data item but associates a constant value with a data label
    - When the label appears in the program, its constant value will be substituted for the label

8051 DATA TYPES AND DIRECTIVES

Assembler directives (cont')

- EQU (equate) (cont')
  - Assume that there is a constant used in many different places in the program, and the programmer wants to change its value throughout
    - By the use of EQU, one can change it once and the assembler will change all of its occurrences

Use EQU for the counter constant

```
COUNT   EQU 25
...     ....
MOV     R3, #COUNT
```

The constant is used to load the R3 register

FLAG BITS AND PSW REGISTER

Program Status Word

- The program status word (PSW) register, also referred to as the flag register, is an 8 bit register
  - Only 6 bits are used
    - These four are CY (carry), AC (auxiliary carry), P (parity), and OV (overflow)
      - They are called conditional flags, meaning that they indicate some conditions that resulted after an instruction was executed
    - The PSW3 and PSW4 are designed as RS0 and RS1, and are used to change the bank
  - The two unused bits are user-definable

FLAG BITS AND PSW REGISTER

Program Status Word (cont')

CY	AC	F0	RS1	RS0	OV	--	P

A carry from D3 to D4

Carry out from the d7 bit

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
--	PSW.5	Available to the user for general purpose
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
--	PSW.1	User definable bit.
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of 1 bits in the accumulator.

Reflect the number of 1s in register A

The result of signed number operation is too large, causing the high-order bit to overflow into the sign bit

RS1	RS0	Register Bank	Address
0	0	0	00H – 07H
0	1	1	08H – 0FH
1	0	2	10H – 17H
1	1	3	18H – 1FH

FLAG BITS AND PSW REGISTER

ADD Instruction And PSW

Instructions that affect flag bits

Instruction	CY	OV	AC
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RPC	X		
PLC	X		
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, bit	X		
ANL C, /bit	X		
ORL C, bit	X		
ORL C, /bit	X		
MOV C, bit	X		
CJNE	X		

FLAG BITS AND PSW REGISTER

ADD Instruction And PSW (cont')

- The flag bits affected by the ADD instruction are CY, P, AC, and OV

Example 2-2

Show the status of the CY, AC and P flag after the addition of 38H and 2FH in the following instructions.

```
MOV A, #38H
ADD A, #2FH ;after the addition A=67H, CY=0
```

Solution:

38	00111000
+ 2F	00101111
67	01100111

CY = 0 since there is no carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bi

P = 1 since the accumulator has an odd number of 1s (it has five 1s)

FLAG BITS AND PSW REGISTER

ADD Instruction And PSW (cont')

Example 2-3

Show the status of the CY, AC and P flag after the addition of 9CH and 64H in the following instructions.

```
MOV A, #9CH
ADD A, #64H  ;after the addition A=00H, CY=1
```

Solution:

9C	10011100
+ 64	01100100
100	00000000

CY = 1 since there is a carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bi

P = 0 since the accumulator has an even number of 1s (it has zero 1s)

FLAG BITS AND PSW REGISTER

ADD Instruction And PSW (cont')

Example 2-4

Show the status of the CY, AC and P flag after the addition of 88H and 93H in the following instructions.

```
MOV A, #88H
ADD A, #93H  ;after the addition A=1BH, CY=1
```

Solution:

88	10001000
+ 93	10010011
11B	00011011

CY = 1 since there is a carry beyond the D7 bit

AC = 0 since there is no carry from the D3 to the D4 bi

P = 0 since the accumulator has an even number of 1s (it has four 1s)

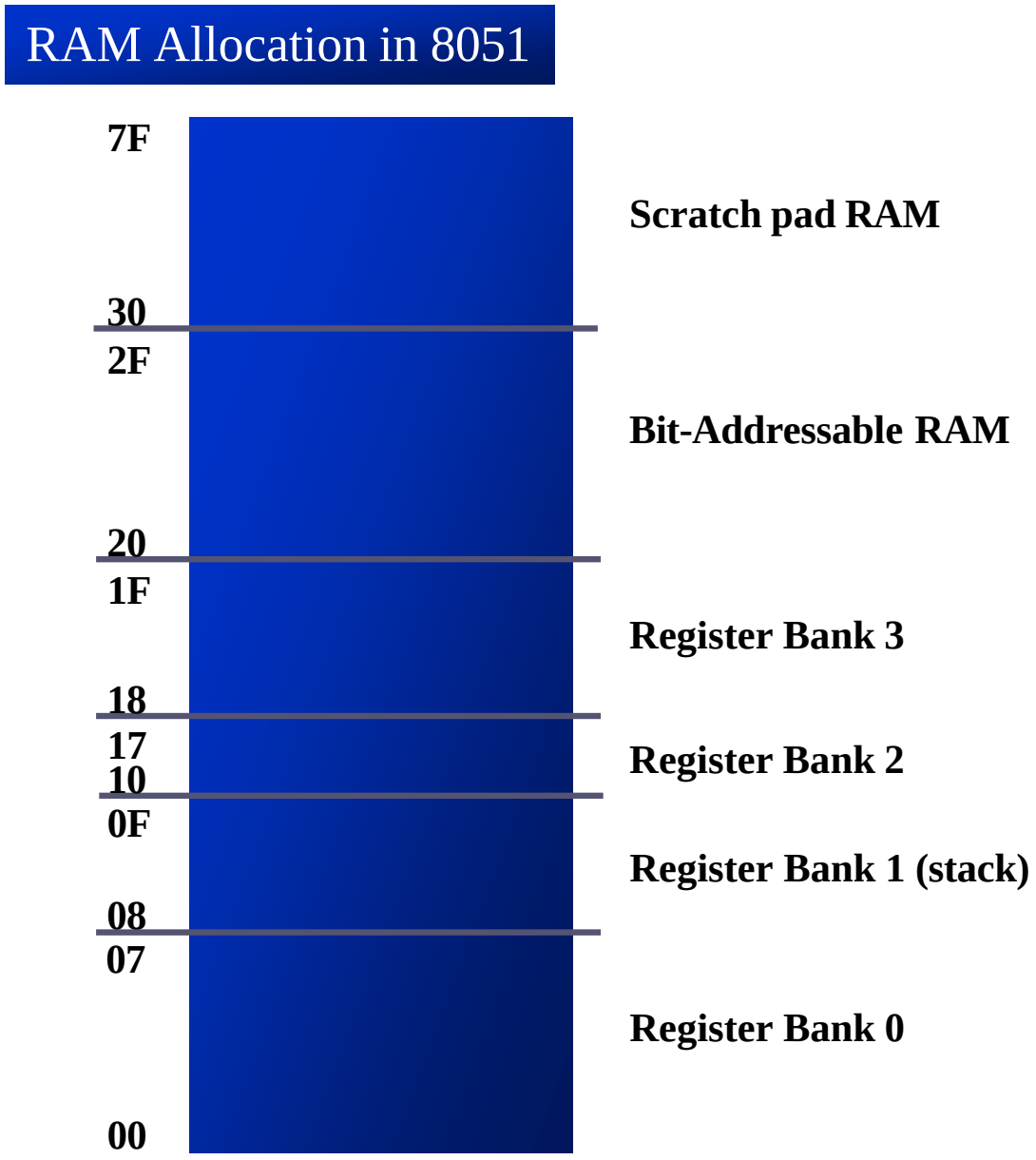
REGISTER BANKS AND STACK

RAM Memory Space Allocation

- There are 128 bytes of RAM in the 8051
  - Assigned addresses 00 to 7FH
- The 128 bytes are divided into three different groups as follows:
  1) A total of 32 bytes from locations 00 to 1F hex are set aside for register banks and the stack
  2) A total of 16 bytes from locations 20H to 2FH are set aside for bit-addressable read/write memory
  3) A total of 80 bytes from locations 30H to 7FH are used for read and write storage, called scratch pad

8051 REGISTER BANKS AND STACK

RAM Memory Space Allocation (cont')



8051 REGISTER BANKS AND STACK

Register Banks

- These 32 bytes are divided into 4 banks of registers in which each bank has 8 registers, R0-R7
  - RAM location from 0 to 7 are set aside for bank 0 of R0-R7 where R0 is RAM location 0, R1 is RAM location 1, R2 is RAM location 2, and so on, until memory location 7 which belongs to R7 of bank 0
  - It is much easier to refer to these RAM locations with names such as R0, R1, and so on, than by their memory locations
- Register bank 0 is the default when 8051 is powered up

8051 REGISTER BANKS AND STACK

Register Banks (cont')

Register banks and their RAM address

Bank 0		Bank 1		Bank 2		Bank 3	
7	R7	F	R7	17	R7	1F	R7
6	R6	E	R6	16	R6	1E	R6
5	R5	D	R5	15	R5	1D	R5
4	R4	C	R4	14	R4	1C	R4
3	R3	B	R3	13	R3	1B	R3
2	R2	A	R2	12	R2	1A	R2
1	R1	9	R1	11	R1	19	R1
0	R0	8	R0	10	R0	18	R0

8051 REGISTER BANKS AND STACK

Register Banks (cont')

- We can switch to other banks by use of the PSW register
  - Bits D4 and D3 of the PSW are used to select the desired register bank
  - Use the bit-addressable instructions SETB and CLR to access PSW.4 and PSW.3

PSW bank selection

	RS1(PSW.4)	RS0(PSW.3)
Bank 0	0	0
Bank 1	0	1
Bank 2	1	0
Bank 3	1	1

8051 REGISTER BANKS AND STACK

Register Banks (cont')

Example 2-5

```
MOV R0, #99H      ;load R0 with 99H
MOV R1, #85H      ;load R1 with 85H
```

Example 2-6

```
MOV 00, #99H      ;RAM location 00H has 99H
MOV 01, #85H      ;RAM location 01H has 85H
```

Example 2-7

```
SETB PSW.4        ;select bank 2
MOV R0, #99H      ;RAM location 10H has 99H
MOV R1, #85H      ;RAM location 11H has 85H
```

8051 REGISTER BANKS AND STACK

Stack

- The stack is a section of RAM used by the CPU to store information temporarily
  - This information could be data or an address
- The register used to access the stack is called the SP (stack pointer) register
  - The stack pointer in the 8051 is only 8 bit wide, which means that it can take value of 00 to FFH
  - When the 8051 is powered up, the SP register contains value 07
    - RAM location 08 is the first location begin used for the stack by the 8051

8051 REGISTER BANKS AND STACK

Stack (cont')

- The storing of a CPU register in the stack is called a `PUSH`
  - SP is pointing to the last used location of the stack
  - As we push data onto the stack, the SP is incremented by one
    - This is different from many microprocessors
- Loading the contents of the stack back into a CPU register is called a `POP`
  - With every pop, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented once

8051 REGISTER BANKS AND STACK

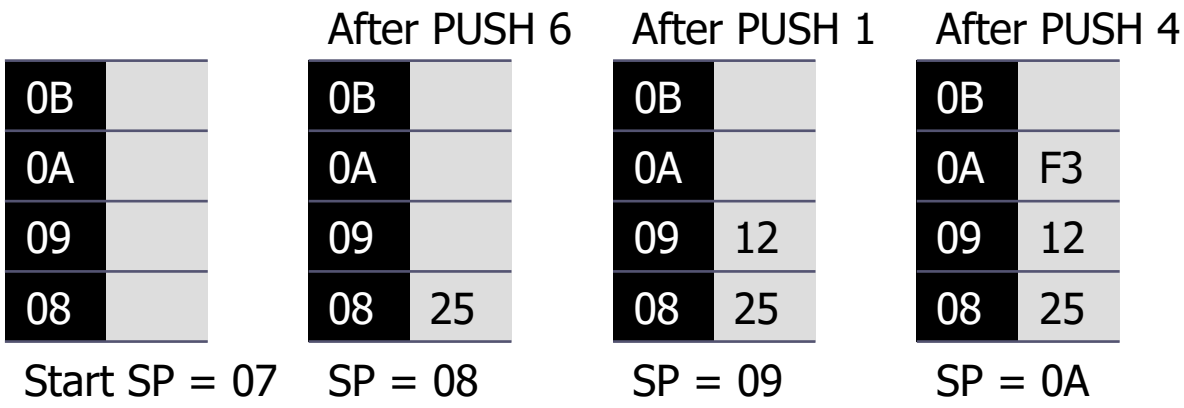
Pushing onto Stack

Example 2-8

Show the stack and stack pointer from the following. Assume the default stack area.

```
MOV  R6,  #25H
MOV  R1,  #12H
MOV R4, #0F3H
PUSH   6
PUSH   1
PUSH   4
```

Solution:



8051 REGISTER BANKS AND STACK

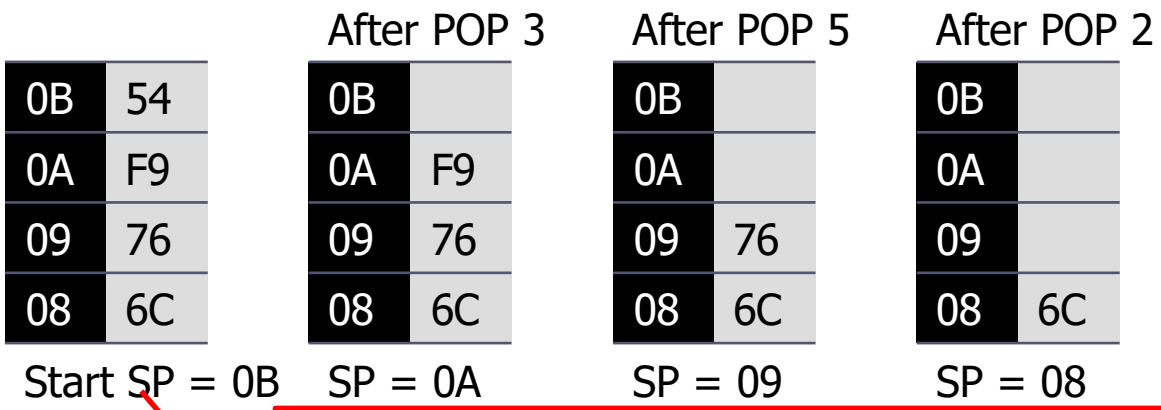
Popping From Stack

Example 2-9

Examining the stack, show the contents of the register and SP after execution of the following instructions. All value are in hex.

```
POP      3          ; POP stack into R3
POP      5          ; POP stack into R5
POP      2          ; POP stack into R2
```

Solution:



Because locations 20-2FH of RAM are reserved for bit-addressable memory, so we can change the SP to other RAM location by using the instruction "`MOV SP, #XX`"

8051 REGISTER BANKS AND STACK

CALL Instruction And Stack

- The CPU also uses the stack to save the address of the instruction just below the `CALL` instruction
  - This is how the CPU knows where to resume when it returns from the called subroutine

8051 REGISTER BANKS AND STACK

Incrementing Stack Pointer

- The reason of incrementing SP after push is
  - Make sure that the stack is growing toward RAM location 7FH, from lower to upper addresses
  - Ensure that the stack will not reach the bottom of RAM and consequently run out of stack space
  - If the stack pointer were decremented after push
    - We would be using RAM locations 7, 6, 5, etc. which belong to R7 to R0 of bank 0, the default register bank

8051 REGISTER BANKS AND STACK

Stack and Bank 1 Conflict

- When 8051 is powered up, register bank 1 and the stack are using the same memory space
  - We can reallocate another section of RAM to the stack

8051 REGISTER BANKS AND STACK

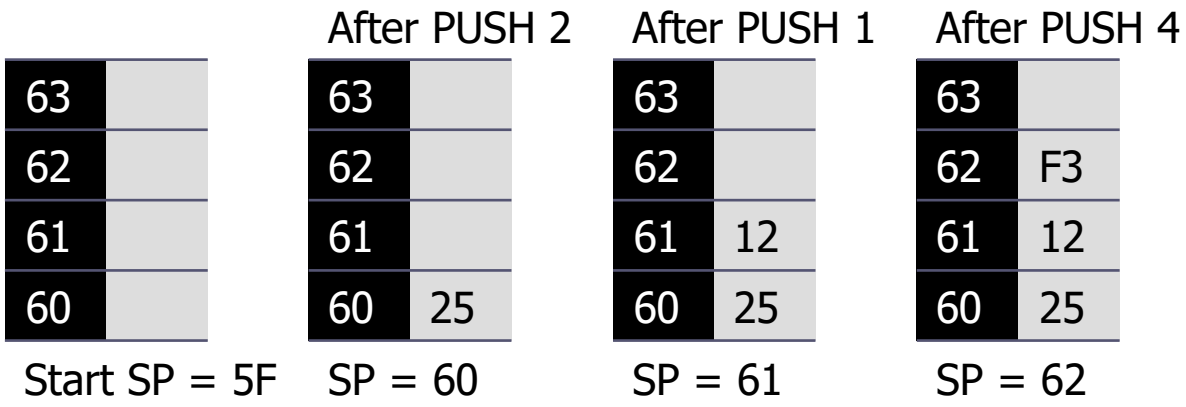
Stack And Bank 1 Conflict (cont')

Example 2-10

Examining the stack, show the contents of the register and SP after execution of the following instructions. All value are in hex.

```
MOV SP, #5FH    ;make RAM location 60H
                ;first stack location
MOV R2, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH  2
PUSH  1
PUSH  4
```

Solution:



ADDRESSING MODES

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

Mazidi, Mazidi and McKinlay

ADDRESSING MODES

- The CPU can access data in various ways, which are called addressing modes
  - Immediate
  - Register
  - Direct
  - Register indirect
  - Indexed



IMMEDIATE ADDRESSING MODE

- The source operand is a constant
  - The immediate data must be preceded by the pound sign, "#"
  - Can load information into any registers, including 16-bit DPTR register
    - DPTR can also be accessed as two 8-bit registers, the high byte DPH and low byte DPL

```
MOV A,#25H        ;load 25H into A
MOV R4,#62        ;load 62 into R4
MOV B,#40H        ;load 40H into B
MOV DPTR,#4521H ;DPTR=4512H
MOV DPL,#21H      ;This is the same
MOV DPH,#45H      ;as above

;illegal!! Value > 65535 (FFFFH)
MOV DPTR,#68975
```

IMMEDIATE ADDRESSING MODE (cont')

- We can use EQU directive to access immediate data

```
        Count   EQU 30
        ...     ...
        MOV     R4,#COUNT          ;R4=1EH
        MOV     DPTR,#MYDATA       ;DPTR=200H

        ORG     200H
MYDATA: DB      "America"
```

- We can also use immediate addressing mode to send data to 8051 ports

```
        MOV P1,#55H
```

REGISTER ADDRESSING MODE

- Use registers to hold the data to be manipulated

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

- The source and destination registers must match in size
  - `MOV DPTR,A` will give an error

```
MOV DPTR,#25F5H
MOV R7,DPL
MOV R6,DPH
```

- The movement of data between Rn registers is not allowed
  - `MOV R4,R7` is invalid

ACCESSING MEMORY

Direct Addressing Mode

- It is most often used the direct addressing mode to access RAM locations 30 – 7FH
  - The entire 128 bytes of RAM can be accessed
  - The register bank locations are accessed by the register names

Direct addressing mode

```
MOV A,4     ;is same as
MOV A,R4    ;which means copy R4 into A
```

- Contrast this with immediate addressing mode
  - There is no "#" sign in the operand

Register addressing mode

```
MOV R0,40H  ;save content of 40H in R0
MOV 56H,A   ;save content of A in 56H
```

ACCESSING MEMORY

SFR Registers and Their Addresses

- The SFR (Special Function Register) can be accessed by their names or by their addresses

```
        MOV 0E0H,#55H    ;is the same as
        MOV A,#55h       ;load 55H into A

        MOV 0F0H,R0      ;is the same as
        MOV B,R0         ;copy R0 into B
```

- The SFR registers have addresses between 80H and FFH
  - Not all the address space of 80 to FF is used by SFR
  - The unused locations 80H to FFH are reserved and must not be used by the 8051 programmer

ACCESSING MEMORY

SFR Registers and Their Addresses (cont')

Special Function Register (SFR) Addresses

Symbol	Name	Address
ACC*	Accumulator	0E0H
B*	B register	0F0H
PSW*	Program status word	0D0H
SP	Stack pointer	81H
DPTR	Data pointer 2 bytes	
DPL	Low byte	82H
DPH	High byte	83H
P0*	Port 0	80H
P1*	Port 1	90H
P2*	Port 2	0A0H
P3*	Port 3	0B0H
IP*	Interrupt priority control	0B8H
IE*	Interrupt enable control	0A8H
…	…	…

ACCESSING MEMORY

SFR Registers and Their Addresses (cont')

Special Function Register (SFR) Addresses

Symbol	Name	Address
TMOD	Timer/counter mode control	89H
TCON*	Timer/counter control	88H
T2CON*	Timer/counter 2 control	0C8H
T2MOD	Timer/counter mode control	OC9H
TH0	Timer/counter 0 high byte	8CH
TL0	Timer/counter 0 low byte	8AH
TH1	Timer/counter 1 high byte	8DH
TL1	Timer/counter 1 low byte	8BH
TH2	Timer/counter 2 high byte	0CDH
TL2	Timer/counter 2 low byte	0CCH
RCAP2H	T/C 2 capture register high byte	0CBH
RCAP2L	T/C 2 capture register low byte	0CAH
SCON*	Serial control	98H
SBUF	Serial data buffer	99H
PCON	Power ontrol	87H

*** Bit addressable**

ACCESSING MEMORY

SFR Registers and Their Addresses (cont')

Example 5-1

Write code to send 55H to ports P1 and P2, using
(a) their names (b) their addresses

Solution :

```
(a)   MOV A,#55H        ;A=55H
      MOV P1,A          ;P1=55H
      MOV P2,A          ;P2=55H
```

(b) From Table 5-1, P1 address=80H; P2 address=A0H

```
      MOV A,#55H        ;A=55H
      MOV 80H,A         ;P1=55H
      MOV 0A0H,A        ;P2=55H
```

ACCESSING MEMORY

Stack and Direct Addressing Mode

- Only direct addressing mode is allowed for pushing or popping the stack
  - `PUSH A` is invalid
  - Pushing the accumulator onto the stack must be coded as `PUSH 0E0H`

Example 5-2

Show the code to push R5 and A onto the stack and then pop them back them into R2 and B, where B = A and R2 = R5

Solution:

```
PUSH 05        ;push R5 onto stack
PUSH 0E0H      ;push register A onto stack
POP  0F0H      ;pop top of stack into B
               ;now register B = register A
POP  02        ;pop top of stack into R2
               ;now R2=R6
```

ACCESSING MEMORY

Register Indirect Addressing Mode

- A register is used as a pointer to the data
  - Only register R0 and R1 are used for this purpose
  - R2 – R7 cannot be used to hold the address of an operand located in RAM
- When R0 and R1 hold the addresses of RAM locations, they must be preceded by the "@" sign

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

ACCESSING MEMORY

Register Indirect Addressing Mode (cont')

Example 5-3

Write a program to copy the value 55H into RAM memory locations 40H to 41H using
(a) direct addressing mode, (b) register indirect addressing mode without a loop, and (c) with a loop

Solution:

```
(a)
    MOV A,#55H  ;load A with value 55H
    MOV 40H,A   ;copy A to RAM location 40H
    MOV 41H.A   ;copy A to RAM location 41H
(b)
    MOV A,#55H  ;load A with value 55H
    MOV R0,#40H ;load the pointer. R0=40H
    MOV @R0,A   ;copy A to RAM R0 points to
    INC R0      ;increment pointer. Now R0=41h
    MOV @R0,A   ;copy A to RAM R0 points to
(c)
        MOV A,#55H    ;A=55H
        MOV R0,#40H   ;load pointer.R0=40H,
        MOV R2,#02    ;load counter, R2=3
AGAIN:  MOV @R0,A     ;copy 55 to RAM R0 points to
        INC R0        ;increment R0 pointer
        DJNZ R2,AGAIN ;loop until counter = zero
```

ACCESSING MEMORY

Register Indirect Addressing Mode (cont')

- The advantage is that it makes accessing data dynamic rather than static as in direct addressing mode
  - Looping is not possible in direct addressing mode

Example 5-4

Write a program to clear 16 RAM locations starting at RAM address 60H

Solution:

```
        CLR A           ;A=0
        MOV R1,#60H     ;load pointer. R1=60H
        MOV R7,#16      ;load counter, R7=16
AGAIN:  MOV @R1,A       ;clear RAM R1 points to
        INC R1          ;increment R1 pointer
        DJNZ R7,AGAIN   ;loop until counter=zero
```

ACCESSING MEMORY

Register Indirect Addressing Mode (cont')

Example 5-5

Write a program to copy a block of 10 bytes of data from 35H to 60H

Solution:

```
      MOV R0,#35H   ;source pointer
      MOV R1,#60H   ;destination pointer
      MOV R3,#10    ;counter
BACK: MOV A,@R0     ;get a byte from source
      MOV @R1,A     ;copy it to destination
      INC R0        ;increment source pointer
      INC R1      ;increment destination pointer
      DJNZ R3,BACK ;keep doing for ten bytes
```

ACCESSING MEMORY

Register Indirect Addressing Mode (cont')

- R0 and R1 are the only registers that can be used for pointers in register indirect addressing mode
- Since R0 and R1 are 8 bits wide, their use is limited to access any information in the internal RAM
- Whether accessing externally connected RAM or on-chip ROM, we need 16-bit pointer
  - In such case, the DPTR register is used

ACCESSING MEMORY

Indexed Addressing Mode and On-chip ROM Access

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM
- The instruction used for this purpose is `MOVC A,@A+DPTR`
  - Use instruction `MOVC`, "C" means code
  - The contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data

ACCESSING MEMORY

Indexed

DPTR=200H, A=0

DPTR=200H, A=55H

DPTR=201H, A=55H

DPTR=201H, A=0

DPTR=201H, A=53H

DPTR=202H, A=53H

202	A
201	S
200	U

Example 5-6

In this program, assume that the word "USA" is burned into ROM locations starting at 200H. And that the program is burned into ROM locations starting at 0. Analyze how the program works and state where "USA" is stored after this program is run.

Solution:

```
        ORG   0000H        ;burn into ROM starting at 0
        MOV   DPTR,#200H   ;DPTR=200H look-up table addr
        CLR   A            ;clear A(A=0)
        MOVC  A,@A+DPTR    ;get the char from code space
        MOV   R0,A         ;save it in R0
        INC   DPTR         ;DPTR=201 point to next char
        CLR   A            ;clear A(A=0)
        MOVC  A,@A+DPTR    ;get the next char
        MOV   R1,A         ;save it in R1
        INC   DPTR         ;DPTR=202 point to next char
        CLR   A            ;clear A(A=0)
        MOVC  A,@A+DPTR    ;get the next char
        MOV   R2,A         ;save it in R2
Here:   SJMP  HERE         ;stay here
;Data is burned into code space starting at 200H

        ORG   200H
MYDATA: DB "USA"
        END                ;end of program
```

R0=55H

R1=53H

R2=41H

ACCESSING MEMORY

Look-up Table (cont')

- The look-up table allows access to elements of a frequently used table with minimum operations

Example 5-8

Write a program to get the x value from P1 and send x^2 to P2, continuously

Solution:

```
      ORG   0
      MOV   DPTR,#300H     ;LOAD TABLE ADDRESS
      MOV   A,#0FFH        ;A=FF
      MOV   P1,A           ;CONFIGURE P1 INPUT PORT
BACK:MOV    A,P1           ;GET X
      MOV   A,@A+DPTR      ;GET X SQAURE FROM TABLE
      MOV   P2,A           ;ISSUE IT TO P2
      SJMP  BACK           ;KEEP DOING IT

      ORG   300H
XSQR_TABLE:
      DB    0,1,4,9,16,25,36,49,64,81
      END
```

ACCESSING MEMORY

Indexed Addressing Mode and MOVX

- In many applications, the size of program code does not leave any room to share the 64K-byte code space with data
  - The 8051 has another 64K bytes of memory space set aside exclusively for data storage
    - This data memory space is referred to as external memory and it is accessed only by the `MOVX` instruction
- The 8051 has a total of 128K bytes of memory space
  - 64K bytes of code and 64K bytes of data
  - The data space cannot be shared between code and data

ACCESSING MEMORY

RAM Locations 30 – 7FH as Scratch Pad

- In many applications we use RAM locations 30 – 7FH as scratch pad
  - We use R0 – R7 of bank 0
  - Leave addresses 8 – 1FH for stack usage
  - If we need more registers, we simply use RAM locations 30 – 7FH

Example 5-10

Write a program to toggle P1 a total of 200 times. Use RAM location 32H to hold your counter value instead of registers R0 – R7

Solution:

```
        MOV     P1,#55H     ;P1=55H
        MOV     32H,#200    ;load counter value
                            ;into RAM loc 32H
LOP1:   CPL     P1          ;toggle P1
        ACALL   DELAY
        DJNZ    32H,LOP1    ;repeat 200 times
```

BIT ADDRESSES

- Many microprocessors allow program to access registers and I/O ports in byte size only
  - However, in many applications we need to check a single bit
- One unique and powerful feature of the 8051 is single-bit operation
  - Single-bit instructions allow the programmer to set, clear, move, and complement individual bits of a port, memory, or register
  - It is registers, RAM, and I/O ports that need to be bit-addressable
    - ROM, holding program code for execution, is not bit-addressable

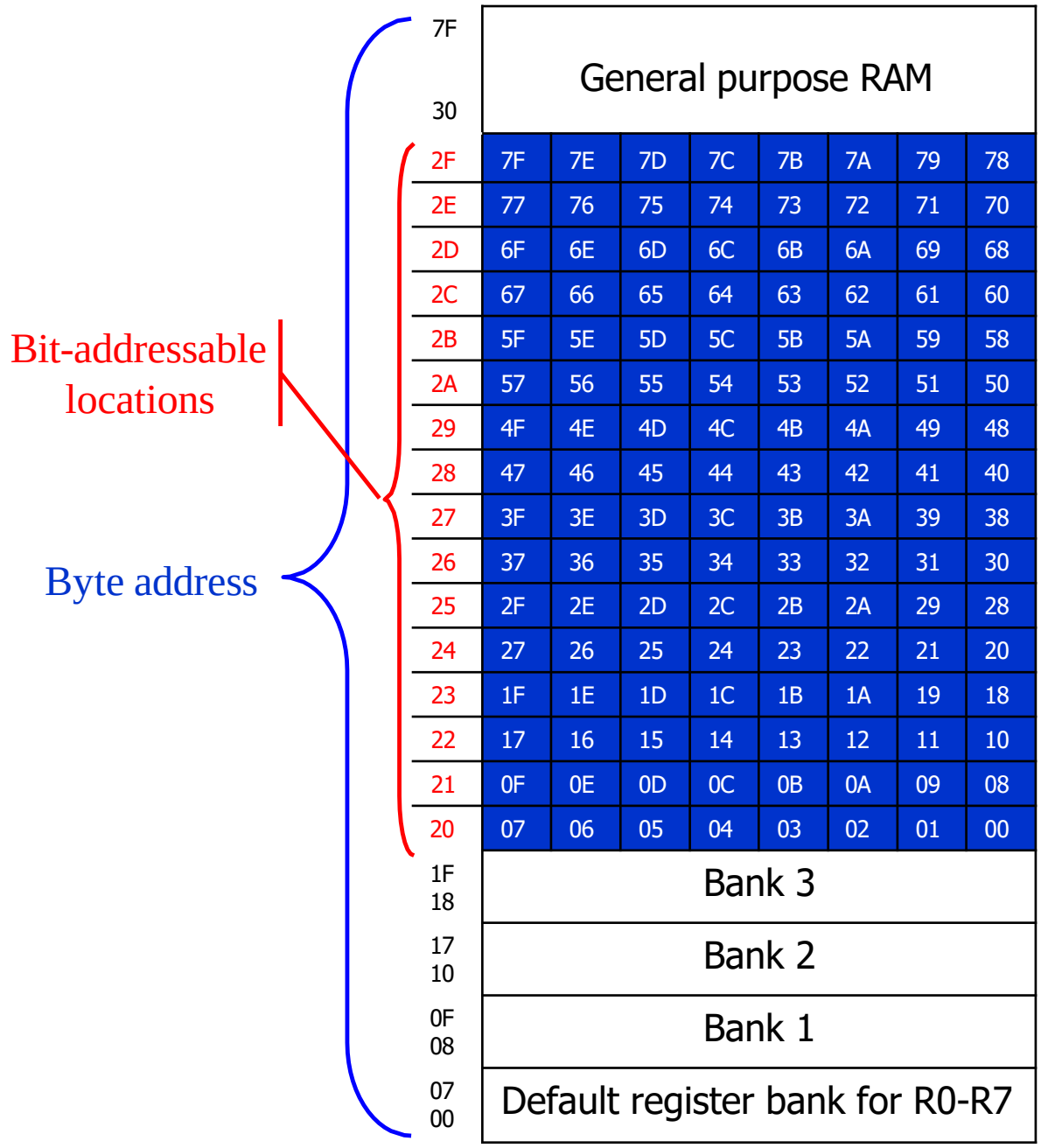
BIT ADDRESSES

Bit-Addressable RAM

- The bit-addressable RAM location are 20H to 2FH
  - These 16 bytes provide 128 bits of RAM bit-addressability, since $16 \times 8 = 128$
    - 0 to 127 (in decimal) or 00 to 7FH
  - The first byte of internal RAM location 20H has bit address 0 to 7H
  - The last byte of 2FH has bit address 78H to 7FH
- Internal RAM locations 20-2FH are both byte-addressable and bit-addressable
  - Bit address 00-7FH belong to RAM byte addresses 20-2FH
  - Bit address 80-F7H belong to SFR P0, P1, …

BIT ADDRESSES

Bit-Addressable RAM (cont')



BIT ADDRESSES

Bit-Addressable RAM (cont')

Example 5-11

Find out to which by each of the following bits belongs. Give the address of the RAM byte in hex

(a) `SETB 42H`, (b) `CLR 67H`, (c) `CLR 0FH`
(d) `SETB 28H`, (e) `CLR 12`, (f) `SETB 05`

Solution:

(a) D2 of RAM location 28H

(b) D7 of RAM location 2CH

(c) D7 of RAM location 21H

(d) D0 of RAM location 25H

(e) D4 of RAM location 21H

(f) D5 of RAM location 20H

	D7	D6	D5	D4	D3	D2	D1	D0
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00

BIT ADDRESSES

Bit-Addressable RAM (cont')

- To avoid confusion regarding the addresses 00 – 7FH
  - The 128 bytes of RAM have the byte addresses of 00 – 7FH can be accessed in byte size using various addressing modes
    - Direct and register-indirect
  - The 16 bytes of RAM locations 20 – 2FH have bit address of 00 – 7FH
    - We can use only the single-bit instructions and these instructions use only direct addressing mode

BIT ADDRESSES

Bit-Addressable RAM (cont')

- Instructions that are used for signal-bit operations are as following

Single-Bit Instructions

Instruction	Function
SETB bit	Set the bit (bit = 1)
CLR bit	Clear the bit (bit = 0)
CPL bit	Complement the bit (bit = NOT bit)
JB bit, target	Jump to target if bit = 1 (jump if bit)
JNB bit, target	Jump to target if bit = 0 (jump if no bit)
JBC bit, target	Jump to target if bit = 1, clear bit (jump if bit, then clear)

BIT ADDRESSES

I/O Port Bit Addresses

- While all of the SFR registers are byte-addressable, some of them are also bit-addressable
  - The P0 – P3 are bit addressable
- We can access either the entire 8 bits or any single bit of I/O ports P0, P1, P2, and P3 without altering the rest
- When accessing a port in a single-bit manner, we use the syntax `SETB X.Y`
  - X is the port number P0, P1, P2, or P3
  - Y is the desired bit number from 0 to 7 for data bits D0 to D7
  - ex. `SETB P1.5` sets bit 5 of port 1 high

BIT ADDRESSES

I/O Port Bit Addresses (cont')

- Notice that when code such as `SETB P1.0` is assembled, it becomes `SETB 90H`
  - The bit address for I/O ports
    - P0 are 80H to 87H
    - P1 are 90H to 97H
    - P2 are A0H to A7H
    - P3 are B0H to B7H

Single-Bit Addressability of Ports

P0	P1	P2	P3	Port Bit
P0.0 (80)	P1.0 (90)	P2.0 (A0)	P3.0 (B0)	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7 (87)	P1.7 (97)	P2.7 (A7)	P3.7 (B7)	D7

BIT ADDRESSES

I/O Port Bit Addresses (cont')

SFR RAM Address (Byte and Bit)

Bit addresses 80 – F7H belong to SFR of P0, TCON, P1, SCON, P2, etc

Byte address	Bit address		
FF			
F0	F7 F6 F5 F4 F3 F2 F1 F0	B	
E0	E7 E6 E5 E4 E3 E2 E1 E0	ACC	
D0	D7 D6 D5 D4 D3 D2 D1 D0	PSW	
B8	-- -- -- BC BB BA B9 B8	IP	
B0	B7 B6 B5 B4 B3 B2 B1 B0	P3	
A8	AF AE AD AC AB AA A9 A8	IE	
A0	A7 A6 A5 A4 A3 A2 A1 A0	P2	
99	not bit addressable	SBUF	

Byte address	Bit address	
98	9F 9E 9D 9C 9B 9A 99 98	SCON
90	97 96 95 94 93 92 91 90	P1
8D	not bit addressable	TH1
8C	not bit addressable	TH0
8B	not bit addressable	TL1
8A	not bit addressable	TL0
89	not bit addressable	TMOD
88	8F 8E 8D 8C 8B 8A 89 88	TCON
87	not bit addressable	PCON
83	not bit addressable	DPH
82	not bit addressable	DPL
81	not bit addressable	SP
80	87 86 85 84 83 82 81 80	P0

Special Function Register

BIT ADDRESSES

Registers Bit-Addressability

- Only registers A, B, PSW, IP, IE, ACC, SCON, and TCON are bit-addressable
  - While all I/O ports are bit-addressable
- In PSW register, two bits are set aside for the selection of the register banks
  - Upon RESET, bank 0 is selected
  - We can select any other banks using the bit-addressability of the PSW

CY	AC	--	RS1	RS0	OV	--	P

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

BIT ADDRESSES

Registers Bit-Addressability (cont')

Example 5-13

Write a program to save the accumulator in R7 of bank 2.

Solution:

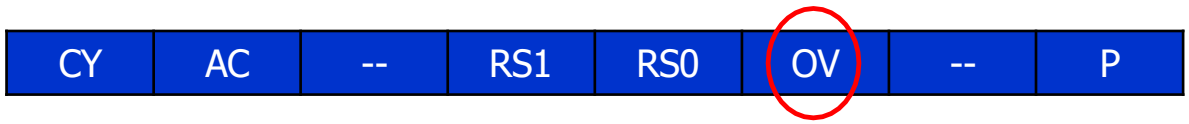
```
        CLR     PSW.3
        SETB    PSW.4
        MOV     R7,A
```

Example 5-14

While there are instructions such as JNC and JC to check the carry flag bit (CY), there are no such instructions for the overflow flag bit (OV). How would you write code to check OV?

Solution:

```
        JB      PSW.2,TARGET    ;jump if OV=1
```



Example 5-18

While a program to save the status of bit P1.7 on RAM address bit 05.

Solution:

```
        MOV     C,P1.7
        MOV     05,C
```

BIT ADDRESSES

Registers Bit-Addressability (cont')

Example 5-15

Write a program to see if the RAM location 37H contains an even value. If so, send it to P2. If not, make it even and then send it to P2.

Solution:

```
        MOV     A,37H       ;load RAM 37H into ACC
        JNB     ACC.0,YES   ;if D0 of ACC 0? If so jump
        INC     A           ;it's odd, make it even
YES:    MOV     P2,A        ;send it to P2
```

Example 5-17

The status of bits P1.2 and P1.3 of I/O port P1 must be saved before they are changed. Write a program to save the status of P1.2 in bit location 06 and the status of P1.3 in bit location 07

Solution:

```
        CLR     06          ;clear bit addr. 06
        CLR     07          ;clear bit addr. 07
        JNB     P1.2,OVER   ;check P1.2, if 0 then jump
        SETB    06          ;if P1.2=1,set bit 06 to 1
OVER:   JNB     P1.3,NEXT   ;check P1.3, if 0 then jump
        SETB    07          ;if P1.3=1,set bit 07 to 1
NEXT:  ...
```

BIT ADDRESSES

Using BIT

- The `BIT` directive is a widely used directive to assign the bit-addressable I/O and RAM locations
  - Allow a program to assign the I/O or RAM bit at the beginning of the program, making it easier to modify them

Example 5-22

A switch is connected to pin P1.7 and an LED to pin P2.0. Write a program to get the status of the switch and send it to the LED.

Solution:

```
LED     BIT     P1.7    ;assign bit
SW      BIT     P2.0    ;assign bit
HERE:   MOV     C,SW    ;get the bit from the port
        MOV     LED,C   ;send the bit to the port
        SJMP    HERE    ;repeat forever
```

BIT ADDRESSES

Using BIT (cont')

Example 5-20

Assume that bit P2.3 is an input and represents the condition of an oven. If it goes high, it means that the oven is hot. Monitor the bit continuously. Whenever it goes high, send a high-to-low pulse to port P1.5 to turn on a buzzer.

Solution:

```
OVEN_HOT  BIT P2.3
BUZZER    BIT P1.5
HERE:   JNB     OVEN_HOT,HERE ;keep monitoring
        ACALL   DELAY
        CPL     BUZZER   ;sound the buzzer
        ACALL   DELAY
        SJMP    HERE
```

BIT ADDRESSES

Using EQU

- Use the `EQU` to assign addresses
  - Defined by names, like P1.7 or P2
  - Defined by addresses, like 97H or 0A0H

Example 5-24

A switch is connected to pin P1.7. Write a program to check the status of the switch and make the following decision.
(a) If SW = 0, send "0" to P2
(b) If SW = 1, send "1" to P2

Solution:

```
SW      EQU P1.7
MYDATA  EQU P2
HERE:   MOV     C,SW
        JC      OVER
        MOV     MYDATA,#'0'
        SJMP    HERE
OVER:   MOV     MYDATA,#'1'
        SJMP    HERE
        END
```

```
SW      EQU 97H
MYDATA  EQU 0A0H
```

EXTRA 128 BYTE ON-CHIP RAM IN 8052

- The 8052 has another 128 bytes of on-chip RAM with addresses 80 – FFH
  - It is often called upper memory
    - Use indirect addressing mode, which uses R0 and R1 registers as pointers with values of 80H or higher
      - `MOV @R0, A` and `MOV @R1, A`
  - The same address space assigned to the SFRs
    - Use direct addressing mode
      - `MOV 90H, #55H` is the same as

        `MOV P1, #55H`

EXTRA 128 BYTE ON-CHIP RAM IN 8052 (cont')

Example 5-27

Assume that the on-chip ROM has a message. Write a program to copy it from code space into the upper memory space starting at address 80H. Also, as you place a byte in upper RAM, give a copy to P0.

Solution:

```
        ORG     0
        MOV     DPTR,#MYDATA
        MOV     R1,#80H         ;access the upper memory
B1:     CLR     A
        MOVC    A,@A+DPTR       ;copy from code ROM
        MOV     @R1,A           ;store in upper memory
        MOV     P0,A            ;give a copy to P0
        JZ      EXIT            ;exit if last byte
        INC     DPTR            ;increment DPTR
        INC     R1              ;increment R1
        SJMP    B1              ;repeat until last byte
EXIT:   SJMP    $               ;stay here when finished
;---------------
        ORG     300H
MYDATA: DB      "The Promise of World Peace",0
        END
```

JUMP, LOOP AND CALL INSTRUCTIONS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

Mazidi, Mazidi and McKinlay

LOOP AND JUMP INSTRUCTIONS

Looping

- Repeating a sequence of instructions a certain number of times is called a loop
  - Loop action is performed by
    `DJNZ reg, Label`
    - The register is decremented
    - If it is not zero, it jumps to the target address referred to by the label
    - Prior to the start of loop the register is loaded with the counter for the number of repetitions
    - Counter can be R0 – R7 or RAM location

A loop can be repeated a maximum of 255 times, if R2 is FFH

```
;This program adds value 3 to the ACC ten times
       MOV  A,#0      ;A=0, clear ACC
       MOV  R2,#10    ;load counter R2=10
AGAIN: ADD  A,#03     ;add 03 to ACC
       DJNZ R2,AGAIN  ;repeat until R2=0,10 times
       MOV  R5,A      ;save A in R5
```

LOOP AND JUMP INSTRUCTIONS

Nested Loop

- If we want to repeat an action more times than 256, we use a loop inside a loop, which is called nested loop
  - We use multiple registers to hold the count

Write a program to (a) load the accumulator with the value 55H, and (b) complement the ACC 700 times

```
        MOV   A,#55H    ;A=55H
        MOV   R3,#10    ;R3=10, outer loop count
NEXT:   MOV   R2,#70    ;R2=70, inner loop count
AGAIN:  CPL   A         ;complement A register
        DJNZ  R2,AGAIN  ;repeat it 70 times
        DJNZ  R3,NEXT
```

LOOP AND JUMP INSTRUCTIONS

Conditional Jumps

- Jump only if a certain condition is met

JZ label ;jump if A=0

```
        MOV   A,R0      ;A=R0
        JZ    OVER      ;jump if A = 0
        MOV   A,R1      ;A=R1
        JZ    OVER      ;jump if A = 0
        ...
OVER:
```

Can be used only for register A, not any other register

Determine if R5 contains the value 0. If so, put 55H in it.

```
        MOV   A,R5      ;copy R5 to A
        JNZ   NEXT      ;jump if A is not zero
        MOV   R5,#55H
NEXT:   ...
```

LOOP AND JUMP INSTRUCTIONS

Conditional Jumps (cont')

- (cont')

JNC label ;jump if no carry, CY=0

  - If CY = 0, the CPU starts to fetch and execute instruction from the address of the label
  - If CY = 1, it will not jump but will execute the next instruction below JNC

Find the sum of the values 79H, F5H, E2H. Put the sum in registers R0 (low byte) and R5 (high byte).

```
        MOV   A,#0       ;A=0
        MOV     R5,A     ;clear R5
        ADD   A,#79H     ;A=0+79H=79H
;       JNC   N_1        ;if CY=0, add next number
;       INC   R5         ;if CY=1, increment R5
N_1:    ADD    A,#0F5H   ;A=79+F5=6E and CY=1
        JNC   N_2        ;jump if CY=0
        INC   R5        ;if CY=1,increment R5 (R5=1)
N_2:    ADD    A,#0E2H   ;A=6E+E2=50 and CY=1
        JNC   OVER       ;jump if CY=0
        INC   R5         ;if CY=1, increment 5
OVER:   MOV   R0,A       ;now R0=50H, and R5=02
```

MOV R5,#0

LOOP AND JUMP INSTRUCTIONS

Conditional Jumps (cont')

8051 conditional jump instructions

Instructions	Actions
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and Jump if A ≠ 0
CJNE A,byte	Jump if A ≠ byte
CJNE reg,#data	Jump if byte ≠ #data
JC	Jump if CY = 1
JNC	Jump if CY = 0
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

- All conditional jumps are short jumps
  - The address of the target must within -128 to +127 bytes of the contents of PC

LOOP AND JUMP INSTRUCTIONS

Unconditional Jumps

- The unconditional jump is a jump in which control is transferred unconditionally to the target location

LJMP (long jump)

- 3-byte instruction
  - First byte is the opcode
  - Second and third bytes represent the 16-bit target address
    - Any memory location from 0000 to FFFFH

SJMP (short jump)

- 2-byte instruction
  - First byte is the opcode
  - Second byte is the relative target address
    - 00 to FFH (forward +127 and backward -128 bytes from the current PC)

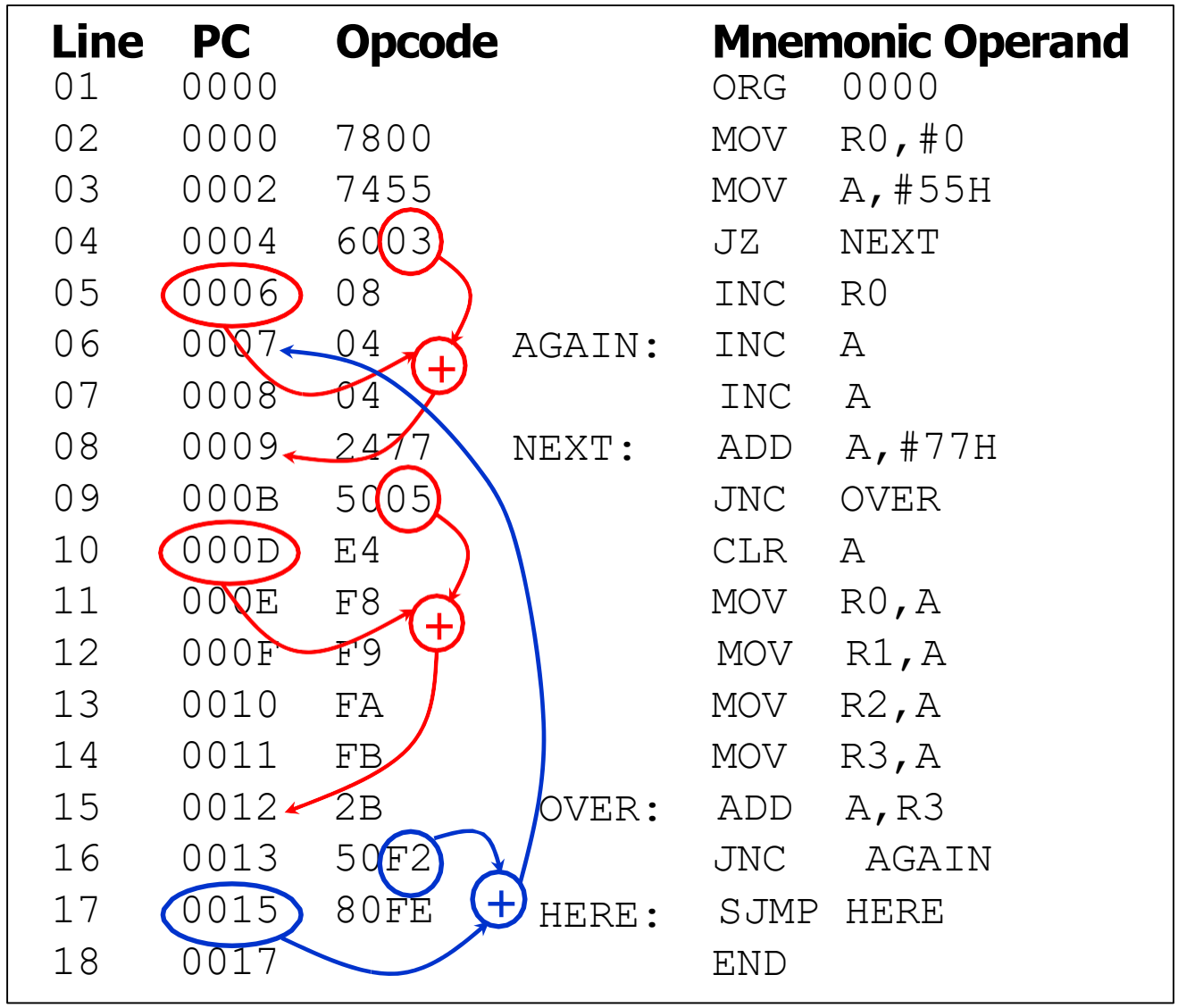
LOOP AND JUMP INSTRUCTIONS

Calculating Short Jump Address

- To calculate the target address of a short jump (`SJMP, JNC, JZ, DJNZ,` etc.)
  - The second byte is added to the PC of the instruction immediately below the jump
- If the target address is more than -128 to +127 bytes from the address below the short jump instruction
  - The assembler will generate an error stating the jump is out of range

LOOP AND JUMP INSTRUCTIONS

Calculating Short Jump Address (cont')



Line	PC	Opcode		Mnemonic	Operand
01	0000			ORG	0000
02	0000	7800		MOV	R0,#0
03	0002	7455		MOV	A,#55H
04	0004	6003		JZ	NEXT
05	0006	08		INC	R0
06	0007	04	AGAIN:	INC	A
07	0008	04		INC	A
08	0009	2477	NEXT:	ADD	A,#77H
09	000B	5005		JNC	OVER
10	000D	E4		CLR	A
11	000E	F8		MOV	R0,A
12	000F	F9		MOV	R1,A
13	0010	FA		MOV	R2,A
14	0011	FB		MOV	R3,A
15	0012	2B	OVER:	ADD	A,R3
16	0013	50F2		JNC	AGAIN
17	0015	80FE	HERE:	SJMP	HERE
18	0017			END	

CALL INSTRUCTIONS

- Call instruction is used to call subroutine
  - Subroutines are often used to perform tasks that need to be performed frequently
  - This makes a program more structured in addition to saving memory space

LCALL (long call)

- 3-byte instruction
  - First byte is the opcode
  - Second and third bytes are used for address of target subroutine
    - Subroutine is located anywhere within 64K byte address space

ACALL (absolute call)

- 2-byte instruction
  - 11 bits are used for address within 2K-byte range

CALL INSTRUCTIONS

LCALL

- When a subroutine is called, control is transferred to that subroutine, the processor
  - Saves on the stack the the address of the instruction immediately below the LCALL
  - Begins to fetch instructions form the new location
- After finishing execution of the subroutine
  - The instruction RET transfers control back to the caller
    - Every subroutine needs RET as the last instruction

CALL INSTRUCTIONS

LCALL (cont')

```
        ORG    0
BACK:   MOV    A,#55H     ;load A with 55H
        MOV    P1,A       ;send 55H to port 1
        LCALL  DELAY      ;time delay
        MOV    A,#0AAH    ;load A with AA (in hex)
        MOV    P1,A       ;send AAH to port 1
        LCALL  DELAY
        SJMP   BACK       ;keep doing this indefinitely


;---------- this is delay subroutine ------------
        ORG    300H       ;put DELAY at address 300H
DELAY:  MOV    R5,#0FFH   ;R5=255 (FF in hex), counter
AGAIN:  DJNZ   R5,AGAIN   ;stay here until R5 become 0
        RET               ;return to caller (when R5 =0)
        END
```

Upon executing "LCALL DELAY", the address of instruction below it, "MOV A,#0AAH" is pushed onto stack, and the 8051 starts to execute at 300H.

The counter R5 is set to FFH; so loop is repeated 255 times.

When R5 becomes 0, control falls to the RET which pops the address from the stack into the PC and resumes executing the instructions after the CALL.

The amount of time delay depends on the frequency of the 8051

CALL INSTRUCTIONS

CALL Instruction and Stack

```
001 0000                ORG   0
002 0000 7455    BACK:  MOV   A,#55H   ;load A with 55H
003 0002 F590           MOV   P1,A     ;send 55H to p1
004 0004 120300         LCALL DELAY    ;time delay
005 0007 74AA           MOV   A,#0AAH ;load A with AAH
006 0009 F590           MOV   P1,A     ;send AAH to p1
007 000B 120300         LCALL DELAY
008 000E 80F0           SJMP  BACK     ;keep doing this
009 0010
010 0010 ;-------this is the delay subroutine------
011 0300                ORG   300H
012 0300         DELAY:
013 0300 7DFF           MOV   R5,#0FFH ;R5=255
14. 0302 DDFE    AGAIN: DJNZ R5,AGAIN ;stay here
15. 0304 22             RET           ;return to caller
16. 0305                END            ;end of asm file
```

Stack frame after the first LCALL

0A	
09	00
08	07

SP = 09

Low byte goes first and high byte is last

CALL INSTRUCTIONS

Use PUSH/POP in Subroutine

Normally, the number of `PUSH` and `POP` instructions must always match in any called subroutine

```
01 0000                    ORG   0
02 0000 7455     BACK:     MOV   A,#55H   ;load A with 55H
03 0002 F590               MOV   P1,A     ;send 55H to p1
04 0004 7C99               MOV   R4,#99H
05 0006 7D67               MOV   R5,#67H
06 0008 120300             LCALL DELAY    ;time delay
07 000B 74AA               MOV   A,#0AAH  ;load A with AA
08 000D F590               MOV   P1,A     ;send AAH to p1
09 000F 120300             LCALL DELAY
10 0012 80EC               SJMP  BACK     ;keeping doing
   this
11 0014 ;-------this is the delay subroutine------
12 0300                    ORG   300H
13 0300 C004     DELAY:    PUSH  4        ;push R4
14 0302 C005               PUSH  5        ;push R5
   0304 7CFF               MOV   R4,#0FFH;R4=FFH
   0306 7DFF     NEXT:     MOV   R5,#0FFH;R5=FFH
   0308 DDFE     AGAIN:    DJNZ  R5,AGAIN
   030A DCFA               DJNZ  R4,NEXT
   030C D005               POP   5        ;POP into R5
   030E D004               POP   4        ;POP into R4
   0310 22                 RET            ;return to caller
22 031                                    e
```

After first LCALL			After PUSH 4			After PUSH 5		
0B			0B			0B	67	R5
0A			0A	99	R4	0A	99	R4
09	00	PCH	09	00	PCH	09	00	PCH
08	0B	PCL	08	0B	PCL	08	0B	PCL

CALL INSTRUCTIONS

Calling Subroutines

```
;MAIN program calling subroutines
        ORG   0
MAIN:   LCALL           SUBR_1
        LCALL           SUBR_2
        LCALL           SUBR_3

HERE:   SJMP            HERE
;-----------end of MAIN

SUBR_1: ...
        ...
        RET
;-----------end of subroutine1

SUBR_2: ...
        ...
        RET
;-----------end of subroutine2

SUBR_3: ...
        ...
        RET
;-----------end of subroutine3
        END             ;end of the asm file
```

It is common to have one main program and many subroutines that are called from the main program

This allows you to make each subroutine into a separate module

- Each module can be tested separately and then brought together with main program
- In a large program, the module can be assigned to different programmers

CALL INSTRUCTIONS

ACALL

- The only difference between `ACALL` and `LCALL` is
  - The target address for `LCALL` can be anywhere within the 64K byte address
  - The target address of `ACALL` must be within a 2K-byte range
- The use of `ACALL` instead of `LCALL` can save a number of bytes of program ROM space

CALL INSTRUCTIONS

ACALL (cont')

```
        ORG    0
BACK:   MOV    A,#55H     ;load A with 55H
        MOV    P1,A       ;send 55H to port 1
        LCALL  DELAY      ;time delay
        MOV    A,#0AAH    ;load A with AA (in hex)
        MOV    P1,A       ;send AAH to port 1
        LCALL  DELAY
        SJMP   BACK       ;keep doing this indefinitely
         ...
        END               ;end of asm file
```

A rewritten program which is more efficiently

```
        ORG    0
        MOV    A,#55H     ;load A with 55H
BACK:   MOV    P1,A       ;send 55H to port 1
        ACALL  DELAY      ;time delay
        CPL    A          ;complement reg A
        SJMP   BACK       ;keep doing this indefinitely
         ...
        END               ;end of asm file
```

TIME DELAY FOR VARIOUS 8051 CHIPS

- CPU executing an instruction takes a certain number of clock cycles
  - These are referred as to as machine cycles
- The length of machine cycle depends on the frequency of the crystal oscillator connected to 8051
- In original 8051, one machine cycle lasts 12 oscillator periods

Find the period of the machine cycle for 11.0592 MHz crystal frequency

Solution:

```
11.0592/12 = 921.6 kHz;
   machine cycle is 1/921.6 kHz = 1.085µs
```

TIME DELAY FOR VARIOUS 8051 CHIPS (cont')

For 8051 system of 11.0592 MHz, find how long it takes to execute each instruction.

```
(a) MOV R3,#55 (b) DEC R3 (c) DJNZ R2 target
(d) LJMP (e) SJMP (f) NOP (g) MUL AB
```

Solution:

	Machine cycles	***Time to execute***
(a)	1	1x1.085μs = 1.085μs
(b)	1	1x1.085μs = 1.085μs
(c)	2	2x1.085μs = 2.17μs
(d)	2	2x1.085μs = 2.17μs
(e)	2	2x1.085μs = 2.17μs
(f)	1	1x1.085μs = 1.085μs
(g)	4	4x1.085μs = 4.34μs

TIME DELAY FOR VARIOUS 8051 CHIPS

Delay Calculation

Find the size of the delay in following program, if the crystal frequency is 11.0592MHz.

```
        MOV   A,#55H
AGAIN:  MOV   P1,A
        ACALL DELAY
        CPL   A
        SJMP  AGAIN
;---time delay-------
DELAY:  MOV   R3,#200
HERE:   DJNZ  R3,HERE
        RET
```

A simple way to short jump to itself in order to keep the microcontroller busy

```
HERE: SJMP HERE
```

We can use the following:

```
SJMP $
```

Solution:

			Machine cycle
DELAY:	MOV	R3,#200	1
HERE:	DJNZ	R3,HERE	2
	RET		2

Therefore, [(200x2)+1+2]x1.085µs = 436.255µs.

TIME DELAY FOR VARIOUS 8051 CHIPS

Increasing Delay Using NOP

Find the size of the delay in following program, if the crystal frequency is 11.0592MHz.

```
                          Machine Cycle
DELAY: MOV  R3,#250          1
HERE:  NOP                   1
       NOP                   1
       NOP                   1
       NOP                   1
       DJNZ R3,HERE          2
       RET                   2
```

Solution:

The time delay inside HERE loop is

[250(1+1+1+1+2)]x1.085μs = 1627.5μs.

Adding the two instructions outside loop we have 1627.5μs + 3 x 1.085μs = 1630.755μs

TIME DELAY FOR VARIOUS 8051 CHIPS

Large Delay Using Nested Loop

Find the size of the delay in following program, if the crystal frequency is 11.0592MHz.

```
                          Machine Cycle
DELAY: MOV   R2,#200          1
AGAIN: MOV   R3,#250          1
HERE:  NOP                    1
       NOP                    1
       DJNZ  R3,HERE          2
       DJNZ  R2,AGAIN         2
       RET                    2
```

Notice in nested loop, as in all other time delay loops, the time is approximate since we have ignored the first and last instructions in the subroutine.

Solution:

For HERE loop, we have (4x250)x1.085μs=1085μs. For AGAIN loop repeats HERE loop 200 times, so we have 200x1085μs=217000μs. But "MOV R3,#250" and "DJNZ R2,AGAIN" at the start and end of the AGAIN loop add (3x200x1.805)=651μs. As a result we have 217000+651=217651μs.

TIME DELAY FOR VARIOUS 8051 CHIPS

Delay Calculation for Other 8051

- Two factors can affect the accuracy of the delay
  - Crystal frequency
    - The duration of the clock period of the machine cycle is a function of this crystal frequency
  - 8051 design
    - The original machine cycle duration was set at 12 clocks
    - Advances in both IC technology and CPU design in recent years have made the 1-clock machine cycle a common feature

Clocks per machine cycle for various 8051 versions

Chip/Maker	Clocks per Machine Cycle
AT89C51 Atmel	12
P89C54X2 Philips	6
DS5000 Dallas Semi	4
DS89C420/30/40/50 Dallas Semi	1

TIME DELAY FOR VARIOUS 8051 CHIPS

Delay Calculation for Other 8051 (cont')

Find the period of the machine cycle (MC) for various versions of 8051, if XTAL=11.0592 MHz.
(a) AT89C51 (b) P89C54X2 (c) DS5000 (d) DS89C4x0

Solution:

```
(a) 11.0592MHz/12 = 921.6kHz;
    MC is 1/921.6kHz = 1.085µs =  1085ns
(b) 11.0592MHz/6 = 1.8432MHz;
    MC is 1/1.8432MHz = 0.5425µs  = 542ns
(c) 11.0592MHz/4 = 2.7648MHz ;
    MC is 1/2.7648MHz = 0.36µs =  360ns
(d) 11.0592MHz/1 = 11.0592MHz;
    MC is 1/11.0592MHz = 0.0904µs =  90ns
```

TIME DELAY FOR VARIOUS 8051 CHIPS

Delay Calculation for Other 8051 (cont')

Instruction	8051	DSC89C4x0
MOV R3,#55	1	2
DEC R3	1	1
DJNZ R2 target	2	4
LJMP	2	3
SJMP	2	3
NOP	1	1
MUL AB	4	9

For an AT8051 and DSC89C4x0 system of 11.0592 MHz, find how long it takes to execute each instruction.

(a) MOV R3,#55 (b) DEC R3 (c) DJNZ R2 target
(d) LJMP (e) SJMP (f) NOP (g) MUL AB

Solution:

	AT8051	***DS89C4x0***
(a)	1×1085ns = 1085ns	2×90ns = 180ns
(b)	1×1085ns = 1085ns	1×90ns = 90ns
(c)	2×1085ns = 2170ns	4×90ns = 360ns
(d)	2×1085ns = 2170ns	3×90ns = 270ns
(e)	2×1085ns = 2170ns	3×90ns = 270ns
(f)	1×1085ns = 1085ns	1×90ns = 90ns
(g)	4×1085ns = 4340ns	9×90ns = 810ns

I/O PORT PROGRAMMING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

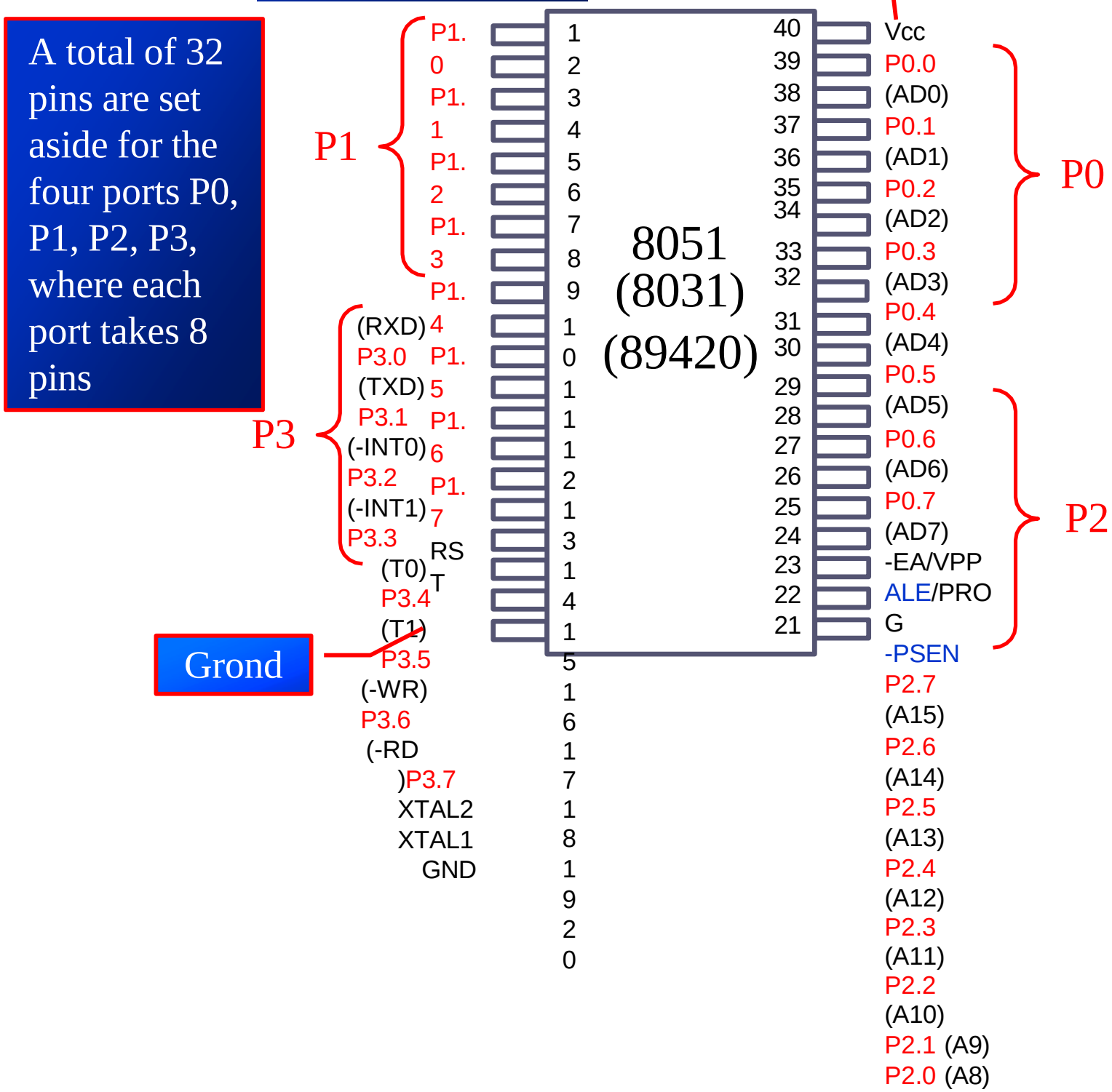
Mazidi, Mazidi and McKinlay

I/O PROGRAMMING

8051 Pin Diagram

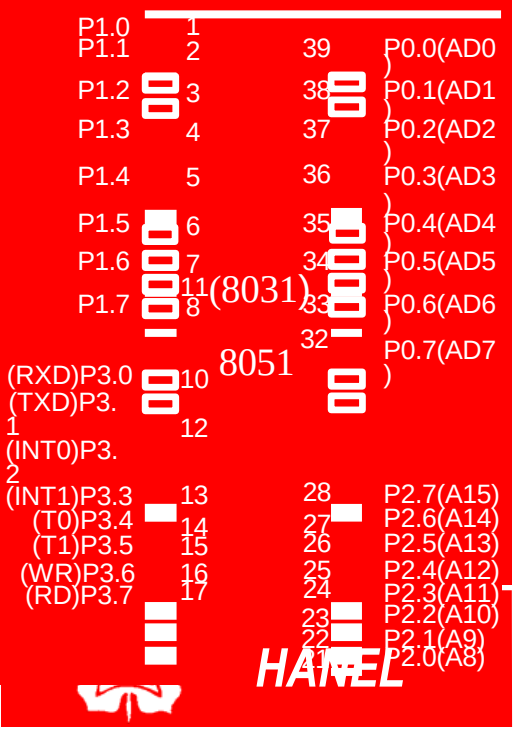
Provides +5V supply voltage to the chip

A total of 32 pins are set aside for the four ports P0, P1, P2, P3, where each port takes 8 pins



I/O PROGRAMMING

I/O Port Pins

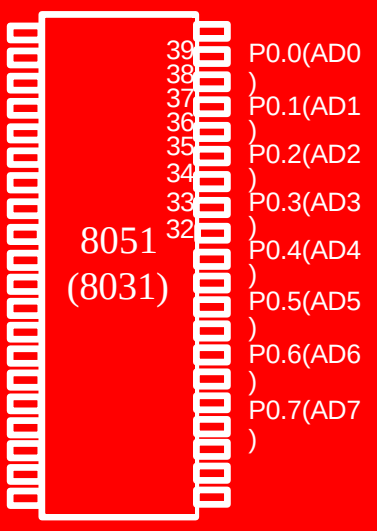


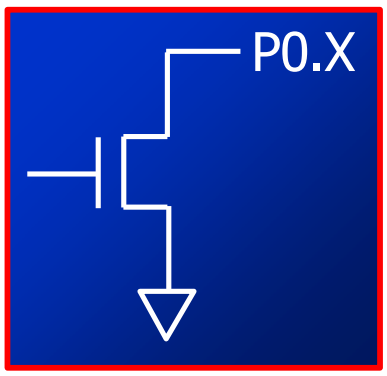
- The four 8-bit I/O ports P0, P1, P2 and P3 each uses 8 pins
- All the ports upon RESET are configured as input, ready to be used as input ports
  - When the first 0 is written to a port, it becomes an output
  - To reconfigure it as an input, a 1 must be sent to the port
    - To use any of these ports as an input port, it must be programmed

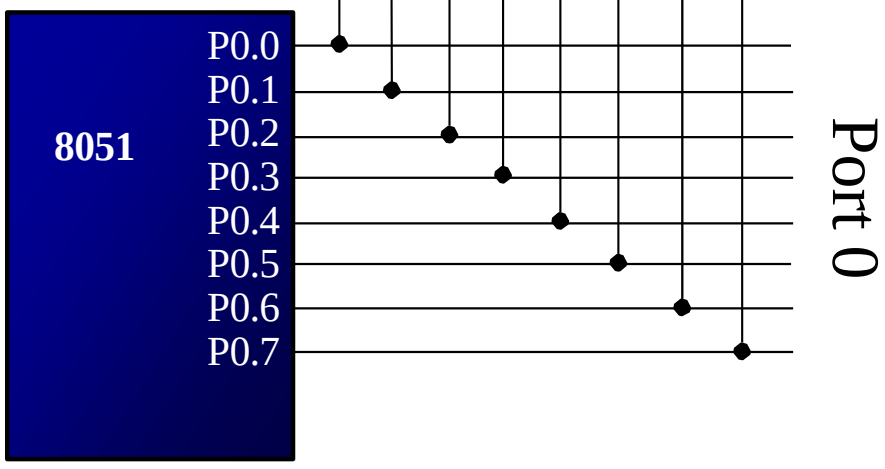
I/O PROGRAMMING

Port 0

- It can be used for input or output, each pin must be connected externally to a 10K ohm pull-up resistor
  - This is due to the fact that P0 is an open drain, unlike P1, P2, and P3
    - Open drain is a term used for MOS chips in the same way that open collector is used for TTL chips





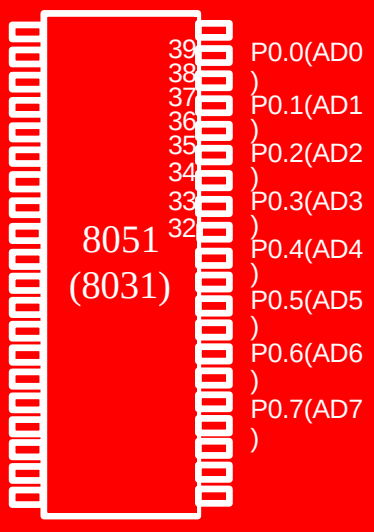


I/O PROGRAMMING

Port 0 (cont')

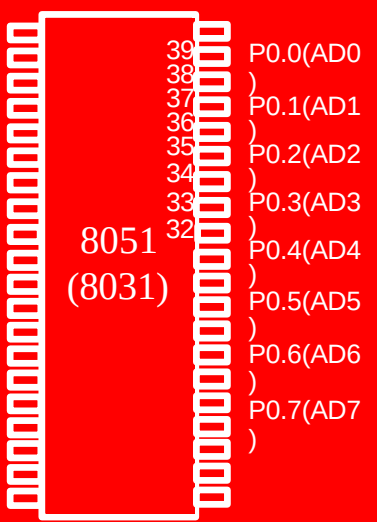
The following code will continuously send out to port 0 the alternating value 55H and AAH

```
BACK:   MOV     A,#55H
        MOV     P0,A
        ACALL   DELAY
        MOV     A,#0AAH
        MOV     P0,A
        ACALL   DELAY
        SJMP    BACK
```



I/O PROGRAMMING

Port 0 as Input



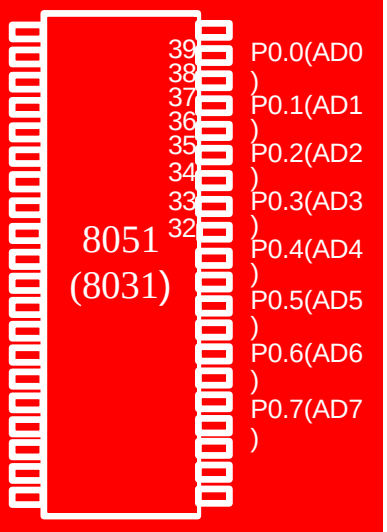
- In order to make port 0 an input, the port must be programmed by writing 1 to all the bits

Port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1

```
        MOV     A,#0FFH         ;A=FF hex
        MOV     P0,A            ;make P0 an i/p port
                                ;by writing it all 1s
BACK:   MOV     A,P0            ;get data from P0
        MOV     P1,A            ;send it to port 1
        SJMP    BACK            ;keep doing it
```

I/O PROGRAMMING

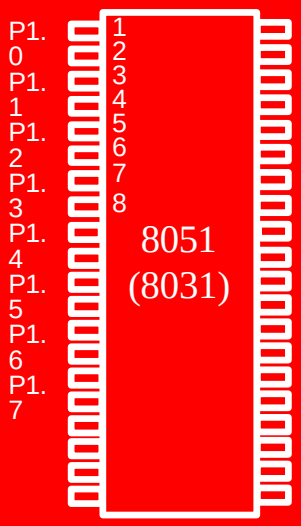
Dual Role of Port 0



- Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data
  - When connecting an 8051/31 to an external memory, port 0 provides both address and data

I/O PROGRAMMING

Port 1



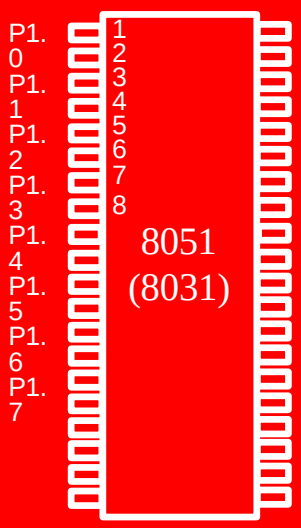
- Port 1 can be used as input or output
  - In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally
  - Upon reset, port 1 is configured as an input port

The following code will continuously send out to port 0 the alternating value 55H and AAH

```
        MOV     A,#55H
BACK:   MOV     P1,A
        ACALL   DELAY
        CPL     A
        SJMP    BACK
```

I/O PROGRAMMING

Port 1 as Input



- To make port 1 an input port, it must be programmed as such by writing 1 to all its bits

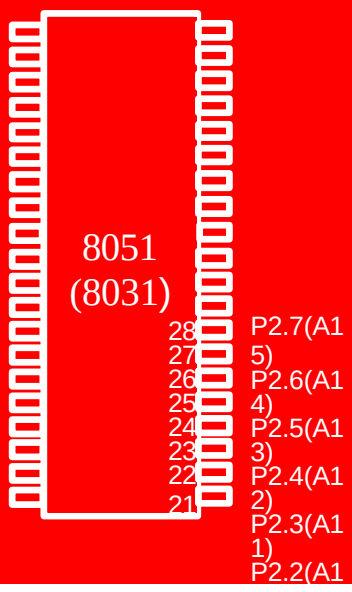
Port 1 is configured first as an input port by writing 1s to it, then data is received from that port and saved in R7 and R5

```
        MOV     A,#0FFH     ;A=FF hex
        MOV     P1,A        ;make P1 an input port
                            ;by writing it all 1s
        MOV     A,P1        ;get data from P1
        MOV     R7,A        ;save it to in reg R7
        ACALL   DELAY       ;wait
        MOV     A,P1        ;another data from P1
        MOV     R5,A        ;save it to in reg R5
```

I/O PROGRAMMING

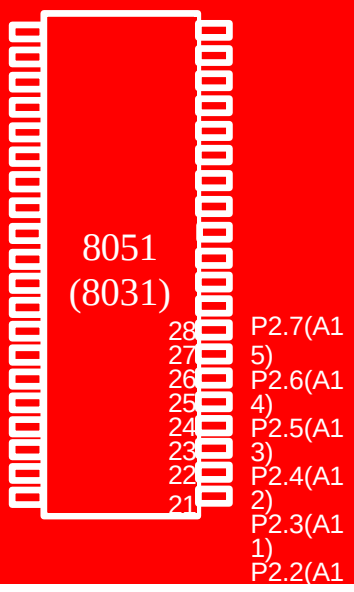
Port 2

- Port 2 can be used as input or output
  - Just like P1, port 2 does not need any pull-up resistors since it already has pull-up resistors internally
  - Upon reset, port 2 is configured as an input port



I/O PROGRAMMING

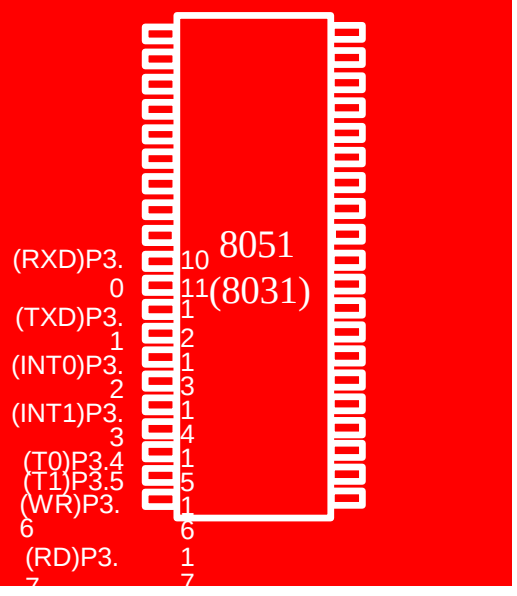
Port 2 as Input or Dual Role



- To make port 2 an input port, it must be programmed as such by writing 1 to all its bits
- In many 8051-based system, P2 is used as simple I/O
- In 8031-based systems, port 2 must be used along with P0 to provide the 16-bit address for the external memory
  - Port 2 is also designated as A8 – A15, indicating its dual function
  - Port 0 provides the lower 8 bits via A0 – A7

I/O PROGRAMMING

Port 3



- Port 3 can be used as input or output
  - Port 3 does not need any pull-up resistors
  - Port 3 is configured as an input port upon reset, this is not the way it is most commonly used

I/O PROGRAMMING

Port 3 (cont')

- Port 3 has the additional function of providing some extremely important signals

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1	TxD	11
P3.2	$\overline{\text{INT0}}$	12
P3.3	$\overline{\text{INT1}}$	13
P3.4	T0	14
P3.5	T1	15
P3.6	$\overline{\text{WR}}$	16
P3.7	$\overline{\text{RD}}$	17

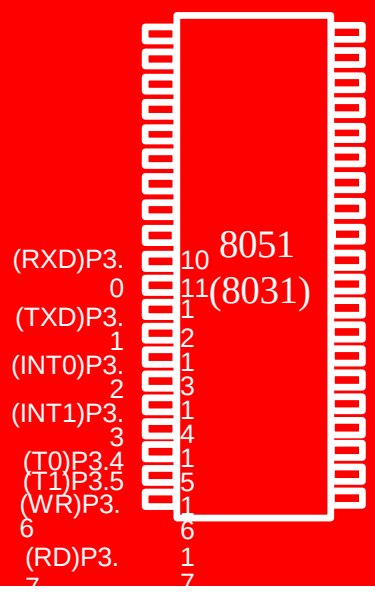
Serial communications

External interrupts

Timers

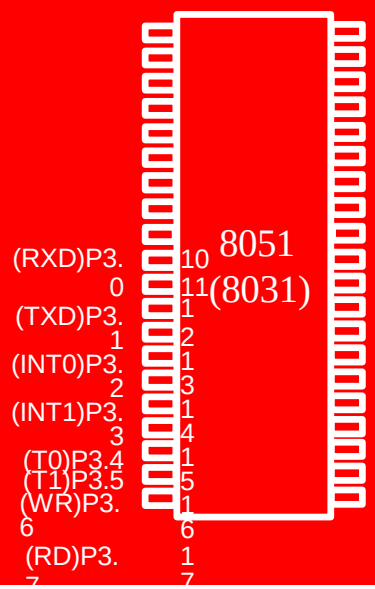
Read/Write signals of external memories

In systems based on 8751, 89C51 or DS89C4x0, pins 3.6 and 3.7 are used for I/O while the rest of the pins in port 3 are normally used in the alternate function role



I/O PROGRAMMING

Port 3 (cont')



Write a program for the DS89C420 to toggle all the bits of P0, P1, and P2 every 1/4 of a second

```
        ORG     0
BACK:   MOV     A,#55H
        MOV     P0,A
        MOV     P1,A
        MOV     P2,A
        ACALL   QSDELAY     ;Quarter of a second
        MOV     A,#0AAH
        MOV     P0,A
        MOV     P1,A
        MOV     P2,A
        ACALL   QSDELAY
        SJMP    BACK
QSDELAY:
        MOV     R5,#11
H3:     MOV     R4,#248
H2:     MOV     R3,#255
H1:     DJNZ    R3,H1       ;4 MC for DS89C4x0
        DJNZ    R4,H2
        DJNZ    R5,H3
        RET
        END
```

Delay
= 11 × 248 × 255 × 4 MC × 90 ns
= 250,430 µs

I/O PROGRAMMING

Different ways of Accessing Entire 8 Bits

The entire 8 bits of Port 1 are accessed

```
BACK:   MOV     A,#55H
        MOV     P1,A
        ACALL   DELAY
        MOV     A,#0AAH
        MOV     P1,A
        ACALL   DELAY
        SJMP    BACK
```

Rewrite the code in a more efficient manner by accessing the port directly without going through the accumulator

```
BACK:   MOV     P1,#55H
        ACALL   DELAY
        MOV     P1,#0AAH
        ACALL   DELAY
        SJMP    BACK
```

Another way of doing the same thing

```
        MOV     A,#55H
BACK:   MOV     P1,A
        ACALL   DELAY
        CPL     A
        SJMP    BACK
```

I/O BIT MANIPULATION PROGRAMMING

I/O Ports and Bit Addressability

- Sometimes we need to access only 1 or 2 bits of the port

```
BACK:   CPL     P1.2            ;complement P1.2
        ACALL   DELAY
        SJMP    BACK

;another variation of the above program
AGAIN:  SETB    P1.2            ;set only P1.2
        ACALL   DELAY
        CLR     P1.2            ;clear only P1.2
        ACALL   DELAY
        SJMP    AGAIN
```

P0	P1	P2	P3	Port Bit
P0.0	P1.0	P2.0	P3.0	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7	P1.7	P2.7	P3.7	D7

I/O BIT MANIPULATION PROGRAMMING

I/O Ports and Bit Addressability (cont')

Example 4-2

Write the following programs.
Create a square wave of 50% duty cycle on bit 0 of port 1.

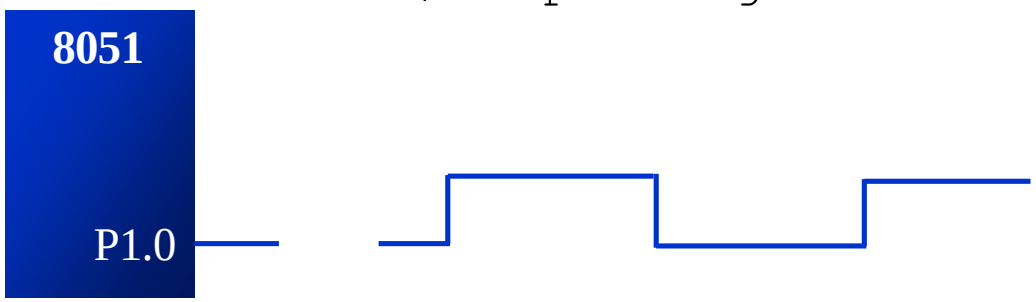
Solution:

The 50% duty cycle means that the "on" and "off" state (or the high and low portion of the pulse) have the same length. Therefore, we toggle P1.0 with a time delay in between each state.

```
HERE:  SETB   P1.0   ;set to high bit 0 of port 1
       LCALL  DELAY  ;call the delay subroutine
       CLR    P1.0   ;P1.0=0
       LCALL  DELAY
       SJMP   HERE   ;keep doing it
```

Another way to write the above program is:

```
HERE:  CPL    P1.0   ;set to high bit 0 of port 1
       LCALL  DELAY  ;call the delay subroutine
       SJMP   HERE   ;keep doing it
```



I/O BIT MANIPULATION PROGRAMMING

I/O Ports and Bit Addressability (cont')

- Instructions that are used for signal-bit operations are as following

Single-Bit Instructions

Instruction	Function
SETB bit	Set the bit (bit = 1)
CLR bit	Clear the bit (bit = 0)
CPL bit	Complement the bit (bit = NOT bit)
JB bit, target	Jump to target if bit = 1 (jump if bit)
JNB bit, target	Jump to target if bit = 0 (jump if no bit)
JBC bit, target	Jump to target if bit = 1, clear bit (jump if bit, then clear)

I/O BIT MANIPULATION PROGRAMMING

Checking an Input Bit

- The JNB and JB instructions are widely used single-bit operations
  - They allow you to monitor a bit and make a decision depending on whether it's 0 or 1
  - These two instructions can be used for any bits of I/O ports 0, 1, 2, and 3
    - Port 3 is typically not used for any I/O, either single-bit or byte-wise

Instructions for Reading an Input Port

Mnemonic	Examples	Description
MOV A,PX	MOV A,P2	Bring into A the data at P2 pins
JNB PX.Y, ..	JNB P2.1,TARGET	Jump if pin P2.1 is low
JB PX.Y, ..	JB P1.3,TARGET	Jump if pin P1.3 is high
MOV C,PX.Y	MOV C,P2.4	Copy status of pin P2.4 to CY

I/O BIT MANIPULATION PROGRAMMING

Checking an Input Bit (cont')

Example 4-3

Write a program to perform the following:
(a) Keep monitoring the P1.2 bit until it becomes high
(b) When P1.2 becomes high, write value 45H to port 0
(c) Send a high-to-low (H-to-L) pulse to P2.3

Solution:

```
        SETB P1.2          ;make P1.2 an input
        MOV  A,#45H        ;A=45H
AGAIN:  JNB  P1.2,AGAIN ; get out when P1.2=1
        MOV  P0,A          ;issue A to P0
        SETB P2.3          ;make P2.3 high
        CLR  P2.3          ;make P2.3 low for H-to-L
```

I/O BIT MANIPULATION PROGRAMMING

Checking an Input Bit (cont')

Example 4-4

Assume that bit P2.3 is an input and represents the condition of an oven. If it goes high, it means that the oven is hot. Monitor the bit continuously. Whenever it goes high, send a high-to-low pulse to port P1.5 to turn on a buzzer.

Solution:

```
HERE:  JNB   P2.3,HERE ;keep monitoring for high
       SETB  P1.5       ;set bit P1.5=1
       CLR   P1.5       ;make high-to-low
       SJMP  HERE       ;keep repeating
```

I/O BIT MANIPULATION PROGRAMMING

Checking an Input Bit (cont')

Example 4-5

A switch is connected to pin P1.7. Write a program to check the status of SW and perform the following:
(a) If SW=0, send letter 'N' to P2
(b) If SW=1, send letter 'Y' to P2

Solution:

```
        SETB P1.7          ;make P1.7 an input
AGAIN:  JB   P1.2,OVER     ;jump if P1.7=1
        MOV  P2,#'N'       ;SW=0, issue 'N' to P2
        SJMP AGAIN         ;keep monitoring
OVER:   MOV  P2,#'Y'       ;SW=1, issue 'Y' to P2
        SJMP AGAIN         ;keep monitoring
```

I/O BIT MANIPULATION PROGRAMMING

Reading Single Bit into Carry Flag

Example 4-6

A switch is connected to pin P1.7. Write a program to check the status of SW and perform the following:
(a) If SW=0, send letter 'N' to P2
(b) If SW=1, send letter 'Y' to P2
Use the carry flag to check the switch status.

Solution:

```
        SETB P1.7        ;make P1.7 an input
AGAIN:  MOV  C,P1.2      ;read SW status into CF
        JC   OVER        ;jump if SW=1
        MOV  P2,#'N'     ;SW=0, issue 'N' to P2
        SJMP AGAIN       ;keep monitoring
OVER:   MOV  P2,#'Y'     ;SW=1, issue 'Y' to P2
        SJMP AGAIN       ;keep monitoring
```

I/O BIT MANIPULATION PROGRAMMING

Reading Single Bit into Carry Flag (cont')

Example 4-7

A switch is connected to pin P1.0 and an LED to pin P2.7. Write a program to get the status of the switch and send it to the LED

Solution:

```
        SETB P1.7        ;make P1.7 an input
AGAIN:  MOV  C,P1.0      ;read SW status into CF
        MOV  P2.7,C      ;send SW status to LED
        SJMP AGAIN       ;keep repeating
```

The instruction 'MOV P2.7,P1.0' is wrong , since such an instruction does not exist

However 'MOV P2,P1' is a valid instruction

I/O BIT MANIPULATION PROGRAMMING

Reading Input Pins vs. Port Latch

- In reading a port
  - Some instructions read the status of port pins
  - Others read the status of an internal port latch
- Therefore, when reading ports there are two possibilities:
  - Read the status of the input pin
  - Read the internal latch of the output port
- Confusion between them is a major source of errors in 8051 programming
  - Especially where external hardware is concerned

READING INPUT PINS VS. PORT LATCH

Reading Latch for Output Port

- Some instructions read the contents of an internal port latch instead of reading the status of an external pin
  - For example, look at the `ANL P1,A` instruction and the sequence of actions is executed as follow
    1. It reads the internal latch of the port and brings that data into the CPU
    2. This data is ANDed with the contents of register A
    3. The result is rewritten back to the port latch
    4. The port pin data is changed and now has the same value as port latch

READING INPUT PINS VS. PORT LATCH

Reading Latch for Output Port (cont')

- Read-Modify-Write
  - The instructions read the port latch normally read a value, perform an operation then rewrite it back to the port latch

Instructions Reading a latch (Read-Modify-Write)

Mnemonics	Example
ANL PX	ANL P1,A
ORL PX	ORL P2,A
XRL PX	XRL P0,A
JBC PX.Y,TARGET	JBC P1.1,TARGET
CPL PX.Y	CPL P1.2
INC PX	INC P1
DEC PX	DEC P2
DJNZ PX.Y,TARGET	DJNZ P1,TARGET
MOV PX.Y,C	MOV P1.2,C
CLR PX.Y	CLR P2.3
SETB PX.Y	SETB P2.3

Note: x is 0, 1, 2, or 3 for P0 – P3

I/O BIT MANIPULATION PROGRAMMING

Read-modify-write Feature

- The ports in 8051 can be accessed by the Read-modify-write technique
  - This feature saves many lines of code by combining in a single instruction all three actions
    1. Reading the port
    2. Modifying it
    3. Writing to the port

```
        MOV    P1,#55H   ;P1=01010101
AGAIN:  XRL    P1,#0FFH  ;EX-OR P1 with 1111 1111
        ACALL  DELAY
        SJMP   BACK
```

ARITHMETIC & LOGIC INSTRUCTIONS AND PROGRAMS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C

Mazidi, Mazidi and McKinlay

ARITHMETIC INSTRUCTIONS

Addition of Unsigned Numbers

```
ADD  A,source  ;A = A + source
```

- The instruction ADD is used to add two operands
  - Destination operand is always in register A
  - Source operand can be a register, immediate data, or in memory
  - Memory-to-memory arithmetic operations are never allowed in 8051 Assembly language

Show how the flag register is affected by the following instruction.

```
MOV A,#0F5H ;A=F5 hex
ADD A,#0BH  ;A=F5+0B=00
```

Solution:

```
      F5H        1111 0101
+     0BH    +   0000 1011
     100H        0000 0000
```

CY =1, since there is a carry out from D7
PF =1, because the number of 1s is zero (an even number), PF is set to 1.
AC =1, since there is a carry from D3 to D4

ARITHMETIC INSTRUCTIONS

Addition of Individual Bytes

Assume that RAM locations 40 – 44H have the following values. Write a program to find the sum of the values. At the end of the program, register A should contain the low byte and R7 the high byte.

40 = (7D)
41 = (EB)
42 = (C5)
43 = (5B)
44 = (30)

Solution:

```
        MOV R0,#40H    ;load pointer
        MOV R2,#5      ;load counter
        CLR A          ;A=0
        MOV R7,A       ;clear R7
AGAIN:  ADD A,@R0      ;add the byte ptr to by R0
        JNC NEXT       ;if CY=0 don't add carry
        INC R7         ;keep track of carry
NEXT:   INC R0         ;increment pointer
        DJNZ R2,AGAIN  ;repeat until R2 is zero
```

ARITHMETIC INSTRUCTIONS

ADDC and Addition of 16-Bit Numbers

- When adding two 16-bit data operands, the propagation of a carry from lower byte to higher byte is concerned

```
    1
  3C E7
+ 3B 8D
-------
  78 74
```

When the first byte is added (E7+8D=74, CY=1).
The carry is propagated to the higher byte, which result in 3C + 3B + 1 =78 (all in hex)

Write a program to add two 16-bit numbers. Place the sum in R7 and R6; R6 should have the lower byte.

Solution:

```
        CLR  C          ;make CY=0
        MOV  A, #0E7H   ;load the low byte now A=E7H
        ADD  A, #8DH    ;add the low byte
        MOV  R6, A      ;save the low byte sum in R6
        MOV  A, #3CH    ;load the high byte
        ADDC A, #3BH    ;add with the carry
        MOV  R7, A      ;save the high byte sum
```

ARITHMETIC INSTRUCTIONS

BCD Number System

- The binary representation of the digits 0 to 9 is called BCD (Binary Coded Decimal)
  - Unpacked BCD
    - In unpacked BCD, the lower 4 bits of the number represent the BCD number, and the rest of the bits are 0
    - Ex. 00001001 and 00000101 are unpacked BCD for 9 and 5
  - Packed BCD
    - In packed BCD, a single byte has two BCD number in it, one in the lower 4 bits, and one in the upper 4 bits
    - Ex. 0101 1001 is packed BCD for 59H

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

ARITHMETIC INSTRUCTIONS

Unpacked and Packed BCD

- Adding two BCD numbers must give a BCD result

```
MOV     A, #17H
ADD     A, #28H
```

Adding these two numbers gives 0011 1111B (3FH), Which is not BCD!

The result above should have been 17 + 28 = 45 (0100 0101).
To correct this problem, the programmer must add 6 (0110) to the low digit: 3F + 06 = 45H.

ARITHMETIC INSTRUCTIONS

DA Instruction

DA A ;decimal adjust for addition

- The DA instruction is provided to correct the aforementioned problem associated with BCD addition
  - The DA instruction will add 6 to the lower nibble or higher nibble if need

Example:

```
MOV  A,#47H      ;A=47H first BCD operand
MOV  B,#25H      ;B=25H second BCD operand
ADD  A,B         ;hex(binary) addition(A=6CH)
DA   A           ;adjust for BCD addition
                 (A=72H)
```

6CH

72H

DA works only after an ADD, but not after INC

The “DA” instruction works only on A. In other word, while the source can be an operand of any addressing mode, the destination must be in register A in order for DA to work.

ARITHMETIC INSTRUCTIONS

DA Instruction (cont')

- Summary of DA instruction
  - After an ADD or ADDC instruction
    1. If the lower nibble (4 bits) is greater than 9, or if AC=1, add 0110 to the lower 4 bits
    2. If the upper nibble is greater than 9, or if CY=1, add 0110 to the upper 4 bits

Example:

	HEX	BCD	
	29	0010 1001	
+	18	+ 0001 1000	
	41	0100 0001	AC=1
+	6	+ 0110	
	47	0100 0111	

Since AC=1 after the addition, "DA A" will add 6 to the lower nibble.

The final result is in BCD format.

ARITHMETIC INSTRUCTIONS

DA Instruction (cont')

Assume that 5 BCD data items are stored in RAM locations starting at 40H, as shown below. Write a program to find the sum of all the numbers. The result must be in BCD.

40=(71)
41=(11)
42=(65)
43=(59)
44=(37)

Solution:

```
        MOV     R0,#40H     ;Load pointer
        MOV     R2,#5       ;Load counter
        CLR     A           ;A=0
        MOV     R7,A        ;Clear R7
AGAIN:  ADD     A,@R0       ;add the byte pointer
                            ;to by R0
        DA      A           ;adjust for BCD
        JNC     NEXT        ;if CY=0 don't
                            ;accumulate carry
        INC     R7          ;keep track of carries
NEXT:   INC     R0          ;increment pointer
        DJNZ    R2,AGAIN    ;repeat until R2 is 0
```

ARITHMETIC INSTRUCTIONS

Subtraction of Unsigned Numbers

- In many microprocessor there are two different instructions for subtraction: SUB and SUBB (subtract with borrow)
  - In the 8051 we have only SUBB
  - The 8051 uses adder circuitry to perform the subtraction

```
SUBB A,source ;A = A - source - CY
```

- To make SUB out of SUBB, we have to make CY=0 prior to the execution of the instruction
  - Notice that we use the CY flag for the borrow

ARITHMETIC INSTRUCTIONS

Subtraction of Unsigned Numbers (cont')

- SUBB when CY = 0
  1. Take the 2's complement of the subtrahend (source operand)
  2. Add it to the minuend (A)
  3. Invert the carry

```
        CLR   C
        MOV   A,#4C   ;load A with value 4CH
        SUBB  A,#6EH  ;subtract 6E from A
        JNC   NEXT    ;if CY=0 jump to NEXT
        CPL   A  ;if CY=1, take 1's complement
        INC   A  ;and increment to get 2's comp
NEXT:   MOV   R1,A    ;save A in R1
```

Solution:

```
  4C      0100 1100        0100 1100
- 6E      0110 1110        1001 0010
 -22                      01101 1110
```

① 2's complement

② +

CY =1

③ Invert carry

CY=0, the result is positive; CY=1, the result is negative and the destination has the 2's complement of the result

ARITHMETIC INSTRUCTIONS

Subtraction of Unsigned Numbers (cont')

- SUBB when CY = 1
  - This instruction is used for multi-byte numbers and will take care of the borrow of the lower operand

```
        CLR     C
        MOV     A,#62H   ;A=62H
        SUBB    A,#96H   ;62H-96H=CCH with CY=1
        MOV     R7,A     ;save the result
        MOV     A,#27H   ;A=27H
        SUBB    A,#12H   ;27H-12H-1=14H
        MOV     R6,A     ;save the result
```

A = 62H – 96H – 0 = CCH
CY = 1

A = 27H - 12H - 1 = 14H
CY = 0

Solution:

We have 2762H - 1296H = 14CCH.

ARITHMETIC INSTRUCTIONS

Unsigned Multiplication

- The 8051 supports byte by byte multiplication only
  - The byte are assumed to be unsigned data

```
MUL AB  ;AxB, 16-bit result in B, A
```

```
        MOV     A,#25H      ;load 25H to reg.  A
        MOV     B,#65H      ;load 65H to reg.  B
        MUL     AB          ;25H * 65H = E99 where
                            ;B = OEH and A = 99H
```

Unsigned Multiplication Summary (MUL AB)

Multiplication	Operand1	Operand2	Result
Byte x byte	A	B	B = high byte A = low byte

ARITHMETIC INSTRUCTIONS

Unsigned Division

- The 8051 supports byte over byte division only
  - The byte are assumed to be unsigned data

```
DIV   AB    ;divide A by B, A/B
```

```
        MOV     A,#95    ;load 95 to reg.   A
        MOV     B,#10    ;load 10 to reg.   B
        MUL     AB       ;A = 09(quotient)  and
                         ;B = 05(remainder)
```

Unsigned Division Summary (DIV AB)

Division	Numerator	Denominator	Quotient	Remainder
Byte / byte	A	B	A	B

CY is always 0
If B ≠ 0, OV = 0
If B = 0, OV = 1 indicates error

ARITHMETIC INSTRUCTIONS

Application for DIV

(a) Write a program to get hex data in the range of 00 – FFH from port 1 and convert it to decimal. Save it in R7, R6 and R5.

(b) Assuming that P1 has a value of FDH for data, analyze program.

Solution:

(a)

```
        MOV   A,#0FFH
        MOV   P1,A        ;make P1 an input port
        MOV   A,P1        ;read data from P1
        MOV   B,#10       ;B=0A hex
        DIV   AB          ;divide by 10
        MOV   R7,B        ;save lower digit
        MOV   B,#10
        DIV   AB          ;divide by 10 once more
        MOV   R6,B        ;save the next digit
        MOV   R5,A        ;save the last digit
```

(b) To convert a binary (hex) value to decimal, we divide it by 10 repeatedly until the quotient is less than 10. After each division the remainder is saves.

```
                  Q          R
FD/0A =           19         3 (low digit)
19/0A =           2          5 (middle digit)
                             2 (high digit)
```

Therefore, we have FDH=253.

SIGNED ARITHMETIC INSTRUCTIONS

Signed 8-bit Operands

- D7 (MSB) is the sign and D0 to D6 are the magnitude of the number
  - If D7=0, the operand is positive, and if D7=1, it is negative



- Positive numbers are 0 to +127
- Negative number representation (2's complement)
  1. Write the magnitude of the number in 8-bit binary (no sign)
  2. Invert each bit
  3. Add 1 to it

SIGNED ARITHMETIC INSTRUCTIONS

Signed 8-bit Operands (cont')

Show how the 8051 would represent -34H

Solution:

```
1.   0011 0100      34H given in binary
2.   1100 1011      invert each bit
3.   1100 1100      add 1 (which is CC in hex)
```

Signed number representation of -34 in 2's complement is CCH

Decimal	Binary	Hex
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
…	… …	…
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
…	… …	…
+127	0111 1111	7F

SIGNED ARITHMETIC INSTRUCTIONS

Overflow Problem

- If the result of an operation on signed numbers is too large for the register
  - An overflow has occurred and the programmer must be noticed

Examine the following code and analyze the result.

```
        MOV     A,#+96          ;A=0110 0000 (A=60H)
        MOV     R1,#+70         ;R1=0100 0110(R1=46H)
        ADD     A,R1            ;A=1010 0110
                                ;A=A6H=-90,INVALID
```

Solution:

```
        +96     0110 0000
      + +70     0100 0110
      + 166     1010 0110 and OV =1
```

According to the CPU, the result is -90, which is wrong. The CPU sets OV=1 to indicate the overflow

SIGNED ARITHMETIC INSTRUCTIONS

OV Flag

- In 8-bit signed number operations, OV is set to 1 if either occurs:
  1. There is a carry from D6 to D7, but no carry out of D7 (CY=0)
  2. There is a carry from D7 out (CY=1), but no carry from D6 to D7

```
MOV  A,#-128   ;A=1000 0000(A=80H)
MOV  R4,#-2    ;R4=1111 1110(R4=FEH)
ADD  A,R4      ;A=0111 1110(A=7EH=+126,INVALID)
     -128           1000 0000
   +   -2           1111 1110
     -130           0111 1110 and OV=1
```

OV = 1
The result +126 is wrong

SIGNED ARITHMETIC INSTRUCTIONS

OV Flag (cont')

```
MOV  A,#-2      ;A=1111 1110(A=FEH)
MOV  R1,#-5     ;R1=1111 1011(R1=FBH)
ADD  A,R1       ;A=1111 1001(A=F9H=-7,
                ;Correct, OV=0)
      -2            1111 1110
  +   -5            1111 1011
      ----          ---------
      -7            1111 1001 and OV=0
```

OV = 0
The result -7 is correct

```
MOV  A,#+7      ;A=0000 0111(A=07H)
MOV  R1,#+18    ;R1=0001 0010(R1=12H)
ADD  A,R1       ;A=0001 1001(A=19H=+25,
                ;Correct,OV=0)
       7            0000 0111
  +   18            0001 0010
      ---           ---------
      25            0001 1001 and OV=0
```

OV = 0
The result +25 is correct

SIGNED ARITHMETIC INSTRUCTIONS

OV Flag (cont')

- In unsigned number addition, we must monitor the status of CY (carry)
  - Use JNC or JC instructions
- In signed number addition, the OV (overflow) flag must be monitored by the programmer
  - `JB PSW.2` or `JNB PSW.2`

SIGNED ARITHMETIC INSTRUCTIONS

2's Complement

- To make the 2's complement of a number

```
        CPL     A           ;1's complement (invert)
        ADD     A,#1        ;add 1 to make 2's comp.
```

LOGIC AND COMPARE INSTRUCTIONS

AND

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

```
ANL destination,source
            ;dest = dest AND source
```

- This instruction will perform a logic AND on the two operands and place the result in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

Show the results of the following.

```
     MOV   A,#35H    ;A = 35H
     ANL   A,#0FH    ;A = A AND 0FH
```

```
35H     0 0 1 1 0 1 0 1
0FH     0 0 0 0 1 1 1 1
05H     0 0 0 0 0 1 0 1
```

ANL is often used to mask (set to 0) certain bits of an operand

LOGIC AND COMPARE INSTRUCTIONS

OR

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

```
ORL destination,source
                ;dest = dest OR source
```

- The destination and source operands are ORed and the result is placed in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

Show the results of the following.

```
      MOV   A,#04H    ;A = 04
      ORL   A,#68H    ;A = 6C

04H     0 0 0 0 0 1 0 0
68H     0 1 1 0 1 0 0 0
6CH     0 1 1 0 1 1 0 0
```

ORL instruction can be used to set certain bits of an operand to 1

LOGIC AND COMPARE INSTRUCTIONS

XOR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

```
XRL destination,source
              ;dest = dest XOR source
```

- This instruction will perform XOR operation on the two operands and place the result in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

Show the results of the following.

```
        MOV   A,#54H
        XRL   A,#78H
```

54H	0	1	0	1	0	1	0	0
78H	0	1	1	1	1	0	0	0
2CH	0	0	1	0	1	1	0	0

XRL instruction can be used to toggle certain bits of an operand

LOGIC AND COMPARE INSTRUCTIONS

XOR (cont')

The `XRL` instruction can be used to clear the contents of a register by XORing it with itself. Show how `XRL A,A` clears A, assuming that AH = 45H.

```
45H     0 1 0 0 0 1 0 1
45H     0 1 0 0 0 1 0 1
00H     0 0 0 0 0 0 0 0
```

Read and test P1 to see whether it has the value 45H. If it does, send 99H to P2; otherwise, it stays cleared.

Solution:

```
        MOV P2,#00      ;clear P2
        MOV P1,#0FFH    ;make P1 an input port
        MOV R3,#45H     ;R3=45H
        MOV A,P1        ;read P1
        XRL A,R3
        JNZ EXIT        ;jump if A is not 0
        MOV P2,#99H
EXIT:   ...
```

XRL can be used to see if two registers have the same value

If both registers have the same value, 00 is placed in A. JNZ and JZ test the contents of the accumulator.

LOGIC AND COMPARE INSTRUCTIONS

Complement Accumulator

```
CPL A ;complements the register A
```

- This is called 1's complement

```
        MOV A, #55H
        CPL A           ;now A=AAH
                        ;0101 0101(55H)
                        ;becomes 1010 1010(AAH)
```

- To get the 2's complement, all we have to do is to to add 1 to the 1's complement

LOGIC AND COMPARE INSTRUCTIONS

Compare Instruction

`CJNE destination,source,rel. addr.`

- The actions of comparing and jumping are combined into a single instruction called `CJNE` (compare and jump if not equal)
  - The `CJNE` instruction compares two operands, and jumps if they are not equal
  - The destination operand can be in the accumulator or in one of the Rn registers
  - The source operand can be in a register, in memory, or immediate
    - The operands themselves remain unchanged
  - It changes the CY flag to indicate if the destination operand is larger or smaller

LOGIC AND COMPARE INSTRUCTIONS

Compare Instruction (cont')

CY flag is always checked for cases of greater or less than, but only after it is determined that they are not equal

```
        CJNE R5,#80,NOT_EQUAL ;check R5 for 80
        ...                   ;R5 = 80
NOT_EQUAL:
        JNC  NEXT             ;jump if R5 > 80
        ...                   ;R5 < 80
NEXT:   ...
```

Compare	Carry Flag
destination ≥ source	CY = 0
destination < source	CY = 1

- Notice in the `CJNE` instruction that any Rn register can be compared with an immediate value
  - There is no need for register A to be involved

LOGIC AND COMPARE INSTRUCTIONS

Compare Instruction (cont')

- The compare instruction is really a subtraction, except that the operands remain unchanged
  - Flags are changed according to the execution of the `SUBB` instruction

Write a program to read the temperature and test it for the value 75. According to the test results, place the temperature value into the registers indicated by the following.

If T = 75 then A = 75
If T < 75 then R1 = T
If T > 75 then R2 = T

Solution:

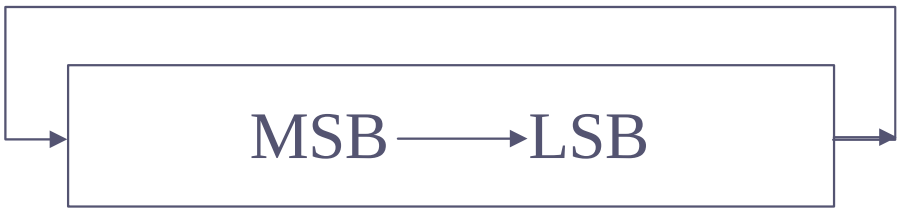
```
        MOV   P1,#0FFH     ;make P1 an input port
        MOV   A,P1         ;read P1 port
        CJNE  A,#75,OVER   ;jump if A is not 75
        SJMP  EXIT         ;A=75, exit
OVER:   JNC   NEXT         ;if CY=0 then A>75
        MOV   R1,A         ;CY=1, A<75, save in R1
        SJMP  EXIT         ; and exit
NEXT:   MOV   R2,A         ;A>75, save it in R2
EXIT:   ...
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

Rotating Right and Left

```
RR A          ;rotate right A
```

- In rotate right
  - The 8 bits of the accumulator are rotated right one bit, and
  - Bit D0 exits from the LSB and enters into MSB, D7




```
        MOV A,#36H     ;A = 0011 0110
        RR  A          ;A = 0001 1011
        RR  A          ;A = 1000 1101
        RR  A          ;A = 1100 0110
        RR  A          ;A = 0110 0011
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

Rotating Right and Left (cont')

```
RL A          ;rotate left A
```

- In rotate left
  - The 8 bits of the accumulator are rotated left one bit, and
  - Bit D7 exits from the MSB and enters into LSB, D0




```
        MOV A,#72H     ;A = 0111 0010
        RL  A          ;A = 1110 0100
        RL  A          ;A = 1100 1001
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

Rotating through Carry

```
RRC A    ;rotate right through carry
```

- In RRC A
  - Bits are rotated from left to right
  - They exit the LSB to the carry flag, and the carry flag enters the MSB




```
        CLR C           ;make CY = 0
        MOV A,#26H      ;A = 0010 0110
        RRC A           ;A = 0001 0011     CY = 0
        RRC A           ;A = 0000 1001     CY = 1
        RRC A           ;A = 1000 0100     CY = 1
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

Rotating through Carry (cont')

```
RLC A    ;rotate left through carry
```

- In RLC A
  - Bits are shifted from right to left
  - They exit the MSB and enter the carry flag, and the carry flag enters the LSB



Write a program that finds the number of 1s in a given byte.

```
        MOV     R1,#0
        MOV     R7,#8       ;count=08
        MOV     A,#97H
AGAIN:  RLC     A
        JNC     NEXT        ;check for CY
        INC     R1          ;if CY=1 add to count
NEXT:   DJNZ    R7,AGAIN
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

Serializing Data

- Serializing data is a way of sending a byte of data one bit at a time through a single pin of microcontroller
  - Using the serial port, discussed in Chapter 10
  - To transfer data one bit at a time and control the sequence of data and spaces in between them

ROTATE INSTRUCTION AND DATA SERIALIZATION

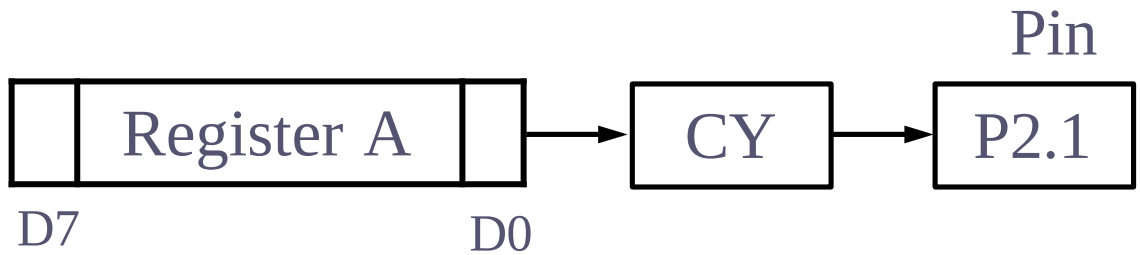
Serializing Data (cont')

- Transfer a byte of data serially by
  - Moving CY to any pin of ports P0 – P3
  - Using rotate instruction

Write a program to transfer value 41H serially (one bit at a time) via pin P2.1. Put two highs at the start and end of the data. Send the byte LSB first.

Solution:

```
        MOV    A,#41H
        SETB   P2.1        ;high
        SETB   P2.1        ;high
        MOV    R5,#8
AGAIN:  RRC    A
        MOV    P2.1,C      ;send CY to P2.1
        DJNZ   R5,HERE
        SETB   P2.1        ;high
        SETB   P2.1        ;high
```



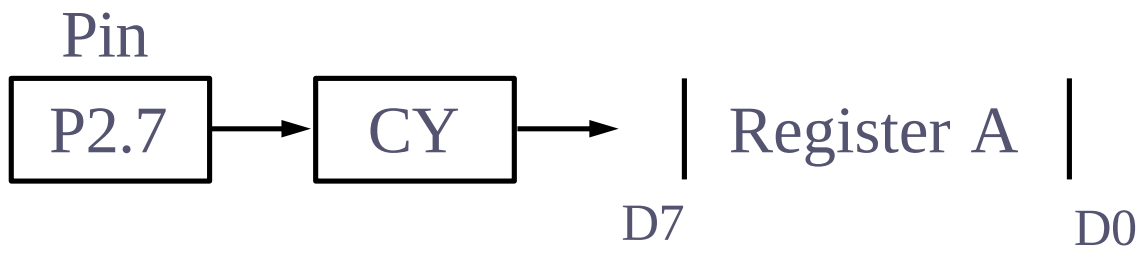
ROTATE INSTRUCTION AND DATA SERIALIZATION

Serializing Data (cont')

Write a program to bring in a byte of data serially one bit at a time via pin P2.7 and save it in register R2. The byte comes in with the LSB first.

Solution:

```
        MOV     R5,#8
AGAIN:  MOV     C,P2.7      ;bring in bit
        RRC     A
        DJNZ    R5,HERE
        MOV     R2,A        ;save it
```



ROTATE INSTRUCTION AND DATA SERIALIZATION

Single-bit Operations with CY

- There are several instructions by which the CY flag can be manipulated directly

Instruction	Function
SETB C	Make CY = 1
CLR C	Clear carry bit (CY = 0)
CPL C	Complement carry bit
MOV b,C	Copy carry status to bit location (CY = b)
MOV C,b	Copy bit location status to carry (b = CY)
JNC target	Jump to target if CY = 0
JC target	Jump to target if CY = 1
ANL C,bit	AND CY with bit and save it on CY
ANL C,/bit	AND CY with inverted bit and save it on CY
ORL C,bit	OR CY with bit and save it on CY
ORL C,/bit	OR CY with inverted bit and save it on CY

ROTATE INSTRUCTION AND DATA SERIALIZATION

Single-bit Operations with CY (cont')

Assume that bit P2.2 is used to control an outdoor light and bit P2.5 a light inside a building. Show how to turn on the outside light and turn off the inside one.

Solution:

```
        SETB    C           ;CY = 1
        ORL     C,P2.2      ;CY = P2.2 ORed w/ CY
        MOV     P2.2,C      ;turn it on if not on
        CLR     C           ;CY = 0
        ANL     C,P2.5      ;CY = P2.5 ANDed w/ CY
        MOV     P2.5,C     ;turn it off if not off
```

Write a program that finds the number of 1s in a given byte.

Solution:

```
        MOV     R1,#0       ;R1 keeps number of 1s
        MOV     R7,#8   ;counter, rotate 8 times
        MOV     A,#97H ;find number of 1s in 97H
AGAIN:  RLC     A           ;rotate it thru CY
        JNC     NEXT        ;check CY
        INC     R1          ;if CY=1, inc count
NEXT:   DJNZ    R7,AGAIN    ;go thru 8 times
```

ROTATE INSTRUCTION AND DATA SERIALIZATION

SWAP

`SWAP A`

- It swaps the lower nibble and the higher nibble
  - In other words, the lower 4 bits are put into the higher 4 bits and the higher 4 bits are put into the lower 4 bits
- SWAP works only on the accumulator (A)



ROTATE INSTRUCTION AND DATA SERIALIZATION

SWAP (cont')

(a) Find the contents of register A in the following code.
(b) In the absence of a SWAP instruction, how would you exchange the nibbles? Write a simple program to show the process.

Solution:

```
(a)
        MOV     A,#72H      ;A = 72H
        SWAP    A           ;A = 27H
(b)
        MOV     A,#72H      ;A = 0111 0010
        RL      A           ;A = 0111 0010
        RL      A           ;A = 0111 0010
        RL      A           ;A = 0111 0010
        RL      A           ;A = 0111 0010
```

BCD AND ASCII APPLICATION PROGRAMS

ASCII code and BCD for digits 0 - 9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

BCD AND ASCII APPLICATION PROGRAMS

Packed BCD to ACSII Conversion

- The DS5000T microcontrollers have a real-time clock (RTC)
  - The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off
- However this data is provided in packed BCD
  - To be displayed on an LCD or printed by the printer, it must be in ACSII format

Packed BCD		Unpacked BCD		ASCII
29H 0010 1001	→	02H & 09H 0000 0010 & 0000 1001	→	32H & 39H 0011 0010 & 0011 1001

BCD AND ASCII APPLICATION PROGRAMS

ASCII to Packed BCD Conversion

- To convert ASCII to packed BCD
  - It is first converted to unpacked BCD (to get rid of the 3)
  - Combined to make packed BCD

key	ASCII	Unpacked BCD	Packed BCD
4	34	0000 0100	
7	37	0000 0111	0100 0111 or 47H

```
        MOV    A, #'4'   ;A=34H, hex for '4'
        MOV    R1,#'7'   ;R1=37H,hex for '7'
        ANL    A, #0FH   ;mask upper nibble (A=04)
        ANL    R1,#0FH   ;mask upper nibble (R1=07)
        SWAP   A         ;A=40H
        ORL    A, R1     ;A=47H, packed BCD
```

BCD AND ASCII APPLICATION PROGRAMS

ASCII to Packed BCD Conversion (cont')

Assume that register A has packed BCD, write a program to convert packed BCD to two ASCII numbers and place them in R2 and R6.

```
MOV    A,#29H   ;A=29H, packed BCD
MOV    R2,A     ;keep a copy of BCD data
ANL    A,#0FH   ;mask the upper nibble (A=09)
ORL    A,#30H   ;make it an ASCII, A=39H('9')
MOV    R6,A     ;save it
MOV    A,R2     ;A=29H, get the original
data
ANL    A,#0F0H ;mask the lower nibble
RR     A        ;rotate right
RR     A        ;rotate right
RR     A        ;rotate right
RR     A        ;rotate right
ORL    A,#30H   ;A=32H, ASCII char. '2'
MOV    R2,A     ;save ASCII char in R2
```

SWAP A

BCD AND ASCII APPLICATION PROGRAMS

Using a Look-up Table for ASCII

Assume that the lower three bits of P1 are connected to three switches. Write a program to send the following ASCII characters to P2 based on the status of the switches.

000	'0'
001	'1'
010	'2'
011	'3'
100	'4'
101	'5'
110	'6'
111	'7'

Solution:

```
        MOV     DPTR,#MYTABLE
        MOV     A,P1          ;get SW status
        ANL     A,#07H        ;mask all but lower 3
        MOVC    A,@A+DPTR     ;get data from table
        MOV     P2,A          ;display value
        SJMP    $             ;stay here
;--------------------
        ORG     400H
MYTABLE DB      '0','1','2','3','4','5','6','7'
        END
```

BCD AND ASCII APPLICATION PROGRAMS

Checksum Byte in ROM

- To ensure the integrity of the ROM contents, every system must perform the checksum calculation
  - The process of checksum will detect any corruption of the contents of ROM
  - The checksum process uses what is called a checksum byte
    - The checksum byte is an extra byte that is tagged to the end of series of bytes of data

BCD AND ASCII APPLICATION PROGRAMS

Checksum Byte in ROM (cont')

- To calculate the checksum byte of a series of bytes of data
  - Add the bytes together and drop the carries
  - Take the 2's complement of the total sum, and it becomes the last byte of the series
- To perform the checksum operation, add all the bytes, including the checksum byte
  - The result must be zero
  - If it is not zero, one or more bytes of data have been changed

BCD AND ASCII APPLICATION PROGRAMS

Checksum Byte in ROM (cont')

Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H.(a) Find the checksum byte, (b) perform the checksum operation to ensure data integrity, and (c) if the second byte 62H has been changed to 22H, show how checksum detects the error.

Solution:

(a) Find the checksum byte.

```
    25H
+   62H
+   3FH
+   52H
   118H
```

The checksum is calculated by first adding the bytes. The sum is 118H, and dropping the carry, we get 18H. The checksum byte is the 2's complement of 18H, which is E8H

(b) Perform the checksum operation to ensure data integrity.

```
    25H
+   62H
+   3FH
+   52H
+   E8H
   200H (dropping the carries)
```

Adding the series of bytes including the checksum byte must result in zero. This indicates that all the bytes are unchanged and no byte is corrupted.

(c) If the second byte 62H has been changed to 22H, show how checksum detects the error.

```
    25H
+   22H
+   3FH
+   52H
+   E8H
   1C0H (dropping the carry, we get C0H)
```

Adding the series of bytes including the checksum byte shows that the result is not zero, which indicates that one or more bytes have been corrupted.

BCD AND ASCII APPLICATION PROGRAMS

Binary (Hex) to ASCII Conversion

- Many ADC (analog-to-digital converter) chips provide output data in binary (hex)
  - To display the data on an LCD or PC screen, we need to convert it to ASCII
    - Convert 8-bit binary (hex) data to decimal digits, 000 – 255
    - Convert the decimal digits to ASCII digits, 30H – 39H