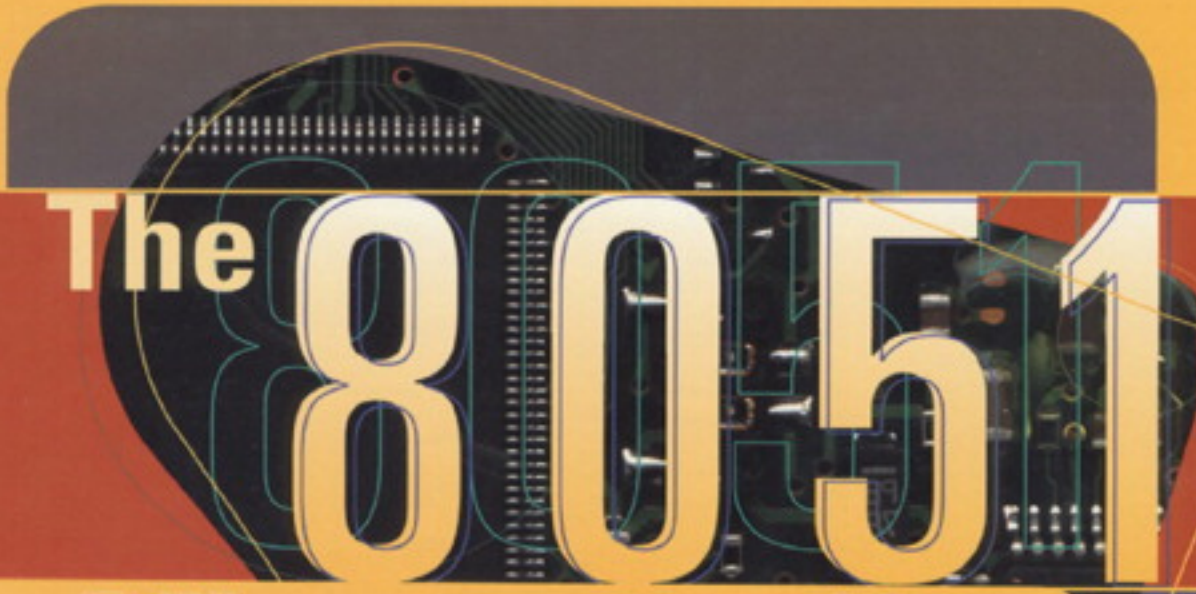


The 8051 Microcontroller and Embedded Systems

Muhammad Ali Mazidi
Janice Gillispie Mazidi

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 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

OUTLINES

- Numbering and coding systems
- Digital primer
- Inside the computer

NUMBERING AND CODING SYSTEMS

Decimal and Binary Number Systems

- Human beings use base 10 (*decimal*) arithmetic
  - There are 10 distinct symbols, 0, 1, 2, …, 9
- Computers use base 2 (*binary*) system
  - There are only 0 and 1
  - These two binary digits are commonly referred to as *bits*

NUMBERING AND CODING SYSTEMS

Converting from Decimal to Binary

- Divide the decimal number by 2 repeatedly
- Keep track of the remainders
- Continue this process until the quotient becomes zero
- Write the remainders in reverse order to obtain the binary number

Ex. Convert 25_{10} to binary

		Quotient	Remainder	
25/2	=	12	1	LSB (least significant bit)
12/2	=	6	0	
6/2	=	3	0	
3/2	=	1	1	
1/2	=	0	1	MSB (most significant bit)

Therefore $25_{10} = 11001_2$

NUMBERING AND CODING SYSTEMS

Converting from Binary to Decimal

- Know the weight of each bit in a binary number
- Add them together to get its decimal equivalent

Ex. Convert 11001_2 to decimal

Weight:	2^4	2^3	2^2	2^1	2^0
Digits:	1	1	0	0	1
Sum:	16 +	8 +	0 +	0 +	1 = 25_{10}

- Use the concept of weight to convert a decimal number to a binary directly

Ex. Convert 39_{10} to binary

32 + 0 + 0 + 4 + 2 + 1 = 39

Therefore, $39_{10} = 100111_2$

NUMBERING AND CODING SYSTEMS

Hexadecimal System

- Base 16, the *hexadecimal* system, is used as a convenient representation of binary numbers
  - ex.

    It is much easier to represent a string of 0s and 1s such as 100010010110 as its hexadecimal equivalent of 896H

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

NUMBERING AND CODING SYSTEMS

Converting between Binary and Hex

- To represent a binary number as its equivalent hexadecimal number
  - Start from the right and group 4 bits at a time, replacing each 4-bit binary number with its hex equivalent

Ex. Represent binary 100111110101 in hex

	1001	1111	0101
=	9	F	5

- To convert from hex to binary
  - Each hex digit is replaced with its 4-bit binary equivalent

Ex. Convert hex 29B to binary

	2	9	B
=	0010	1001	1011

NUMBERING AND CODING SYSTEMS

Converting from Decimal to Hex

- Convert to binary first and then convert to hex
- Convert directly from decimal to hex by repeated division, keeping track of the remainders

Ex. Convert 45_{10} to hex

32	16	8	4	2	1	
1	0	1	1	0	1	32 + 8 + 4 + 1 = 45

$45_{10} = 0010\ 1101_2 = 2D_{16}$

Ex. Convert 629_{10} to hex

512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	0	1	0	1

$629_{10} = 512+64+32+16+4+1 = 0010\ 0111\ 0101_2 = 275_{16}$

NUMBERING AND CODING SYSTEMS

Converting from Hex to Decimal

- Convert from hex to binary and then to decimal
- Convert directly from hex to decimal by summing the weight of all digits

Ex. $6B2_{16} = 0110\ 1011\ 0010_2$

1024	512	256	128	64	32	16	8	4	2	1
1	1	0	1	0	1	1	0	0	1	0

$1024 + 512 + 128 + 32 + 16 + 2 = 1714_{10}$

NUMBERING AND CODING SYSTEMS

Addition of Hex Numbers

- Adding the digits together from the least significant digits
  - If the result is less than 16, write that digit as the sum for that position
  - If it is greater than 16, subtract 16 from it to get the digit and carry 1 to the next digit

Ex. Perform hex addition: 23D9 + 94BE

```
  23D9    LSD:  9 + 14 = 23          23 – 16 =  7 w/ carry
+ 94BE          1 + 13 + 11 = 25     25 – 16 = 9 w/ carry
  B897          1 + 3 + 4 = 8
          MSD: 2 + 9 = B
```

Subtraction of Hex Numbers

- If the second digit is greater than the first, borrow 16 from the preceding digit

Ex. Perform hex subtraction: 59F – 2B8

```
  59F     LSD:  15 – 8 = 7
– 2B8           9 + 16 – 11 = 14 = E16
  2E7           5 – 1 – 2 = 2
```

- The ASCII (pronounced “ask-E”) code assigns binary patterns for
  - Numbers 0 to 9
  - All the letters of English alphabet, uppercase and lowercase
  - Many control codes and punctuation marks
- The ASCII system uses 7 bits to represent each code

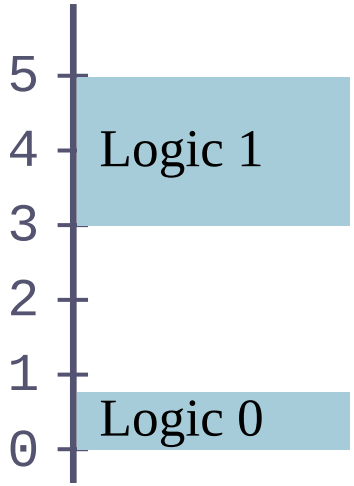
Selected ASCII codes

Hex	*Symbol*	*Hex*	*Symbol*
41	A	61	a
42	B	62	b
43	C	63	c
44	D	64	d
…	…	…	…
59	Y	79	y
5A	Z	7A	z

DIGITAL PRIMER

Binary Logic

- Two voltage levels can be represented as the two digits 0 and 1
- Signals in digital electronics have two distinct voltage levels with built-in tolerances for variations in the voltage
- A valid digital signal should be within either of the two shaded areas

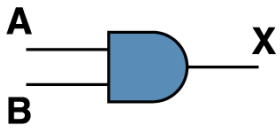


AND gate

Boolean Expression

$X = A \cdot B$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

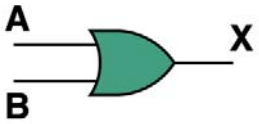
Computer Science Illuminated, Dale and Lewis

OR gate

Boolean Expression

$X = A + B$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Computer Science Illuminated, Dale and Lewis

DIGITAL PRIMER

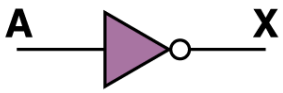
Logic Gates (cont')

- Tri-state buffer
- Inverter

Boolean Expression

$X = A'$

Logic Diagram Symbol



Truth Table

A	X
0	1
1	0

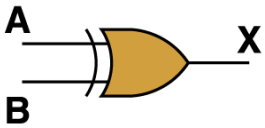
Computer Science Illuminated, Dale and Lewis

- XOR gate

Boolean Expression

$X = A \oplus B$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

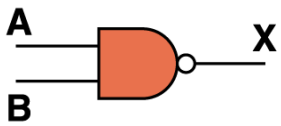
Computer Science Illuminated, Dale and Lewis

NAND gate

Boolean Expression

$X = (A \cdot B)'$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

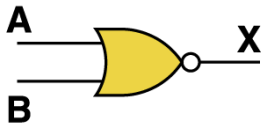
Computer Science Illuminated, Dale and Lewis

NOR gate

Boolean Expression

$X = (A + B)'$

Logic Diagram Symbol



Truth Table

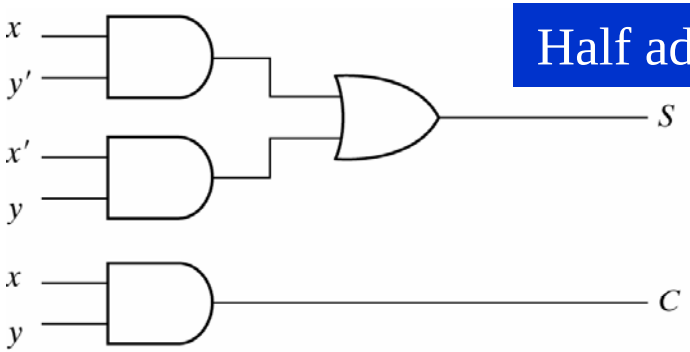
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Computer Science Illuminated, Dale and Lewis

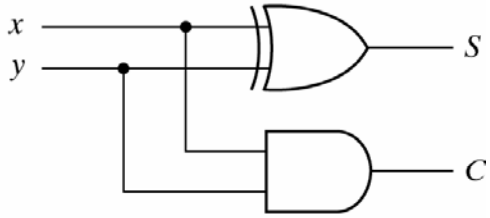
DIGITAL PRIMER

Logic Design Using Gates

Half adder

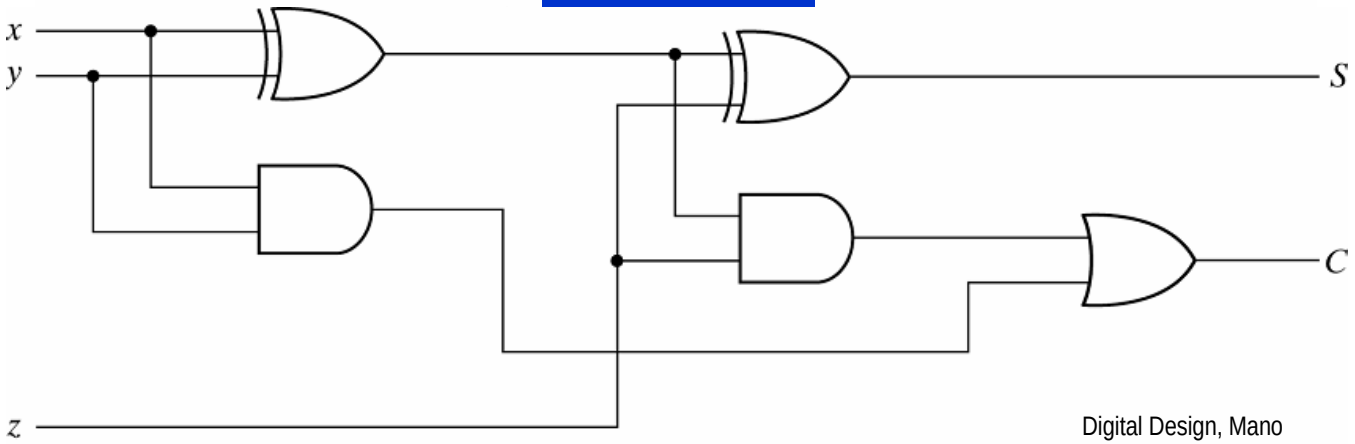


(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

Full adder

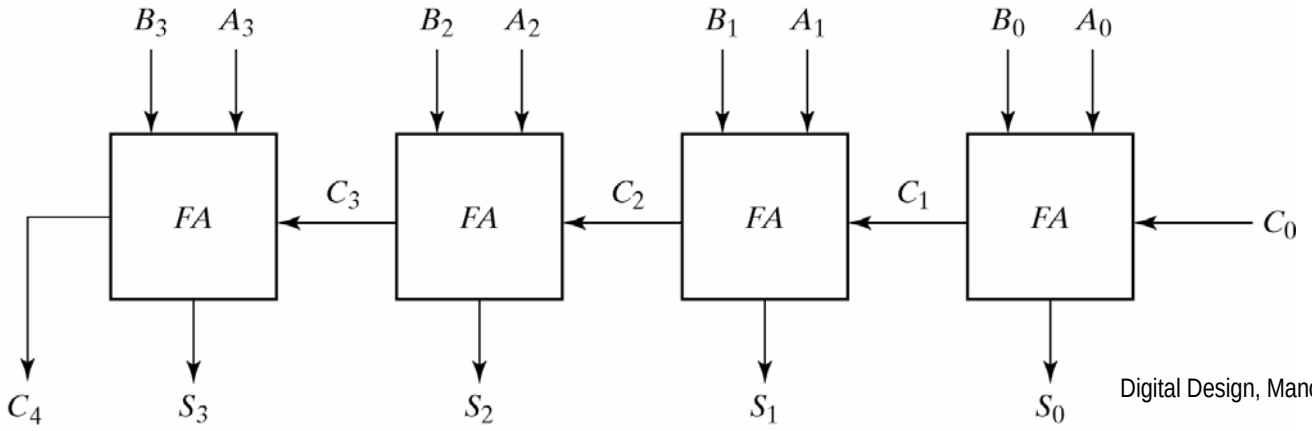


Digital Design, Mano

DIGITAL PRIMER

Logic Design Using Gates (cont')

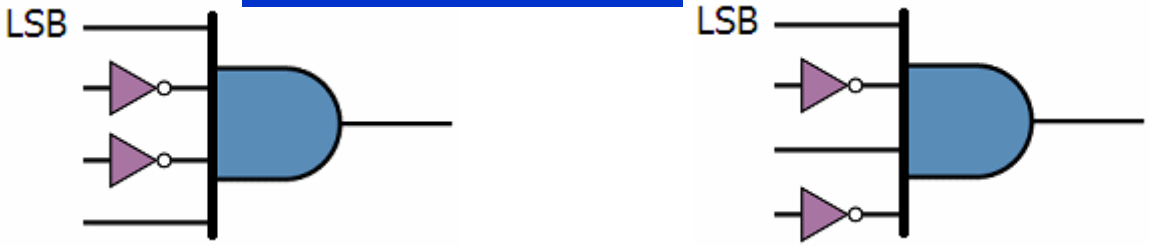
4-bit adder



Digital Design, Mano

- Decoders
  - Decoders are widely used for address decoding in computer design

Address Decoders



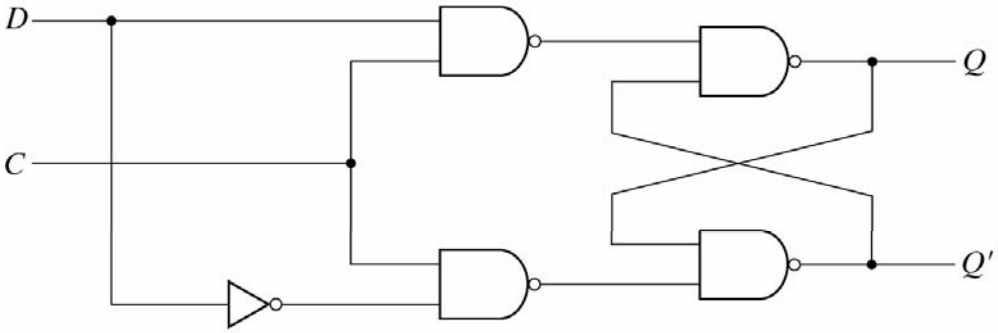
Address decoder for 9 (1001_2)

The output will be 1 if and only if the input is 1001_2

Address decoder for 5 (0101_2)

The output will be 1 if and only if the input is 0101_2

- Flip-flops
  - Flip-flops are frequently used to store data

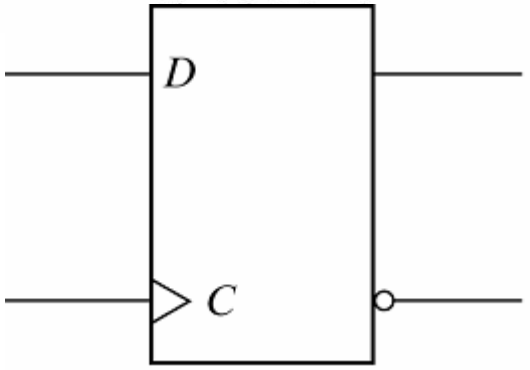


Digital Design, Mano

(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table



INSIDE THE COMPUTER

Important Terminology

- The unit of data size
  - *Bit* : a binary digit that can have the value 0 or 1
  - *Byte* : 8 bits
  - *Nibble* : half of a bye, or 4 bits
  - *Word* : two bytes, or 16 bits
- The terms used to describe amounts of memory in IBM PCs and compatibles
  - *Kilobyte* (K): 2^{10} bytes
  - *Megabyte* (M) : 2^{20} bytes, over 1 million
  - *Gigabyte* (G) : 2^{30} bytes, over 1 billion
  - *Terabyte* (T) : 2^{40} bytes, over 1 trillion

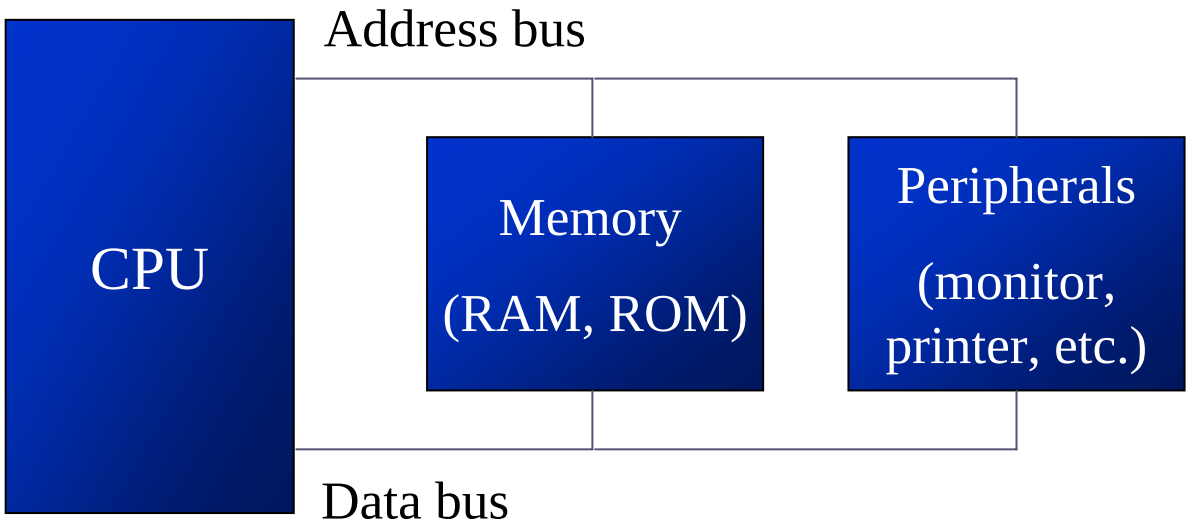
INSIDE THE COMPUTER

Internal Organization of Computers

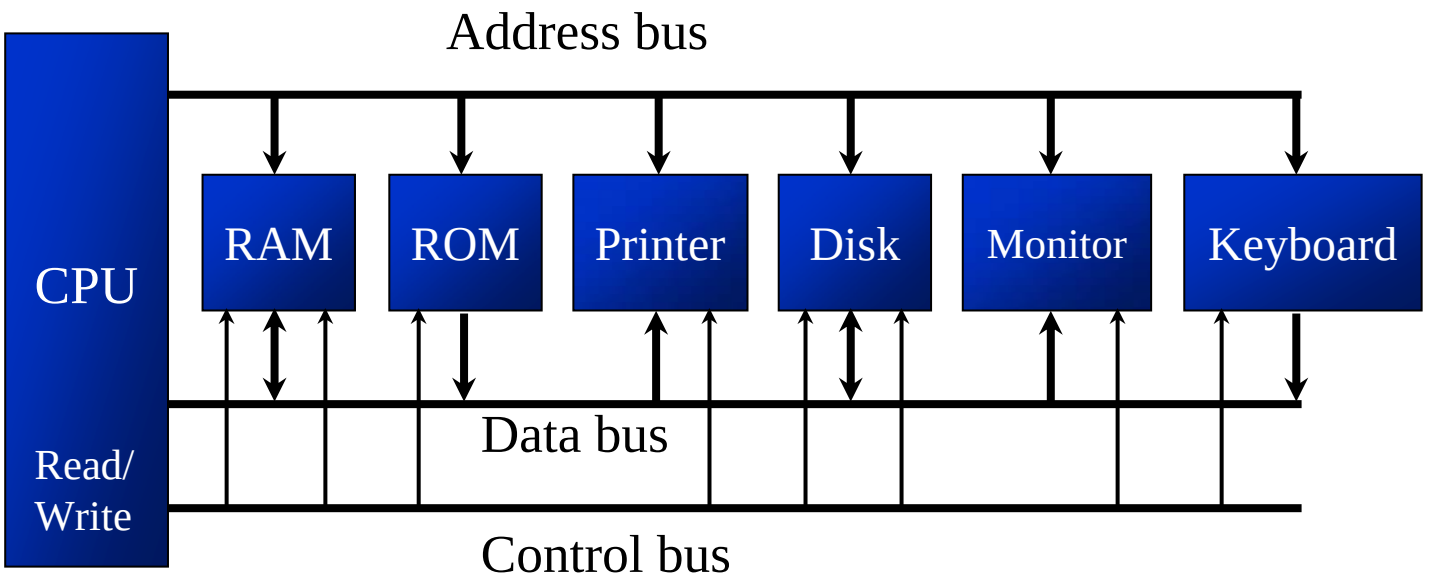
- CPU (Central Processing Unit)
  - Execute information stored in memory
- I/O (Input/output) devices
  - Provide a means of communicating with CPU
- Memory
  - RAM (Random Access Memory) – temporary storage of programs that computer is running
    - The data is lost when computer is off
  - ROM (Read Only Memory) – contains programs and information essential to operation of the computer
    - The information cannot be changed by use, and is not lost when power is off
      - It is called *nonvolatile memory*

INSIDE THE COMPUTER

Internal Organization of Computers (cont')



- The CPU is connected to memory and I/O through strips of wire called a *bus*
  - Carries information from place to place
    - Address bus
    - Data bus
    - Control bus



INSIDE THE COMPUTER

Internal Organization of Computers (cont')

- Address bus
  - For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address
    - The address assigned to a given device must be unique
    - The CPU puts the address on the address bus, and the decoding circuitry finds the device
- Data bus
  - The CPU either gets data from the device or sends data to it
- Control bus
  - Provides read or write signals to the device to indicate if the CPU is asking for information or sending it information

INSIDE THE COMPUTER

More about Data Bus

- The more data buses available, the better the CPU
  - Think of data buses as highway lanes
- More data buses mean a more expensive CPU and computer
  - The average size of data buses in CPUs varies between 8 and 64
- Data buses are bidirectional
  - To receive or send data
- The processing power of a computer is related to the size of its buses

INSIDE THE COMPUTER

More about Address Bus

- The more address buses available, the larger the number of devices that can be addressed
- The number of locations with which a CPU can communicate is always equal to 2^x, where x is the address lines, regardless of the size of the data bus
  - ex. a CPU with 24 address lines and 16 data lines can provide a total of 2^{24} or 16M bytes of addressable memory
  - Each location can have a maximum of 1 byte of data, since all general-purpose CPUs are *byte addressable*
- The address bus is unidirectional

INSIDE THE COMPUTER

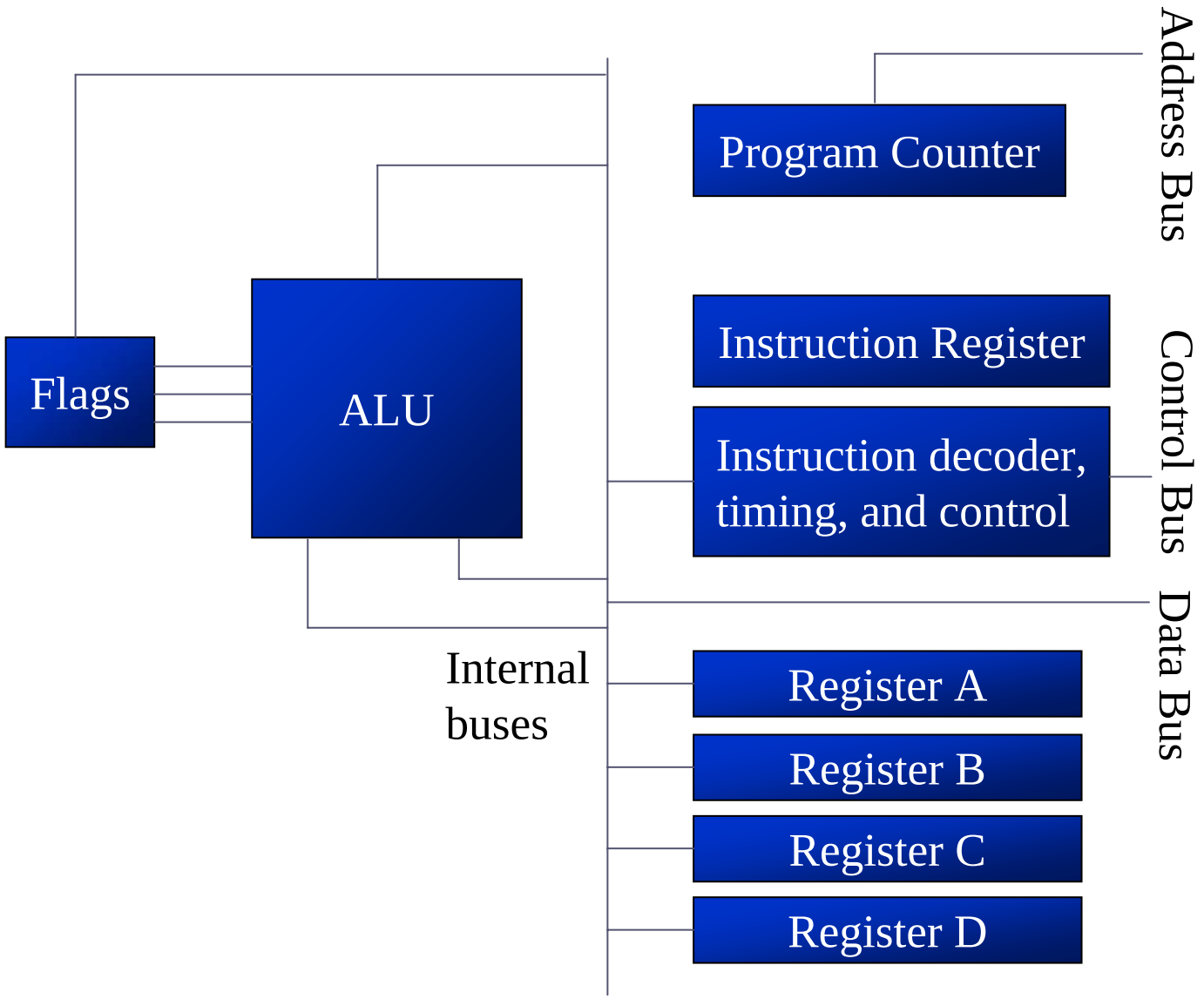
CPU's Relation to RAM and ROM

- For the CPU to process information, the data must be stored in RAM or ROM, which are referred to as *primary memory*
- ROM provides information that is fixed and permanent
  - Tables or initialization program
- RAM stores information that is not permanent and can change with time
  - Various versions of OS and application packages
  - CPU gets information to be processed
    - first form RAM (or ROM)
    - if it is not there, then seeks it from a mass storage device, called *secondary memory*, and transfers the information to RAM

- Registers
  - The CPU uses registers to store information temporarily
    - Values to be processed
    - Address of value to be fetched from memory
  - In general, the more and bigger the registers, the better the CPU
    - Registers can be 8-, 16-, 32-, or 64-bit
    - The disadvantage of more and bigger registers is the increased cost of such a CPU

INSIDE THE COMPUTER

Inside CPUs (cont')



- ALU (arithmetic/logic unit)
  - Performs arithmetic functions such as add, subtract, multiply, and divide, and logic functions such as AND, OR, and NOT
- Program counter
  - Points to the address of the next instruction to be executed
    - As each instruction is executed, the program counter is incremented to point to the address of the next instruction to be executed
- Instruction decoder
  - Interprets the instruction fetched into the CPU
    - A CPU capable of understanding more instructions requires more transistors to design

Ex. A CPU has registers A, B, C, and D and it has an 8-bit data bus and a 16-bit address bus. The CPU can access memory from addresses 0000 to FFFFH

Assume that the code for the CPU to move a value to register A is B0H and the code for adding a value to register A is 04H

The action to be performed by the CPU is to put 21H into register A, and then add to register A values 42H and 12H

…

Ex. (cont')

Action	*Code*	*Data*
Move value 21H into reg. A	B0H	21H
Add value 42H to reg. A	04H	42H
Add value 12H to reg. A	04H	12H

Mem. addr.	*Contents of memory address*
1400	(B0) code for moving a value to register A
1401	(21) value to be moved
1402	(04) code for adding a value to register A
1403	(42) value to be added
1404	(04) code for adding a value to register A
1405	(12) value to be added
1406	(F4) code for halt

…

Ex. (cont')

The actions performed by CPU are as follows:

1. The program counter is set to the value 1400H, indicating the address of the first instruction code to be executed
2.
   - The CPU puts 1400H on address bus and sends it out
     - The memory circuitry finds the location
   - The CPU activates the READ signal, indicating to memory that it wants the byte at location 1400H
     - This causes the contents of memory location 1400H, which is B0, to be put on the data bus and brought into the CPU

…

以動畫表示

Ex. (cont')

3.

- The CPU decodes the instruction B0
- The CPU commands its controller circuitry to bring into register A of the CPU the byte in the next memory location
  - The value 21H goes into register A
- The program counter points to the address of the next instruction to be executed, which is 1402H
  - Address 1402 is sent out on the address bus to fetch the next instruction

…

INSIDE THE COMPUTER

Internal Working of Computers (cont')

Ex. (cont')

4.

- From memory location 1402H it fetches code 04H
- After decoding, the CPU knows that it must add to the contents of register A the byte sitting at the next address (1403)
- After the CPU brings the value (42H), it provides the contents of register A along with this value to the ALU to perform the addition
  - It then takes the result of the addition from the ALU's output and puts it in register A
  - The program counter becomes 1404, the address of the next instruction

…

Ex. (cont')

5.

- Address 1404H is put on the address bus and the code is fetched into the CPU, decoded, and executed
  - This code is again adding a value to register A
  - The program counter is updated to 1406H

6.

- The contents of address 1406 are fetched in and executed
- This HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction

8051 MICROCONTROLLERS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

OUTLINES

- Microcontrollers and embedded processors
- Overview of the 8051 family

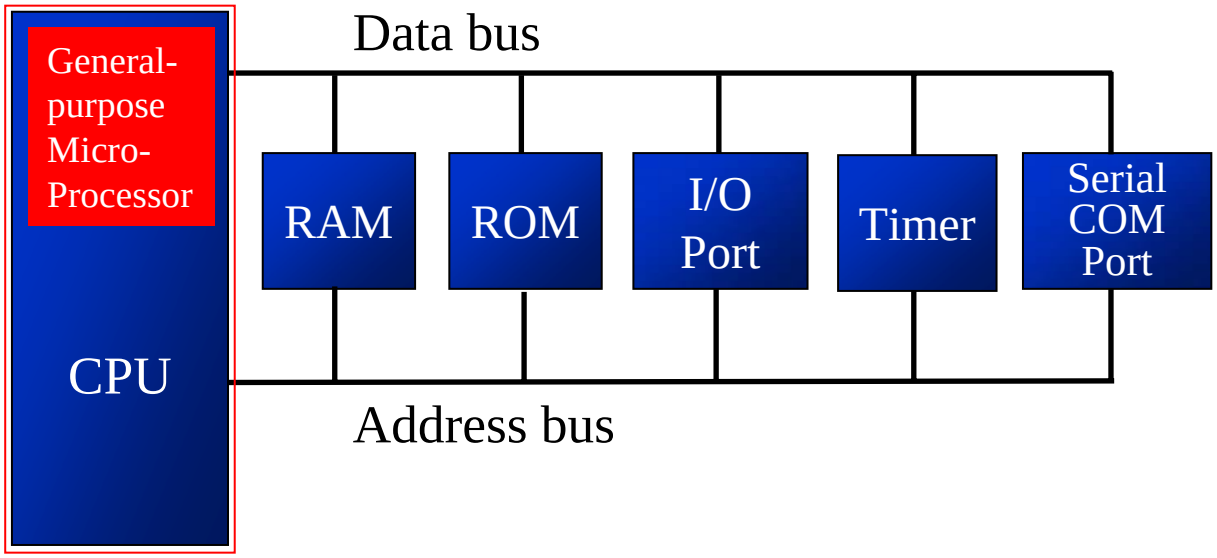
MICRO-CONTROLLERS AND EMBEDDED PROCESSORS

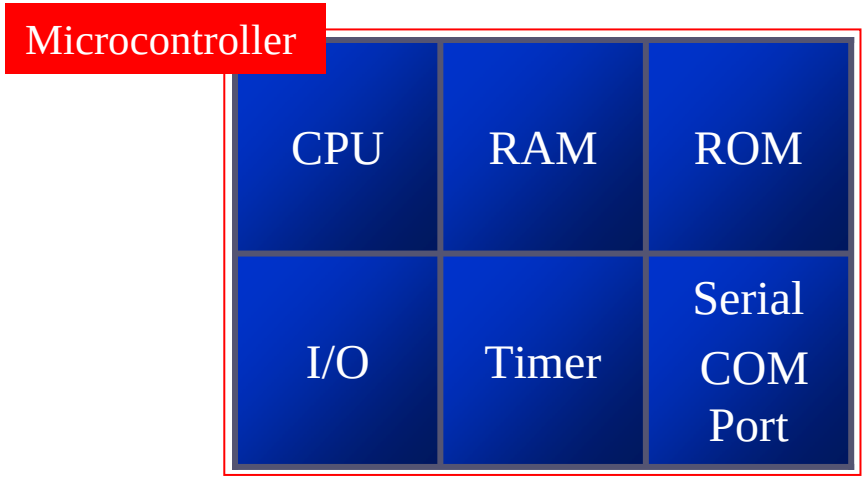
Microcontroller vs. General-Purpose Microprocessor

- 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

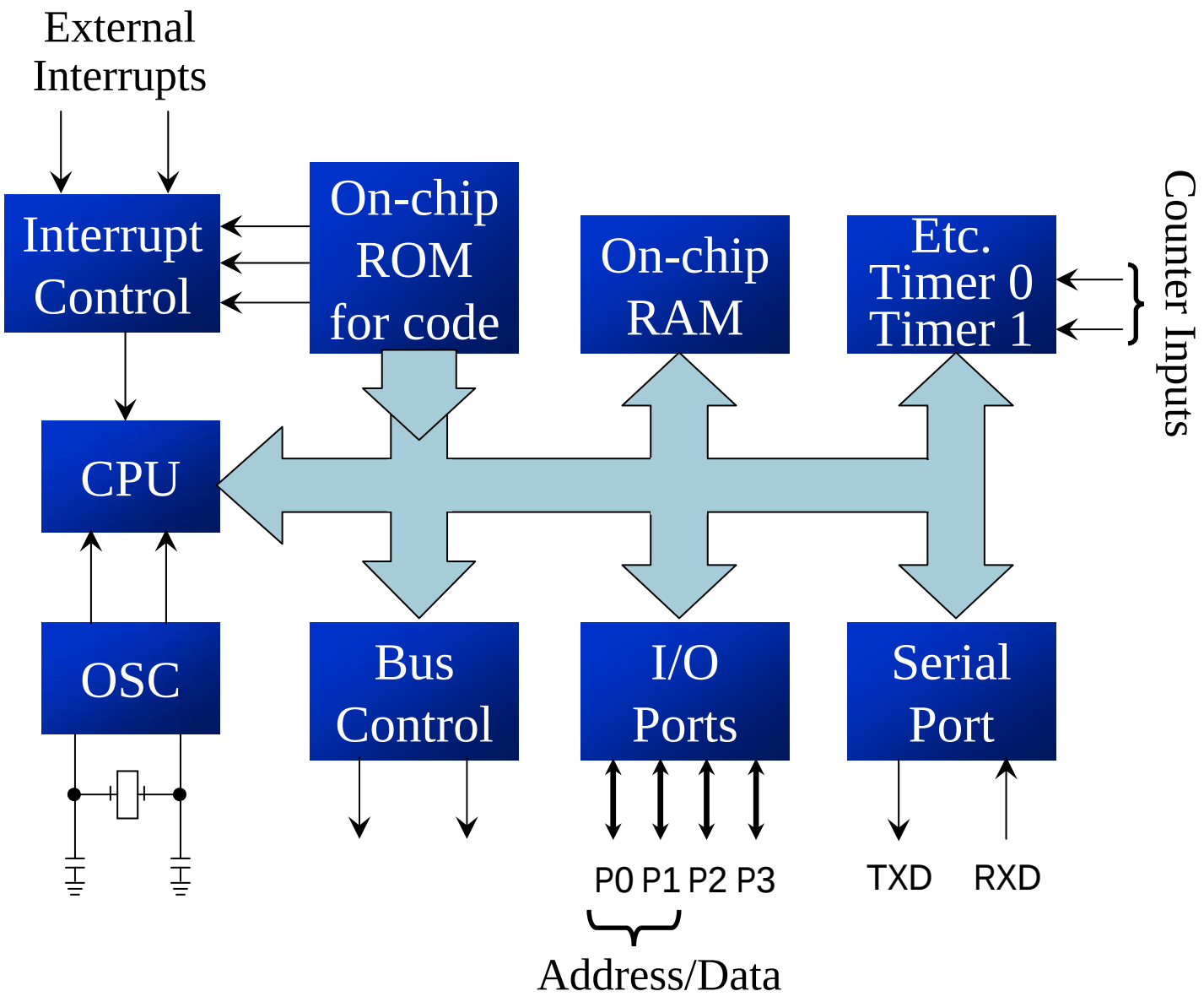
OVERVIEW OF 8051 FAMILY

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')



- 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

- 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

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

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

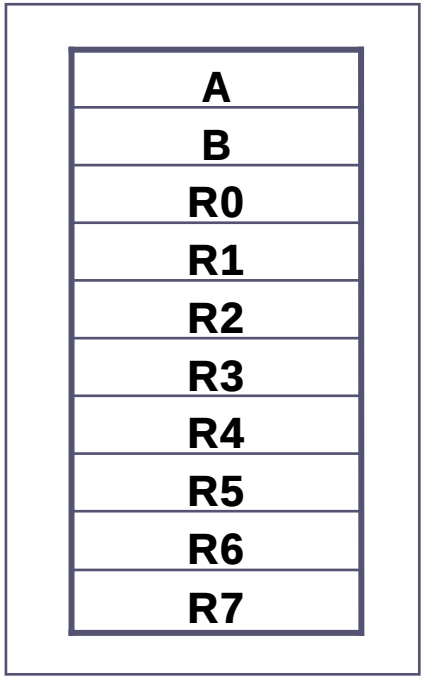
INSIDE THE 8051

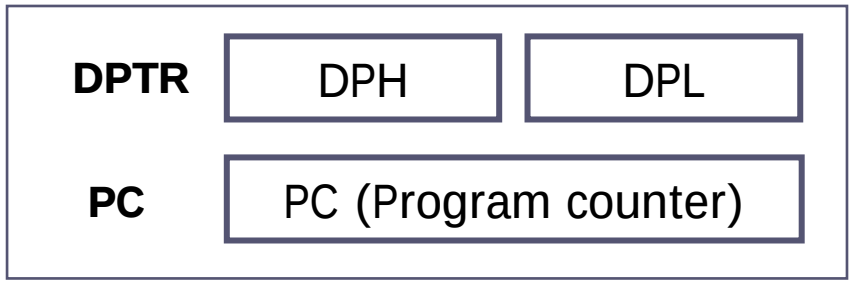
Registers (cont')

- 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

- 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

- 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

- 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')

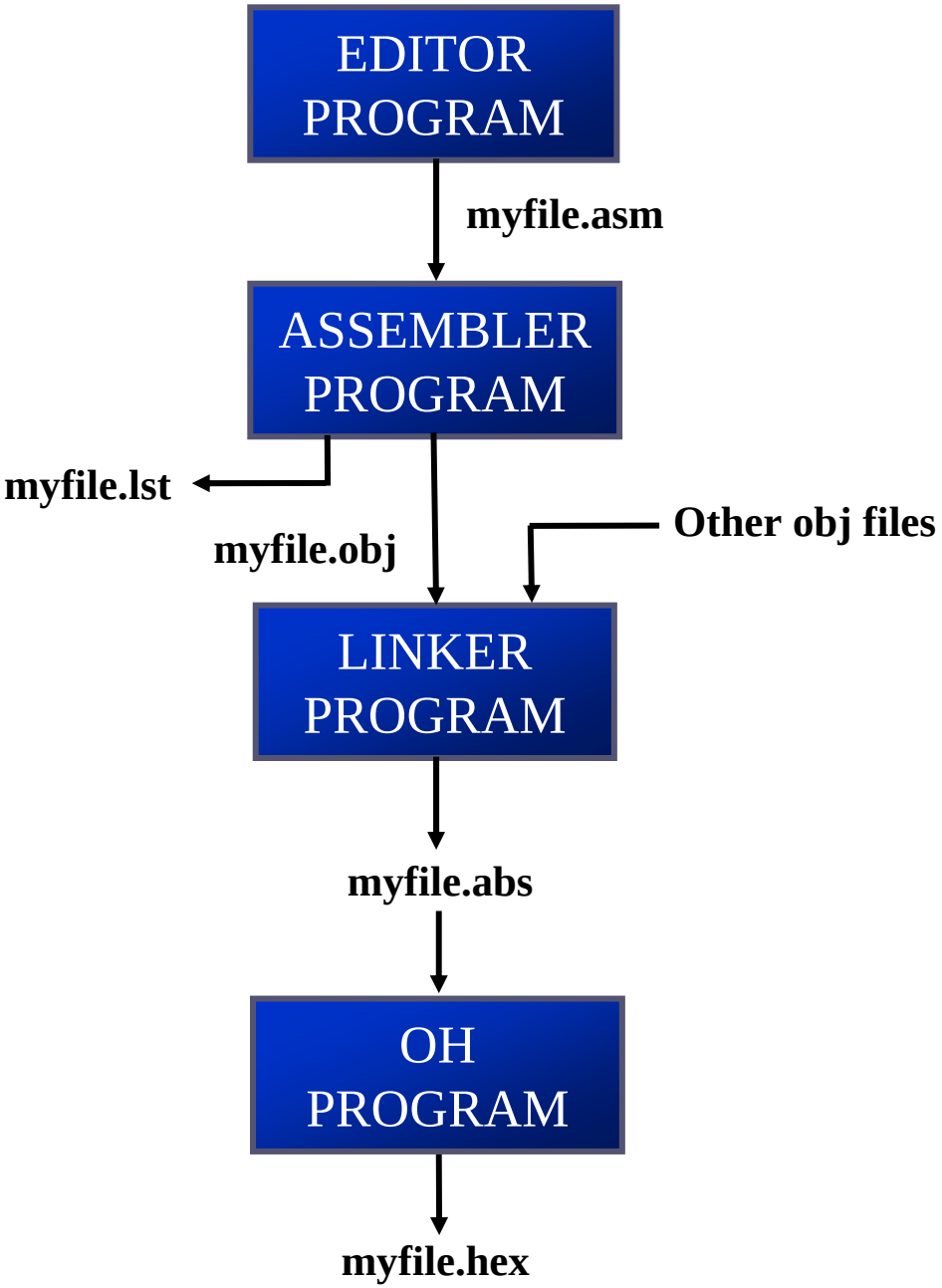
2) The "asm" source file containing the program code created in step 1 is fed to an 8051 assembler
   - The assembler converts the instructions into machine code
   - The assembler will produce an object file and a list file
   - The extension for the object file is "obj" while the extension for the 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
                                ;now A = A + 12H
8 000A  80EF HERE: SJMP HERE;stay in this loop
9 000C          END             ;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

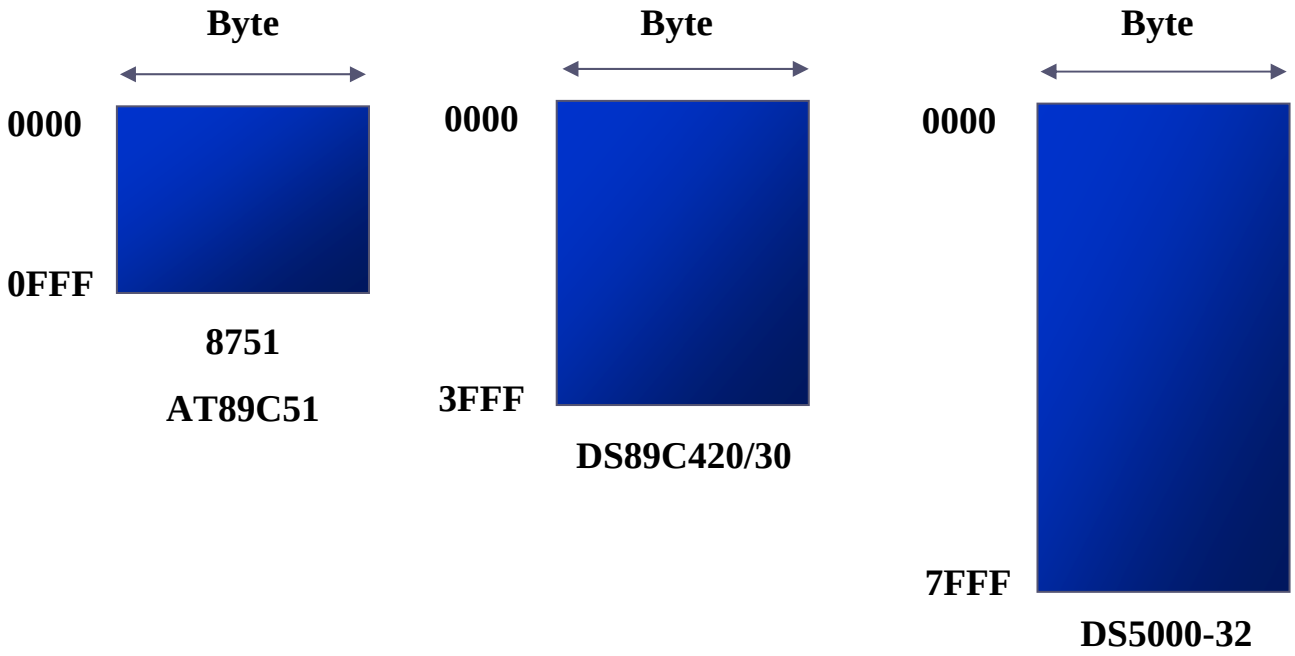
Executing Program (cont')

- (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

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)

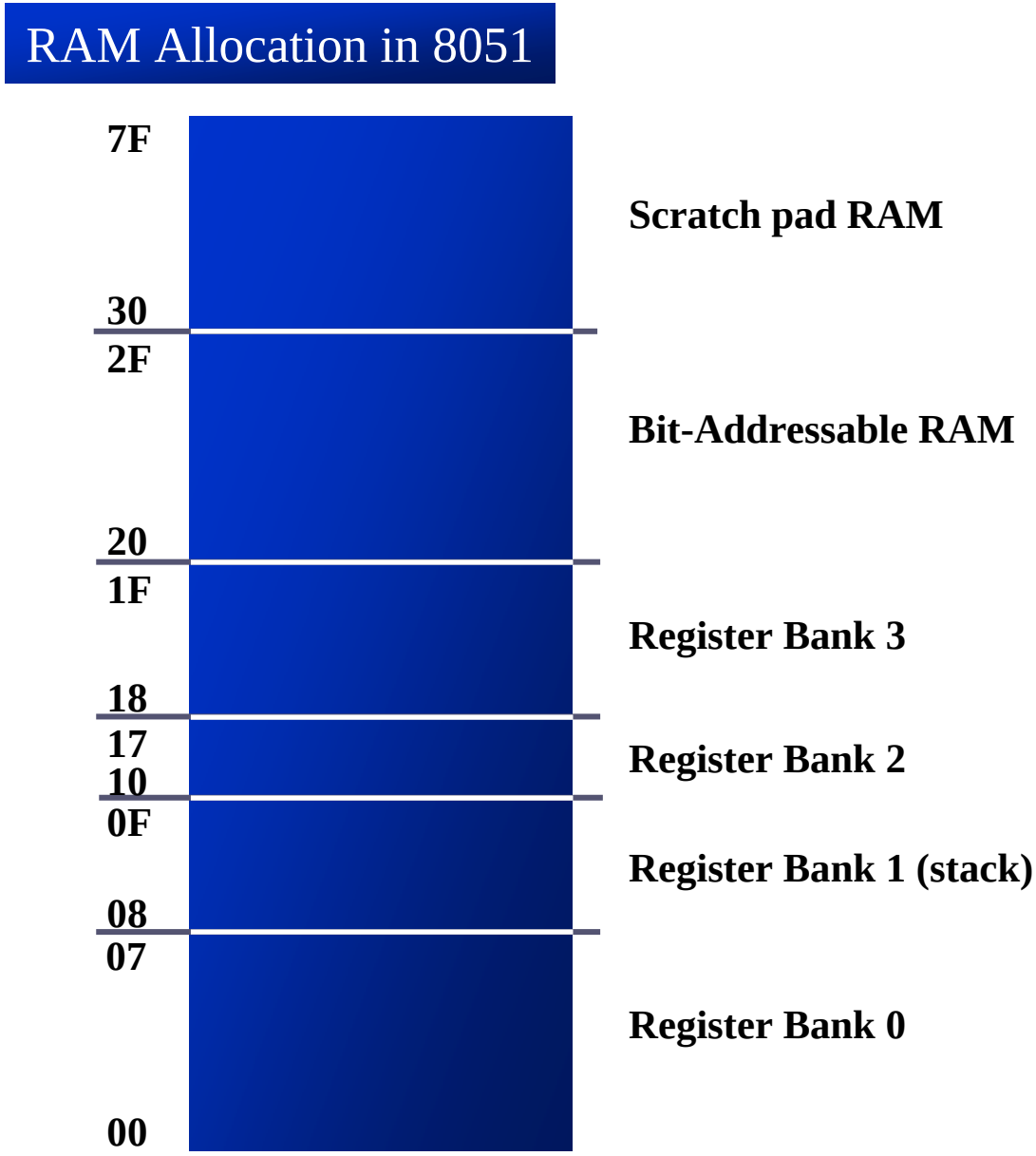
REGISTER BANKS AND STACK

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

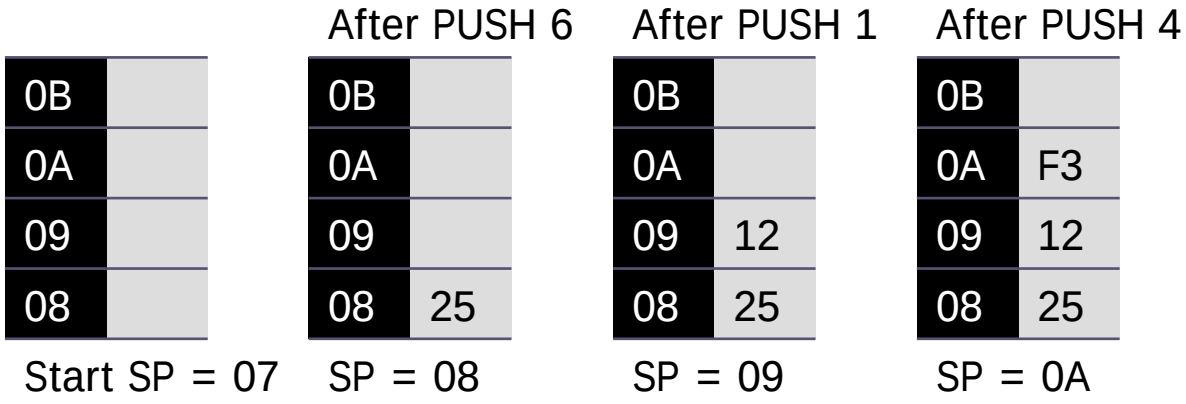
Pushing onto Stack

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

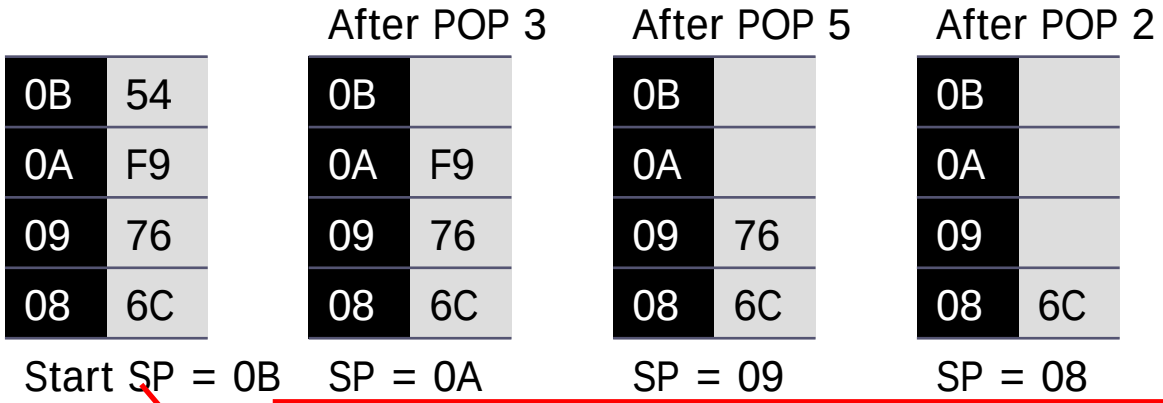
Popping From Stack

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

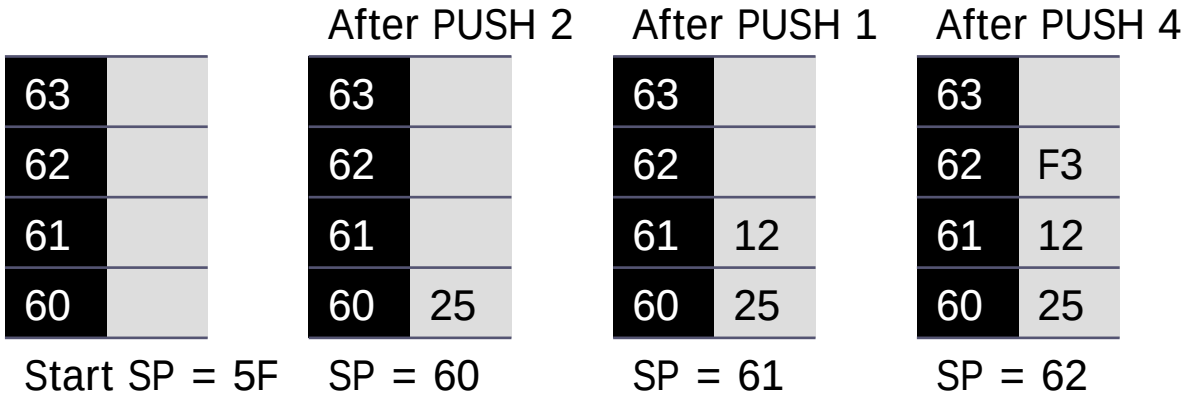
Stack And Bank 1 Conflict (cont')

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:



JUMP, LOOP AND CALL INSTRUCTIONS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

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 R5,#0

```
        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
```

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

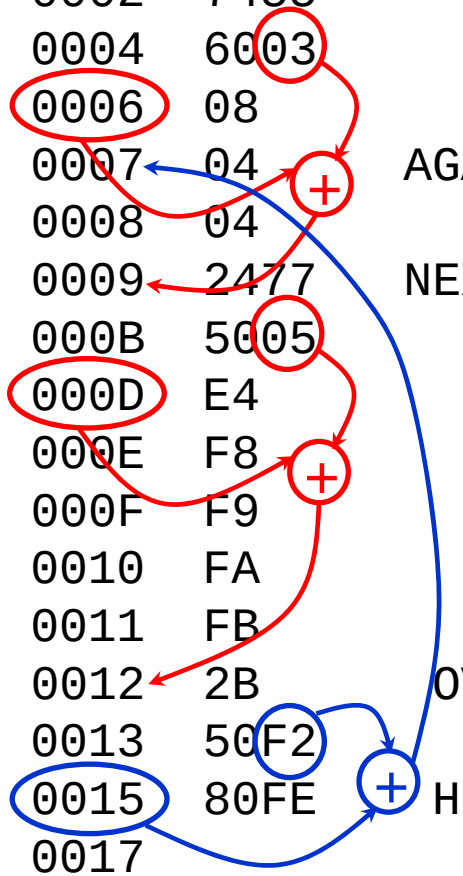
Calculating Short Jump Address

- 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.

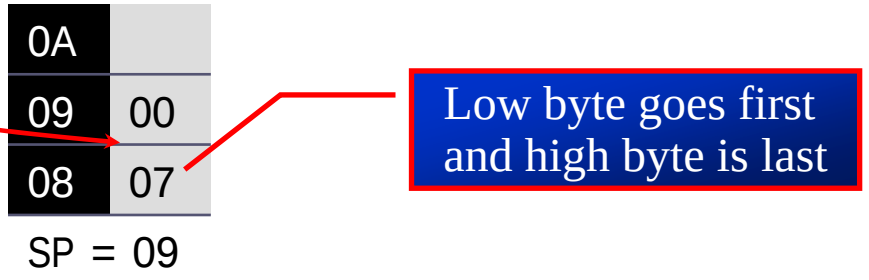
The amount of time delay depends on the frequency of the 8051

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
014 0302 DDFE    AGAIN:     DJNZ  R5,AGAIN ;stay here
015 0304 22                 RET            ;return to caller
016 0305                    END             ;end of asm file
```

Stack frame after the first LCALL



CALL INSTRUCTIONS

Use PUSH/POP in 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
   031                     RET            ;return to caller
22 031                                    e
```

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

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 = 1.085 μs
```

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	1x1.085 μs = 1.085 μs
(b)	1	1x1.085 μs = 1.085 μs
(c)	2	2x1.085 μs = 2.17 μs
(d)	2	2x1.085 μs = 2.17 μs
(e)	2	2x1.085 μs = 2.17 μs
(f)	1	1x1.085 μs = 1.085 μs
(g)	4	4x1.085 μs = 4.34 μs

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

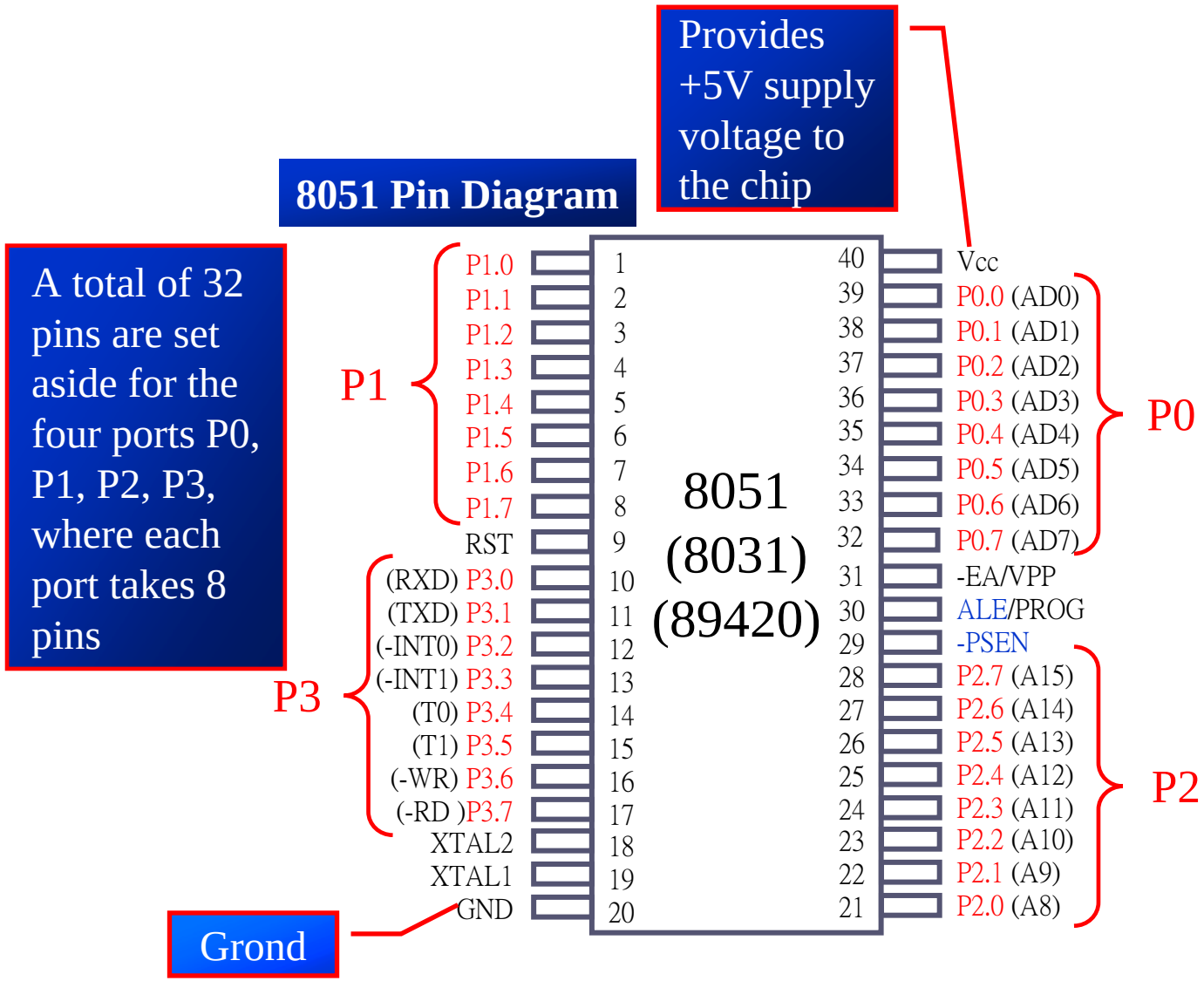
The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

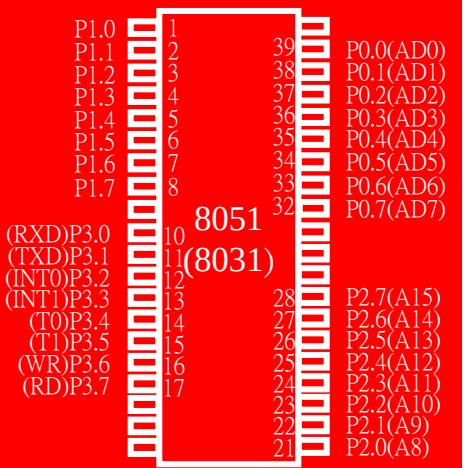
Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

I/O PROGRAMMING



I/O PROGRAMMING

I/O Port Pins

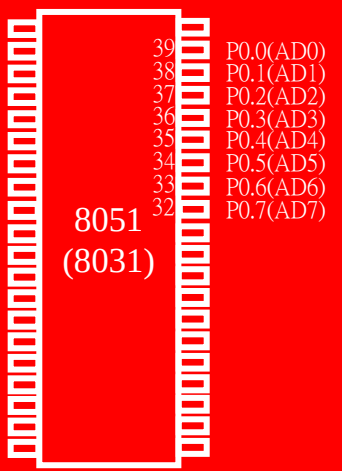


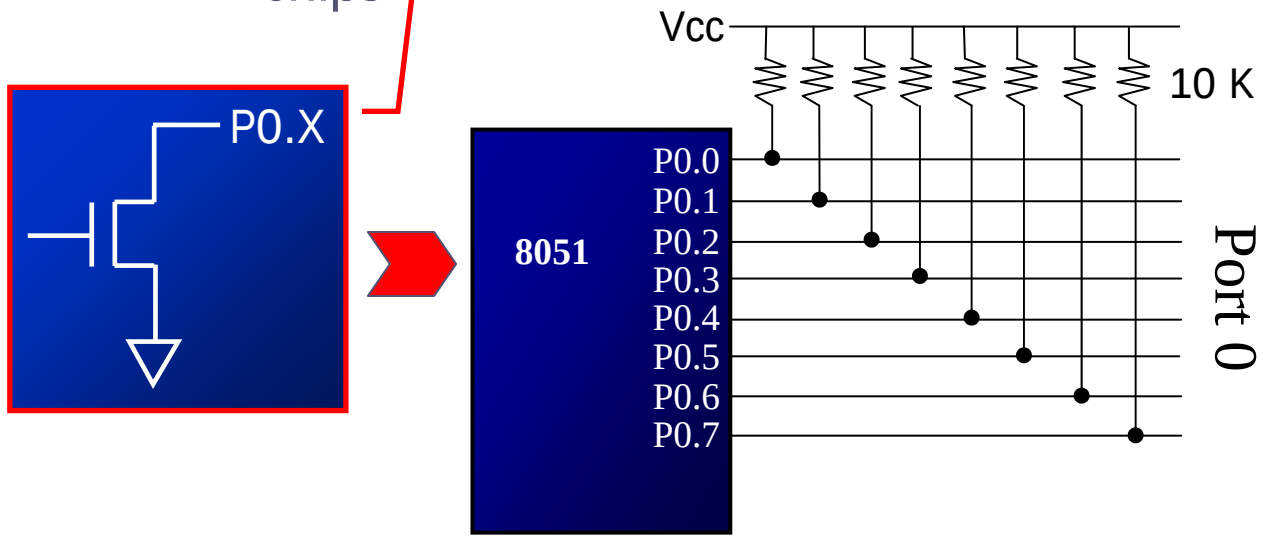
- The four 8-bit I/O ports P0, P1, P2 and P3 each uses 8 pins
- All the ports upon RESET are configured as input, ready to be used as input ports
  - When the first 0 is written to a port, it becomes an output
  - To reconfigure it as an input, a 1 must be sent to the port
    - To use any of these ports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



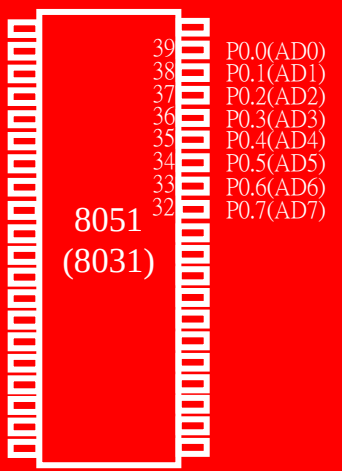


I/O PROGRAMMING

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
```



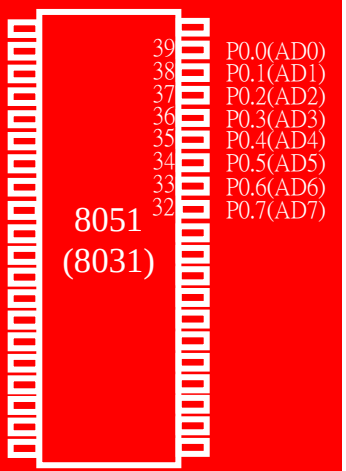
I/O PROGRAMMING

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

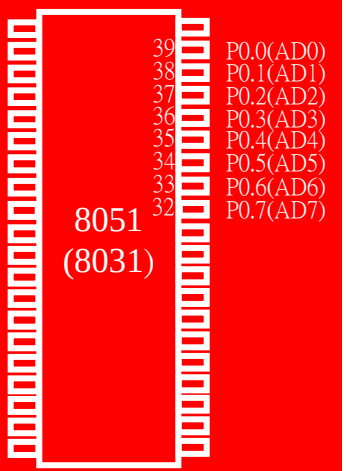
```
        MOV     A,#0FFH         ;A=FF hex
        MOV     P0,A            ;make P0 an i/p port
                                ;by writing it all 1s
BACK:   MOV     A,P0            ;get data from P0
        MOV     P1,A            ;send it to port 1
        SJMP    BACK            ;keep doing it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



I/O PROGRAMMING

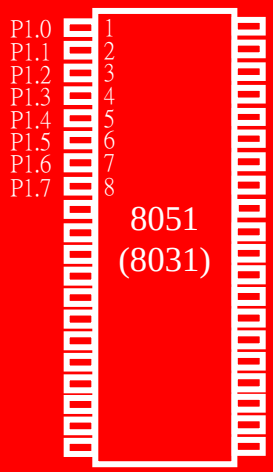
Dual Role of Port 0

- 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
```

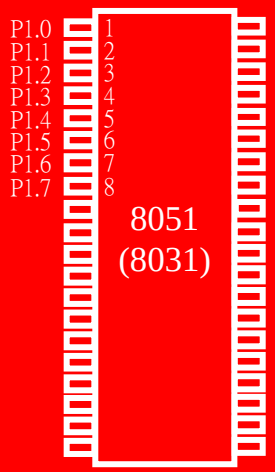
I/O PROGRAMMING

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

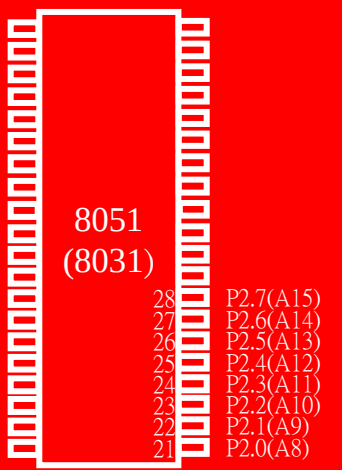
```
        MOV     A,#0FFH     ;A=FF hex
        MOV     P1,A        ;make P1 an input port
                            ;by writing it all 1s
        MOV     A,P1        ;get data from P1
        MOV     R7,A        ;save it to in reg R7
        ACALL   DELAY       ;wait
        MOV     A,P1       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



I/O PROGRAMMING

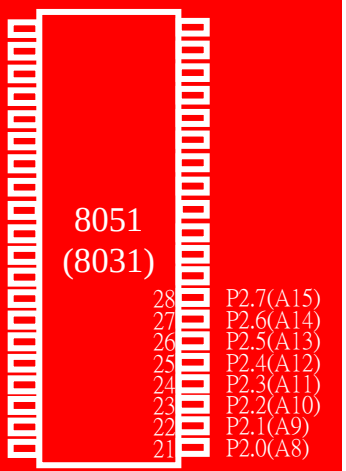
Port 2

- 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

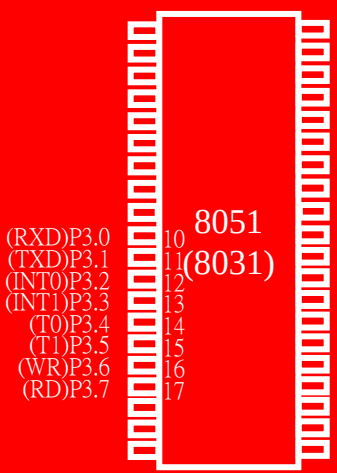


- To make port 2 an input port, it must be programmed as such by writing 1 to all its bits
- In many 8051-based system, P2 is used as simple I/O
- In 8031-based systems, port 2 must be used along with P0 to provide the 16-bit address for the external memory
  - Port 2 is also designated as A8 – A15,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



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

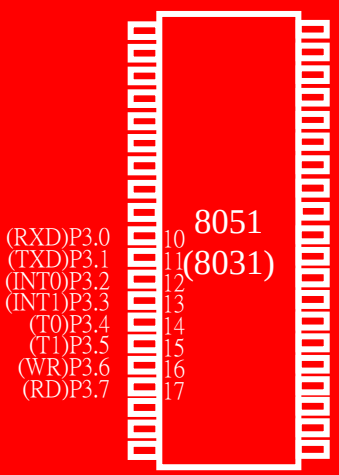
Serial communications

External interrupts

Timers

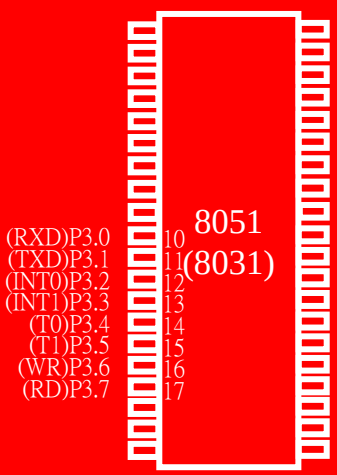
Read/Write signals of external memories

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

Example 4-2

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

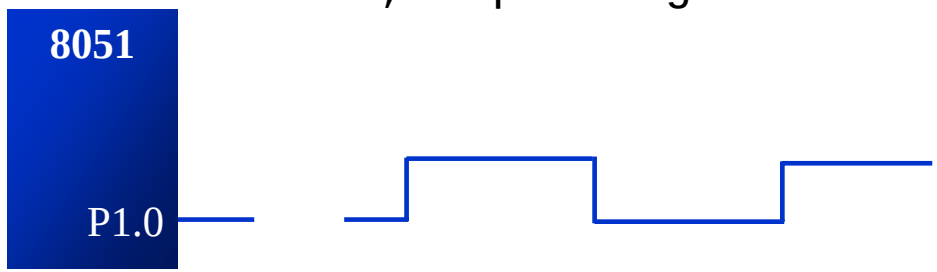
Solution:

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
```



- 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

- 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
```

ADDRESSING MODES

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

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

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

- 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

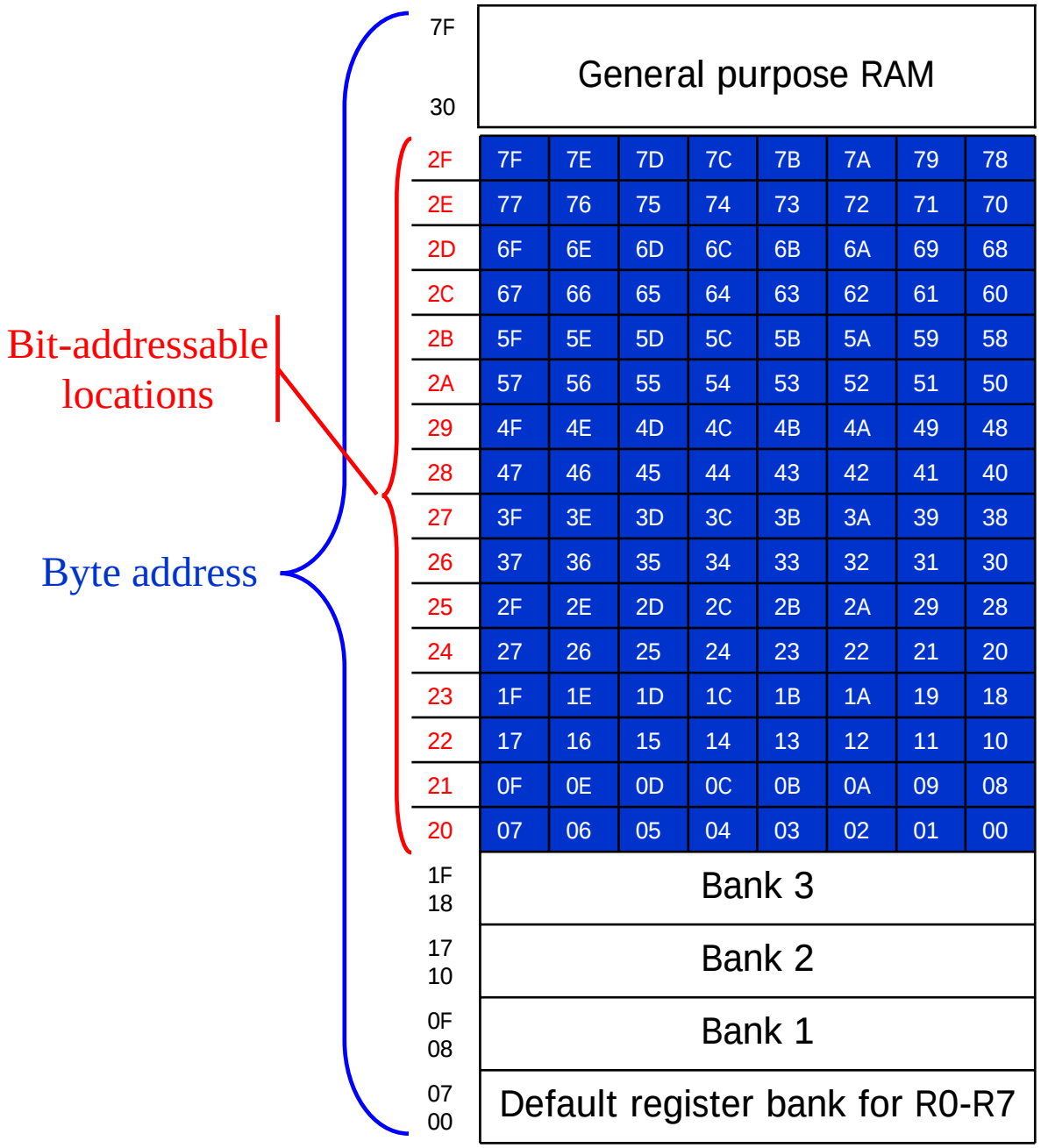
BIT ADDRESSES

Bit-Addressable RAM

- The bit-addressable RAM location are 20H to 2FH
  - These 16 bytes provide 128 bits of RAM bit-addressability, since 16 × 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
```

ARITHMETIC & LOGIC INSTRUCTIONS AND PROGRAMS

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

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
```

- 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

- 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.

```
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.

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

- 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

- 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.

- 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

- 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

```
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

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

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

OR

```
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

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

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

```
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

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

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.

```
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')

```
        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

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

- 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

- 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.

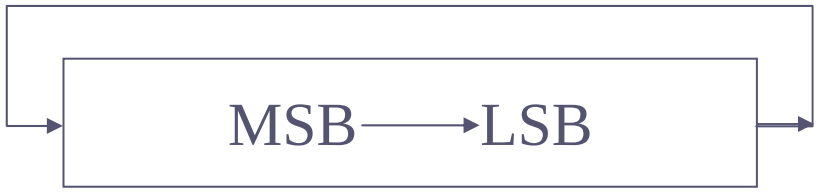
If T = 75 then A = 75
If T < 75 then R1 = T
If T > 75 then R2 = T

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:   ...
```

```
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
```

```
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
```

```
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
```

```
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
```

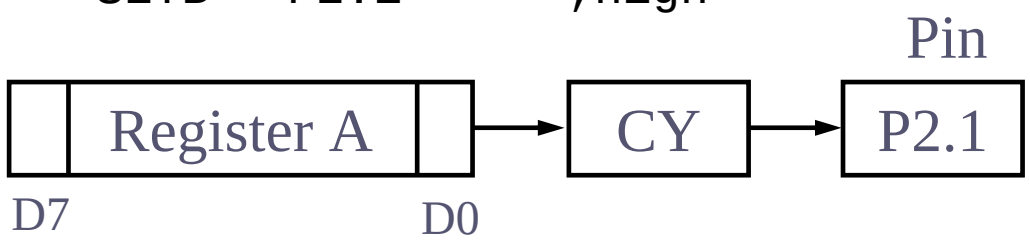
- Serializing data is a way of sending a byte of data one bit at a time through a single pin of microcontroller
  - Using the serial port, discussed in Chapter 10
  - To transfer data one bit at a time and control the sequence of data and spaces in between them

- 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
```



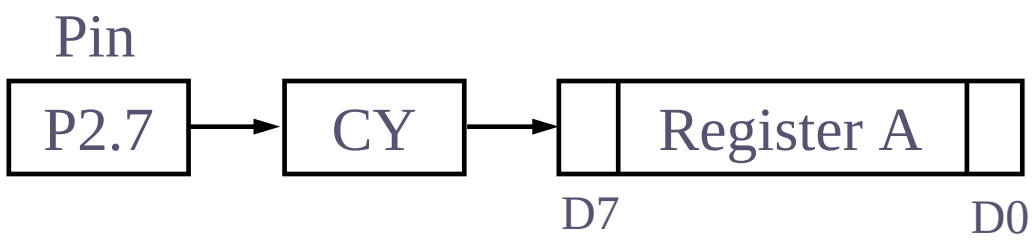
ROTATE INSTRUCTION AND DATA SERIALIZATION

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
```

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
```

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

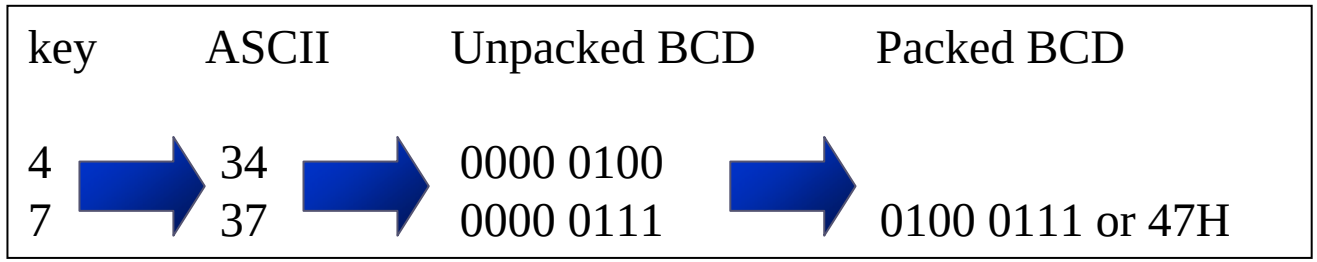
- The DS5000T microcontrollers have a real-time clock (RTC)
  - The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off
- However this data is provided in packed BCD
  - To be displayed on an LCD or printed by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




```
        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
```

- 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     The checksum is calculated by first adding the
+   3FH     bytes. The sum is 118H, and dropping the carry,
+   52H     we get 18H. The checksum byte is the 2's
-------     complement of 18H, which is E8H
   118H
```

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

```
    25H
+   62H     Adding the series of bytes including the checksum
+   3FH     byte must result in zero. This indicates that all the
+   52H     bytes are unchanged and no byte is corrupted.
+   E8H
-------
   200H (dropping the carries)
```

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

```
    25H
+   22H     Adding the series of bytes including the checksum
+   3FH     byte shows that the result is not zero, which indicates
+   52H     that one or more bytes have been corrupted.
+   E8H
-------
  1C0H (dropping the carry, we get C0H)
```

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

8051 PROGRAMMING IN C

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

WHY PROGRAM 8051 IN C

- Compilers produce hex files that is downloaded to ROM of microcontroller
  - The size of hex file is the main concern
    - Microcontrollers have limited on-chip ROM
    - Code space for 8051 is limited to 64K bytes
- C programming is less time consuming, but has larger hex file size
- The reasons for writing programs in C
  - It is easier and less time consuming to write in C than Assembly
  - C is easier to modify and update
  - You can use code available in function libraries
  - C code is portable to other microcontroller with little of no modification

DATA TYPES

- A good understanding of C data types for 8051 can help programmers to create smaller hex files
  - Unsigned char
  - Signed char
  - Unsigned int
  - Signed int
  - Sbit (single bit)
  - Bit and sfr

DATA TYPES

Unsigned char

- The character data type is the most natural choice
  - 8051 is an 8-bit microcontroller
- Unsigned char is an 8-bit data type in the range of 0 – 255 (00 – FFH)
  - One of the most widely used data types for the 8051
    - Counter value
    - ASCII characters
- C compilers use the signed char as the default if we do not put the keyword *unsigned*

DATA TYPES

Unsigned char (cont')

Write an 8051 C program to send values 00 – FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char z;
    for (z=0;z<=255;z++)
       P1=z;
  }
```

1. Pay careful attention to the size of the data
2. Try to use unsigned *char* instead of *int* if possible

Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mynum[]="012345ABCD";
    unsigned char z;
    for (z=0;z<=10;z++)
       P1=mynum[z];
  }
```

DATA TYPES

Unsigned char (cont')

Write an 8051 C program to toggle all the bits of P1 continuously.

Solution:

```
//Toggle P1 forever
#include <reg51.h>
void main(void)
  {
    for (;;)
      {
         p1=0x55;
         p1=0xAA;
       }
  }
```

DATA TYPES

Signed char

- The signed char is an 8-bit data type
  - Use the MSB D7 to represent – or +
  - Give us values from –128 to +127
- We should stick with the unsigned char unless the data needs to be represented as signed numbers
  - temperature

Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
//Singed numbers
#include <reg51.h>
void main(void)
  {
    char mynum[]={+1,-1,+2,-2,+3,-3,+4,-4};
    unsigned char z;
    for (z=0;z<=8;z++)
       P1=mynum[z];
  }
```

DATA TYPES

Unsigned and Signed int

- The unsigned int is a 16-bit data type
  - Takes a value in the range of 0 to 65535 (0000 – FFFFH)
  - Define 16-bit variables such as memory addresses
  - Set counter values of more than 256
  - Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file
- Signed int is a 16-bit data type
  - Use the MSB D15 to represent – or +
  - We have 15 bits for the magnitude of the number from –32768 to +32767

DATA TYPES

Single Bit (cont')

Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

Solution:

```
#include <reg51.h>
sbit MYBIT=P1^0;

void main(void)
  {
    unsigned int z;
    for (z=0;z<=50000;z++)
       {
         MYBIT=0;
         MYBIT=1;
       }
  }
```

sbit keyword allows access to the single bits of the SFR registers

DATA TYPES

Bit and sfr

- The bit data type allows access to single bits of bit-addressable memory spaces 20 – 2FH
- To access the byte-size SFR registers, we use the sfr data type

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
(signed) char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65535
(signed) int	16-bit	-32768 to +32767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 – FFH only

TIME DELAY

- There are two way s to create a time delay in 8051 C
  - Using the 8051 timer (Chap. 9)
  - Using a simple for loop

    be mindful of three factors that can affect the accuracy of the delay
    - The 8051 design
      - The number of machine cycle
      - The number of clock periods per machine cycle
    - The crystal frequency connected to the X1 – X2 input pins
    - Compiler choice
      - C compiler converts the C statements and functions to Assembly language instructions
      - Different compilers produce different code

TIME DELAY (cont')

Write an 8051 C program to toggle bits of P1 continuously forever with some delay.

Solution:

```
//Toggle P1 forever with some delay in between
//"on" and "off"
#include <reg51.h>
void main(void)
  {
    unsigned int x;
    for (;;)                    //repeat forever
      {
        p1=0x55;
        for (x=0;x<40000;x++); //delay size
                               //unknown
        p1=0xAA;
        for (x=0;x<40000;x++);
       }
  }
```

We must use the oscilloscope to measure the exact duration

TIME DELAY (cont')

Write an 8051 C program to toggle bits of P1 ports continuously with a 250 ms.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    while (1)                 //repeat forever
       {
         p1=0x55;
         MSDelay(250);
         p1=0xAA;
         MSDelay(250);
       }
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i,j;
    for (i=0;i<itime;i++)
       for (j=0;j<1275;j++);
  }
```

I/O PROGRAMMING

Byte Size I/O

LEDs are connected to bits P1 and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

```
#include <reg51.h>
#defind LED P2;

void main(void)
  {
    P1=00;              //clear P1
    LED=0;              //clear P2
    for (;;)            //repeat forever
       {
         P1++;          //increment P1
         LED++;         //increment P2
       }
  }
```

Ports P0 – P3 are byte-accessable and we use the P0 – P3 labels as defined in the 8051/52 header file.

I/O PROGRAMMING

Byte Size I/O (cont')

Write an 8051 C program to get a byte of data form P1, wait 1/2 second, and then send it to P2.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);

void main(void)
  {
    unsigned char mybyte;
    P1=0xFF;                  //make P1 input port
    while (1)
       {
         mybyte=P1;           //get a byte from P1
         MSDelay(500);
         P2=mybyte;           //send it to P2
       }
  }
```

I/O PROGRAMMING

Byte Size I/O (cont')

Write an 8051 C program to get a byte of data form P0. If it is less than 100, send it to P1; otherwise, send it to P2.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char mybyte;
    P0=0xFF;                  //make P0 input port
    while (1)
       {
         mybyte=P0;           //get a byte from P0
         if (mybyte<100)
            P1=mybyte;        //send it to P1
         else
            P2=mybyte;        //send it to P2
       }
  }
```

I/O PROGRAMMING

Bit-addressable I/O

Write an 8051 C program to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2.

Solution:

```
//Toggling an individual bit
#include <reg51.h>
sbit mybit=P2^4;

void main(void)
  {
    while (1)
       {
         mybit=1;            //turn on P2.4
         mybit=0;            //turn off P2.4
       }
  }
```

Ports P0 – P3 are bit-addressable and we use *sbit* data type to access a single bit of P0 - P3

Use the Px^y format, where x is the port 0, 1, 2, or 3 and y is the bit 0 – 7 of that port

I/O PROGRAMMING

Bit-addressable I/O (cont')

Write an 8051 C program to monitor bit P1.5. If it is high, send 55H to P0; otherwise, send AAH to P2.

Solution:

```
#include <reg51.h>
sbit mybit=P1^5;

void main(void)
  {
    mybit=1;                  //make mybit an input
    while (1)
       {
         if (mybit==1)
            P0=0x55;
         else
            P2=0xAA;
       }
  }
```

I/O PROGRAMMING

Bit-addressable I/O (cont')

A door sensor is connected to the P1.1 pin, and a buzzer is connected to P1.7. Write an 8051 C program to monitor the door sensor, and when it opens, sound the buzzer. You can sound the buzzer by sending a square wave of a few hundred Hz.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
sbit Dsensor=P1^1;
sbit Buzzer=P1^7;

void main(void)
  {
    Dsensor=1;              //make P1.1 an input
    while (1)
       {
         while (Dsensor==1)//while it opens
           {
              Buzzer=0;
              MSDelay(200);
              Buzzer=1;
              MSDelay(200);
           }
       }
  }
```

I/O PROGRAMMING

Bit-addressable I/O (cont')

The data pins of an LCD are connected to P1. The information is latched into the LCD whenever its Enable pin goes from high to low. Write an 8051 C program to send "The Earth is but One Country" to this LCD.

Solution:

```
#include <reg51.h>
#define LCDData P1  //LCDData declaration
sbit En=P2^0;       //the enable pin

void main(void)
  {
    unsigned char message[]
              ="The Earth is but One Country";
    unsigned char z;
    for (z=0;z<28;z++) //send 28 characters
       {
         LCDData=message[z];
         En=1;    //a high-
         En=0;    //-to-low pulse to latch data
       }
  }
```

I/O PROGRAMMING

Accessing SFR Addresses 80 - FFH

Write an 8051 C program to toggle all the bits of P0, P1, and P2 continuously with a 250 ms delay. Use the sfr keyword to declare the port addresses.

Another way to access the SFR RAM space 80 – FFH is to use the *sfr* data type

Solution:

```
//Accessing Ports as SFRs using sfr data type
sfr P0=0x80;
sfr P1=0x90;
sfr P2=0xA0;
void MSDelay(unsigned int);

void main(void)
  {
    while (1)
       {
         P0=0x55;
         P1=0x55;
         P2=0x55;
         MSDelay(250);
         P0=0xAA;
         P1=0xAA;
         P2=0xAA;
         MSDelay(250);
       }
  }
```

I/O PROGRAMMING

Accessing SFR Addresses 80 - FFH (cont')

Write an 8051 C program to turn bit P1.5 on and off 50,000 times.

Solution:

```
sbit MYBIT=0x95;

void main(void)
  {
    unsigned int z;
    for (z=0;z<50000;z++)
       {
         MYBIT=1;
         MYBIT=0;
       }
  }
```

We can access a single bit of any SFR if we specify the bit address

Notice that there is no `#include <reg51.h>`. This allows us to access any byte of the SFR RAM space 80 – FFH. This is widely used for the new generation of 8051 microcontrollers.

I/O PROGRAMMING

Using bit Data Type for Bit-addressable RAM

Write an 8051 C program to get the status of bit P1.0, save it, and send it to P2.7 continuously.

Solution:

```
#include <reg51.h>
sbit inbit=P1^0;
sbit outbit=P2^7;
bit membit;             //use bit to declare
                        //bit- addressable memory

void main(void)
  {
    while (1)
       {
         membit=inbit;    //get a bit from P1.0
         outbit=membit;   //send it to P2.7
       }
  }
```

We use bit data type to access data in a bit-addressable section of the data RAM space 20 – 2FH

LOGIC OPERATIONS

Bit-wise Operators in C

- Logical operators
  - AND (&&), OR (||), and NOT (!)
- Bit-wise operators
  - AND (&), OR (|), EX-OR (^), Inverter (~), Shift Right (>>), and Shift Left (<<)
    - These operators are widely used in software engineering for embedded systems and control

Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

LOGIC OPERATIONS

Bit-wise Operators in C (cont')

Run the following program on your simulator and examine the results.

Solution:

```
#include <reg51.h>

void main(void)
  {
    P0=0x35 & 0x0F;          //ANDing
    P1=0x04 | 0x68;          //ORing
    P2=0x54 ^ 0x78;          //XORing
    P0=~0x55;                //inversing
    P1=0x9A >> 3;            //shifting right 3
    P2=0x77 >> 4;            //shifting right 4
    P0=0x6 << 4;             //shifting left 4
  }
```

LOGIC OPERATIONS

Bit-wise Operators in C (cont')

Write an 8051 C program to toggle all the bits of P0 and P2 continuously with a 250 ms delay. Using the inverting and Ex-OR operators, respectively.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);

void main(void)
  {
    P0=0x55;
    P2=0x55;
    while (1)
       {
         P0=~P0;
         P2=P2^0xFF;
         MSDelay(250);
       }
  }
```

LOGIC OPERATIONS

Bit-wise Operators in C (cont')

Write an 8051 C program to get bit P1.0 and send it to P2.7 after inverting it.

Solution:

```
#include <reg51.h>
sbit inbit=P1^0;
sbit outbit=P2^7;
bit membit;

void main(void)
  {
    while (1)
       {
         membit=inbit;   //get a bit from P1.0
         outbit=~membit; //invert it and send
                          //it to P2.7
       }
  }
```

LOGIC OPERATIONS

Bit-wise Operators in C (cont')

Write an 8051 C program to read the P1.0 and P1.1 bits and issue an ASCII character to P0 according to the following table.

P1.1	P1.0	
0	0	send '0' to P0
0	1	send '1' to P0
1	0	send '2' to P0
1	1	send '3' to P0

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsignbed char z;
    z=P1;
    z=z&0x3;

...
```

LOGIC OPERATIONS

Bit-wise Operators in C (cont')

```
...
switch (z)
        {
          case(0):
            {
               P0='0';
               break;
            }
          case(1):
            {
               P0='1';
               break;
            }
          case(2):
            {
               P0='2';
               break;
            }
          case(3):
            {
               P0='3';
               break;
            }
        }
   }
```

DATA CONVERSION

Packed BCD to ASCII Conversion

Write an 8051 C program to convert packed BCD 0x29 to ASCII and display the bytes on P1 and P2.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char x,y,z;
    unsigned char mybyte=0x29;
    x=mybyte&0x0F;
    P1=x|0x30;
    y=mybyte&0xF0;
    y=y>>4;
    P2=y|0x30;
  }
```

DATA CONVERSION

ASCII to Packed BCD Conversion

Write an 8051 C program to convert ASCII digits of '4' and '7' to packed BCD and display them on P1.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char bcdbyte;
    unsigned char w='4';
    unsigned char z='7';
    w=w&0x0F;
    w=w<<4;
    z=z&0x0F;
    bcdbyte=w|z;
    P1=bcdbyte;
  }
```

DATA CONVERSION

Checksum Byte in ROM

Write an 8051 C program to calculate the checksum byte for the data 25H, 62H, 3FH, and 52H.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char mydata[]={0x25,0x62,0x3F,0x52};
    unsigned char sum=0;
    unsigned char x;
    unsigned char chksumbyte;
    for (x=0;x<4;x++)
       {
         P2=mydata[x];
         sum=sum+mydata[x];
         P1=sum;
       }
    chksumbyte=~sum+1;
    P1=chksumbyte;
  }
```

DATA CONVERSION

Checksum Byte in ROM (cont')

Write an 8051 C program to perform the checksum operation to ensure data integrity. If data is good, send ASCII character 'G' to P0. Otherwise send 'B' to P0.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char mydata[]
                    ={0x25,0x62,0x3F,0x52,0xE8};
    unsigned char shksum=0;
    unsigned char x;
    for (x=0;x<5;x++)
       chksum=chksum+mydata[x];
    if (chksum==0)
       P0='G';
    else
       P0='B';
}
```

DATA CONVERSION

Binary (hex) to Decimal and ASCII Conversion

Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on P0, P1 and P2.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char x,binbyte,d1,d2,d3;
    binbyte=0xFD;
    x=binbyte/10;
    d1=binbyte%10;
    d2=x%10;
    d3=x/10;
    P0=d1;
    P1=d2;
    P2=d3;
  }
```

ACCESSING CODE ROM

RAM Data Space Usage by 8051 C Compiler

- The 8051 C compiler allocates RAM locations
  - Bank 0 – addresses 0 – 7
  - Individual variables – addresses 08 and beyond
  - Array elements – addresses right after variables
    - Array elements need contiguous RAM locations and that limits the size of the array due to the fact that we have only 128 bytes of RAM for everything
  - Stack – addresses right after array elements

ACCESSING CODE ROM

RAM Data Space Usage by 8051 C Compiler (cont')

Compile and single-step the following program on your 8051 simulator. Examine the contents of the 128-byte RAM space to locate the ASCII values.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char mynum[]="ABCDEF"; //RAM space
    unsigned char z;
    for (z=0;z<=6;z++)
       P1=mynum[z];
  }
```

ACCESSING CODE ROM

RAM Data Space Usage by 8051 C Compiler (cont')

Write, compile and single-step the following program on your 8051 simulator. Examine the contents of the code space to locate the values.

Solution:

```
#include <reg51.h>

void main(void)
  {
    unsigned char mydata[100]; //RAM space
    unsigned char x,z=0;
    for (x=0;x<100;x++)
       {
         z--;
         mydata[x]=z;
         P1=z;
       }
  }
```

ACCESSING CODE ROM

8052 RAM Data Space

- One of the new features of the 8052 was an extra 128 bytes of RAM space
  - The extra 128 bytes of RAM helps the 8051/52 C compiler to manage its registers and resources much more effectively
- We compile the C programs for the 8052 microcontroller
  - Use the reg52.h header file
  - Choose the8052 option when compiling the program

ACCESSING CODE ROM (cont')

Compile and single-step the following program on your 8051 simulator. Examine the contents of the code space to locate the ASCII values.

Solution:

```
#include <reg51.h>

void main(void)
  {
    code unsigned char mynum[]="ABCDEF";
    unsigned char z;
    for (z=0;z<=6;z++)
       P1=mynum[z];
  }
```

To make the C compiler use the code space instead of the RAM space, we need to put the keyword code in front of the variable declaration

ACCESSING CODE ROM (cont')

Compare and contrast the following programs and discuss the advantages and disadvantages of each one.

(a)

```
#include <reg51.h>
void main(void)
  {
    P1='H';
    P1='E';
    P1='L';
    P1='L';
    P1='O';
  }

...
```

Short and simple, but the individual characters are embedded into the program and it mixes the code and data together

ACCESSING CODE ROM (cont')

```
...

(b)
#include <reg51.h>
void main(void)
  {
    unsigned char mydata[]="HELLO";
    unsigned char z;
    for (z=0;z<=5;z++)
       P1=mydata[z];
  }

(c)
#include <reg51.h>
void main(void)
  {
    code unsigned char mydata[]="HELLO";
    unsigned char z;
    for (z=0;z<=5;z++)
       P1=mydata[z];
  }
```

Use the RAM data space to store array elements, therefore the size of the array is limited

Use a separate area of the code space for data. This allows the size of the array to be as long as you want if you have the on-chip ROM.

However, the more code space you use for data, the less space is left for your program code

DATA SERIALIZATION

- 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 (Chap. 10)
  - Transfer data one bit a time and control the sequence of data and spaces in between them
    - In many new generations of devices such as LCD, ADC, and ROM the serial versions are becoming popular since they take less space on a PCB

DATA SERIALIZATION (cont')

Write a C program to send out the value 44H serially one bit at a time via P1.0. The LSB should go out first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regALSB=ACC^0;

void main(void)
  {
    unsigned char conbyte=0x44;
    unsigned char x;
    ACC=conbyte;
    for (x=0;x<8;x++)
       {
         P1b0=regALSB;
         ACC=ACC>>1;
       }
  }
```

DATA SERIALIZATION (cont')

Write a C program to send out the value 44H serially one bit at a time via P1.0. The MSB should go out first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regAMSB=ACC^7;

void main(void)
  {
    unsigned char conbyte=0x44;
    unsigned char x;
    ACC=conbyte;
    for (x=0;x<8;x++)
       {
         P1b0=regAMSB;
         ACC=ACC<<1;
       }
  }
```

DATA SERIALIZATION (cont')

Write a C program to bring in a byte of data serially one bit at a time via P1.0. The LSB should come in first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit ACCMSB=ACC^7;
bit membit;

void main(void)
  {
    unsigned char x;
    for (x=0;x<8;x++)
       {
         membit=P1b0;
         ACC=ACC>>1;
         ACCMSB=membit;
       }
    P2=ACC;
  }
```

DATA SERIALIZATION (cont')

Write a C program to bring in a byte of data serially one bit at a time via P1.0. The MSB should come in first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regALSB=ACC^0;
bit membit;

void main(void)
  {
    unsigned char x;
    for (x=0;x<8;x++)
       {
         membit=P1b0;
         ACC=ACC<<1;
         regALSB=membit;
       }
    P2=ACC;
  }
```

HARDWARE CONNECTION AND INTEL HEX FILE

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

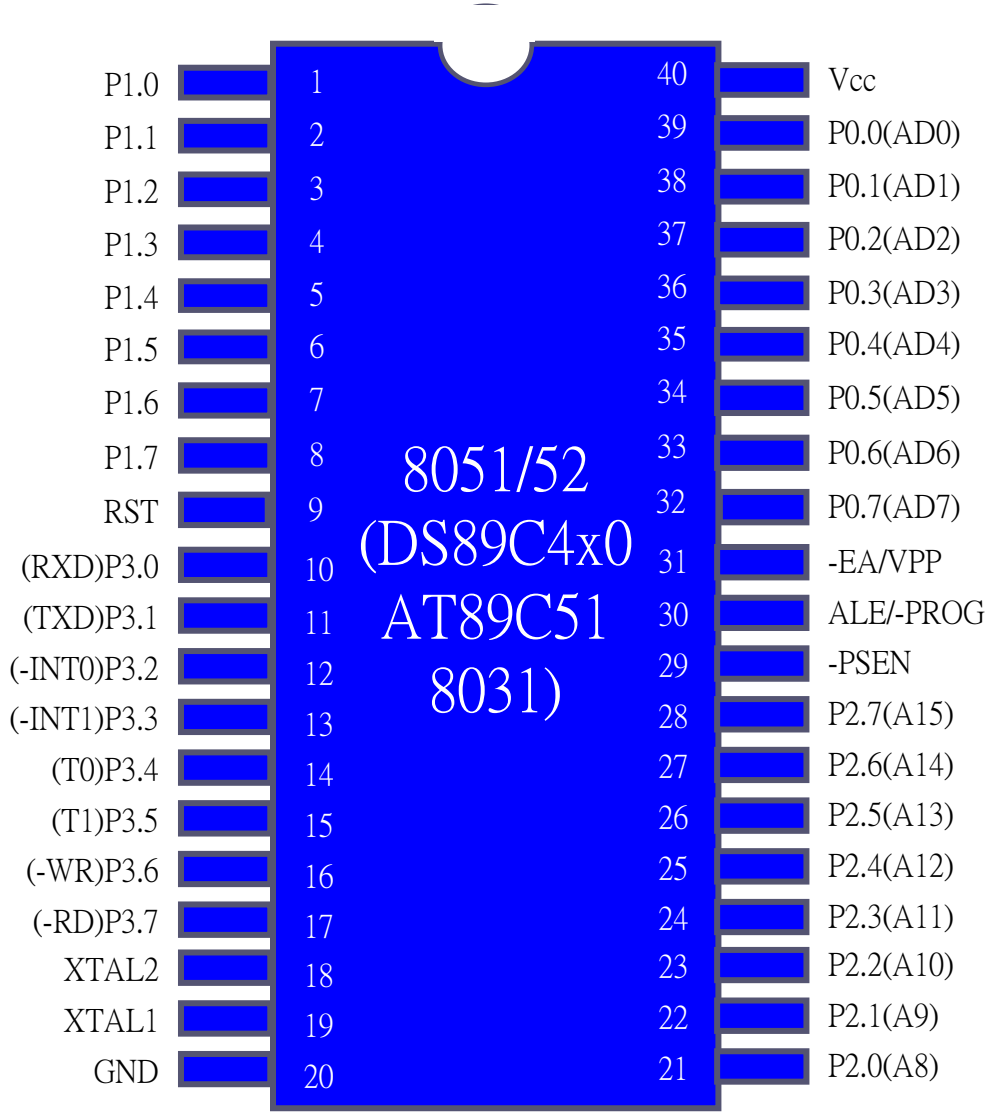
Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

PIN DESCRIPTION

- 8051 family members (e.g, 8751, 89C51, 89C52, DS89C4x0)
  - Have 40 pins dedicated for various functions such as I/O, -RD, -WR, address, data, and interrupts
  - Come in different packages, such as
    - DIP(dual in-line package),
    - QFP(quad flat package), and
    - LLC(leadless chip carrier)
  - Some companies provide a 20-pin version of the 8051 with a reduced number of I/O ports for less demanding applications

PIN DESCRIPTION (cont')

8051 pin diagram

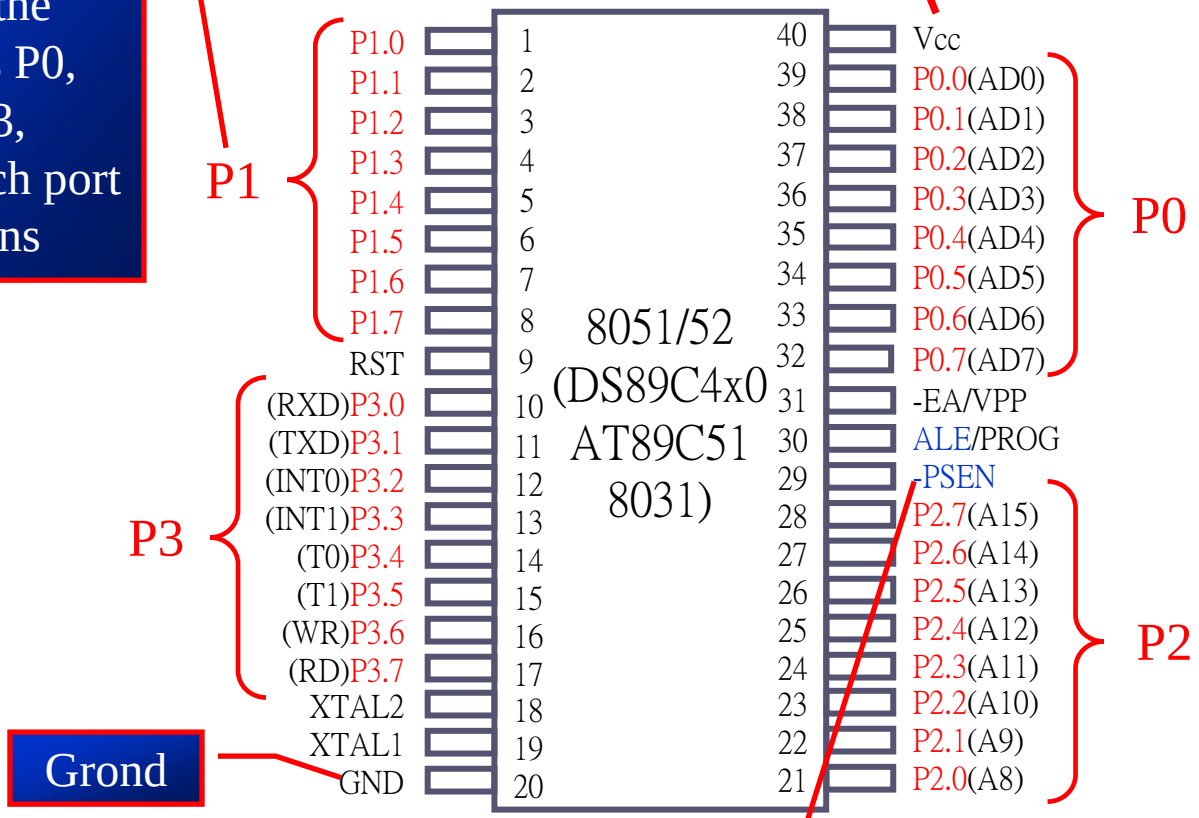


PIN DESCRIPTION (cont')

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

Provides +5V supply voltage to the chip

Vcc, GND, XTAL1, XTAL2, RST, -EA are used by all members of 8051 and 8031 families

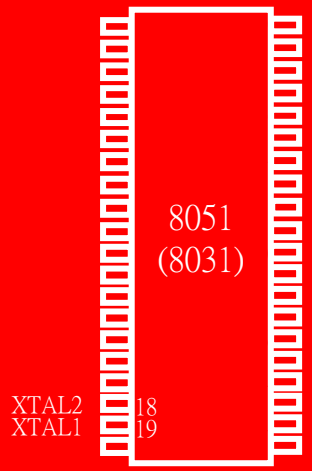


Grond

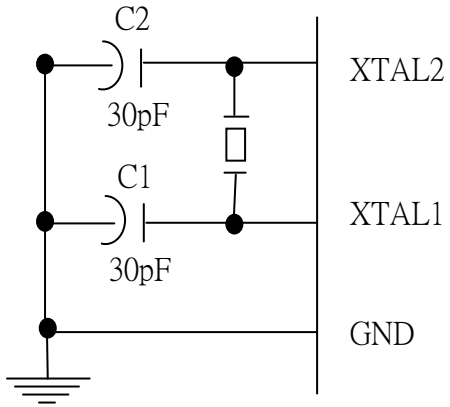
-PSEN and ALE are used mainly in 8031-baded systems

PIN DESCRIPTION

XTAL1 and XTAL2

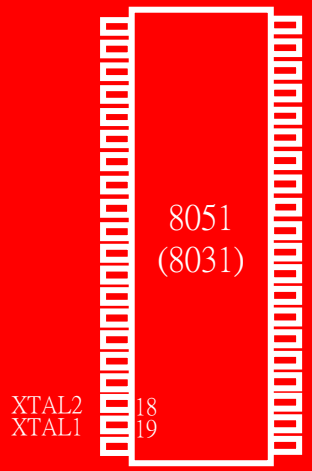


- The 8051 has an on-chip oscillator but requires an external clock to run it
  - A quartz crystal oscillator is connected to inputs XTAL1 (pin19) and XTAL2 (pin18)
    - The quartz crystal oscillator also needs two capacitors of 30 pF value

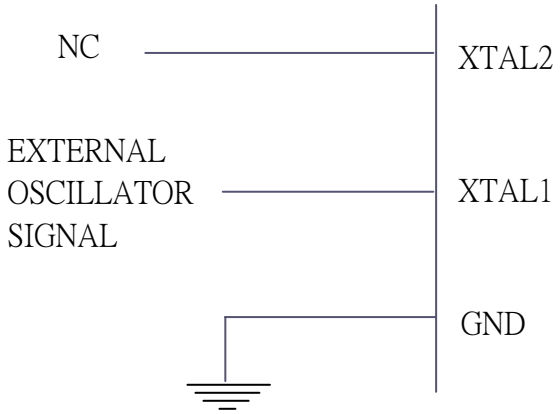


PIN DESCRIPTION

XTAL1 and XTAL2 (cont')

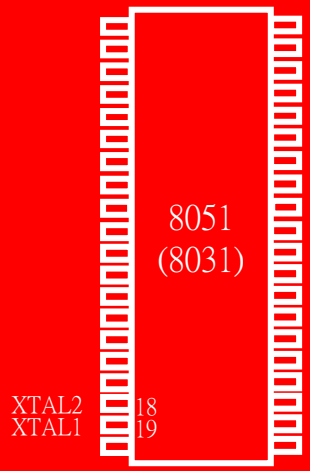


- If you use a frequency source other than a crystal oscillator, such as a TTL oscillator
  - It will be connected to XTAL1
  - XTAL2 is left unconnected



PIN DESCRIPTION

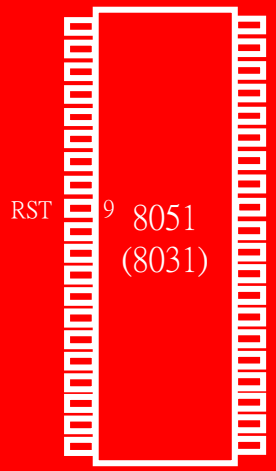
XTAL1 and XTAL2 (cont')



- The speed of 8051 refers to the maximum oscillator frequency connected to XTAL
  - ex. A 12-MHz chip must be connected to a crystal with 12 MHz frequency or less
  - We can observe the frequency on the XTAL2 pin using the oscilloscope

PIN DESCRIPTION

RST



- RESET pin is an input and is active high (normally low)
  - Upon applying a high pulse to this pin, the microcontroller will reset and terminate all activities
    - This is often referred to as a *power-on* reset
    - Activating a power-on reset will cause all values in the registers to be lost

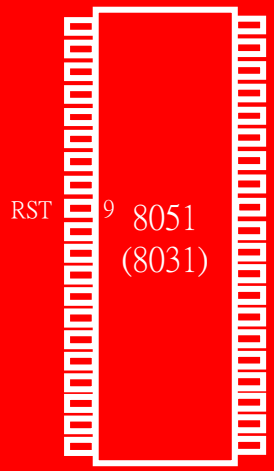
RESET value of some 8051 registers

we must place the first line of source code in ROM location 0

Register	Reset Value
PC	0000
DPTR	0000
ACC	00
PSW	00
SP	07
B	00
P0-P3	FF

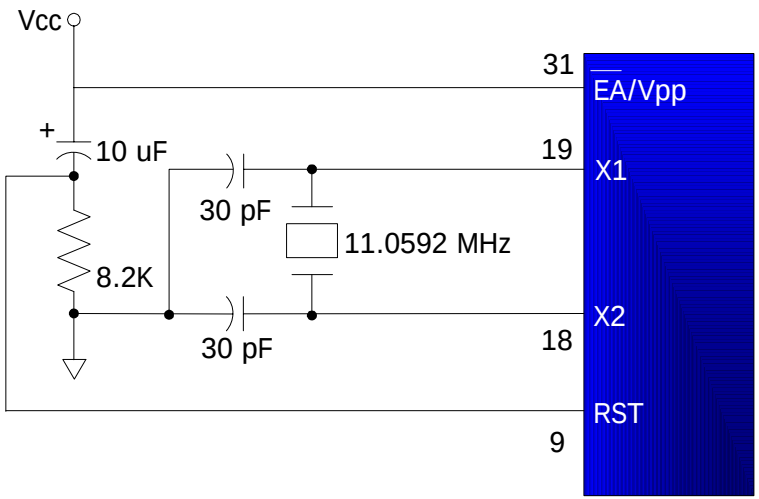
PIN DESCRIPTION

RST (cont')

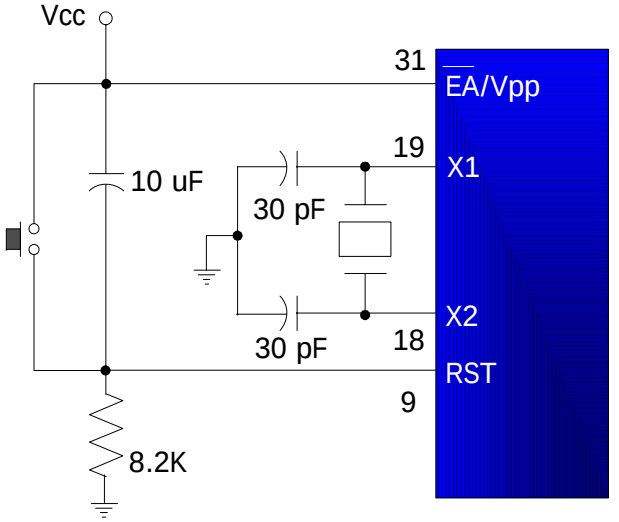


- In order for the RESET input to be effective, it must have a minimum duration of 2 machine cycles
  - In other words, the high pulse must be high for a minimum of 2 machine cycles before it is allowed to go low

Power-on RESET circuit

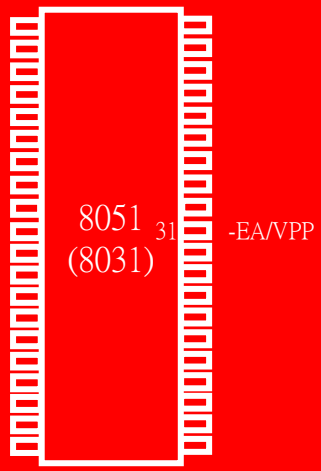


Power-on RESET with debounce



PIN DESCRIPTION

$\overline{EA}$



- EA, “external access”, is an input pin and must be connected to Vcc or GND
  - The 8051 family members all come with on-chip ROM to store programs
    - -EA pin is connected to Vcc
  - The 8031 and 8032 family members do no have on-chip ROM, so code is stored on an external ROM and is fetched by 8031/32
    - -EA pin must be connected to GND to indicate that the code is stored externally

PIN DESCRIPTION

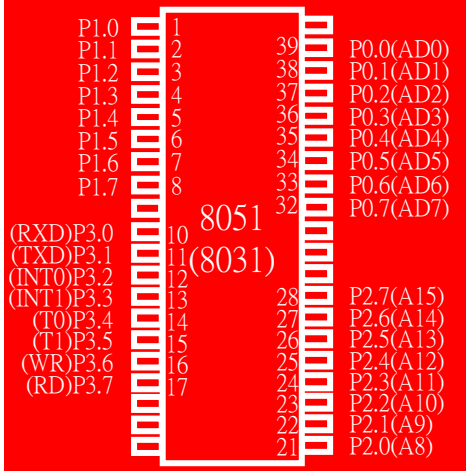
$\overline{\text{PSEN}}$ And ALE

- The following two pins are used mainly in 8031-based systems
- $\overline{\text{PSEN}}$, "program store enable", is an output pin
  - This pin is connected to the OE pin of the ROM
- ALE, "address latch enable", is an output pin and is active high
  - Port 0 provides both address and data
    - The 8031 multiplexes address and data through port 0 to save pins
    - ALE pin is used for demultiplexing the address and data by connecting to the G pin of the 74LS373 chip

PIN DESCRIPTION

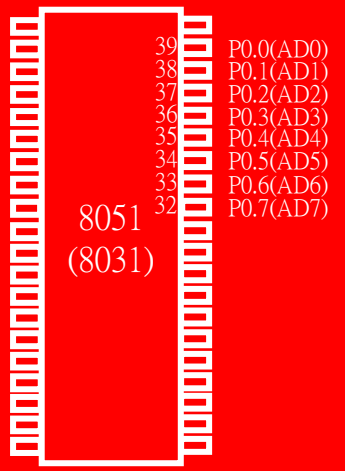
I/O Port Pins

- The four 8-bit I/O ports P0, P1, P2 and P3 each uses 8 pins
- All the ports upon RESET are configured as output, ready to be used as input ports



PIN DESCRIPTION

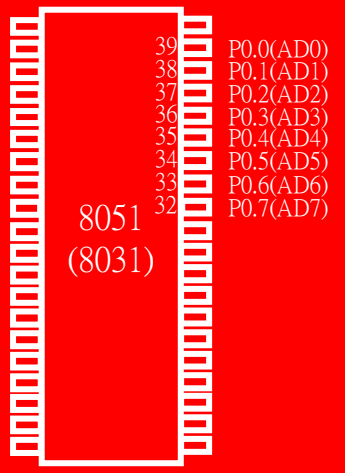
Port 0



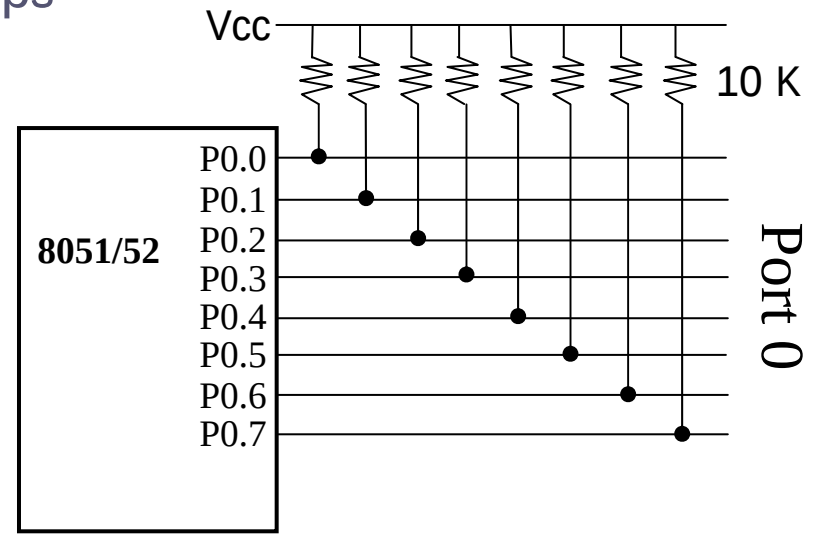
- Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data
  - When connecting an 8051/31 to an external memory, port 0 provides both address and data
  - The 8051 multiplexes address and data through port 0 to save pins
  - ALE indicates if P0 has address or data
    - When ALE=0, it provides data D0-D7
    - When ALE=1, it has address A0-A7

PIN DESCRIPTION

Port 0 (cont')

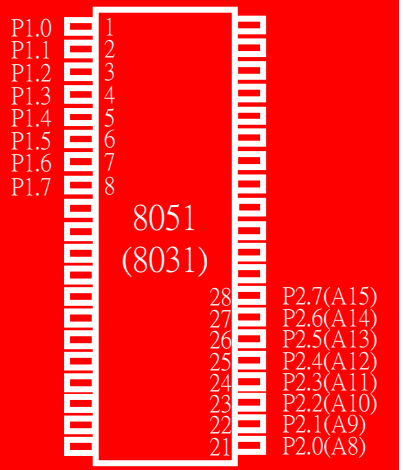


- It can be used for input or output, each pin must be connected externally to a 10K ohm pull-up resistor
  - This is due to the fact that P0 is an open drain, unlike P1, P2, and P3
    - Open drain is a term used for MOS chips in the same way that open collector is used for TTL chips



PIN DESCRIPTION

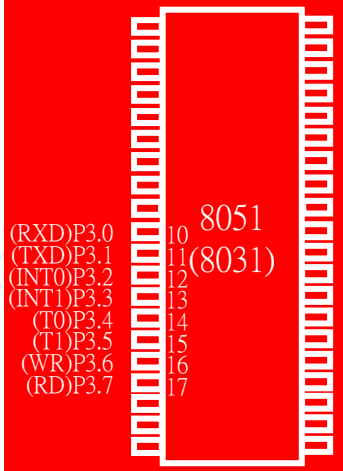
Port 1 and Port 2



- In 8051-based systems with no external memory connection
  - Both P1 and P2 are used as simple I/O
- In 8031/51-based systems with external memory connections
  - Port 2 must be used along with P0 to provide the 16-bit address for the external memory
    - P0 provides the lower 8 bits via A0 – A7
    - P2 is used for the upper 8 bits of the 16-bit address, designated as A8 – A15, and it cannot be used for I/O

PIN DESCRIPTION

Port 3



- Port 3 can be used as input or output
  - Port 3 does not need any pull-up resistors
- 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

EXPLAINING INTEL HEX FILE

- Intel hex file is a widely used file format
  - Designed to standardize the loading of executable machine codes into a ROM chip
- Loaders that come with every ROM burner (programmer) support the Intel hex file format
  - In many newer Windows-based assemblers the Intel hex file is produced automatically (by selecting the right setting)
  - In DOS-based PC you need a utility called OH (object-to-hex) to produce that

EXPLAINING INTEL HEX FILE (cont')

- In the DOS environment
  - The object file is fed into the linker program to produce the abs file
    - The abs file is used by systems that have a monitor program
  - Then the abs file is fed into the OH utility to create the Intel hex file
    - The hex file is used only by the loader of an EPROM programmer to load it into the ROM chip

EXPLAINING INTEL HEX FILE (cont')

The location is the address where the opcodes (object codes) are placed

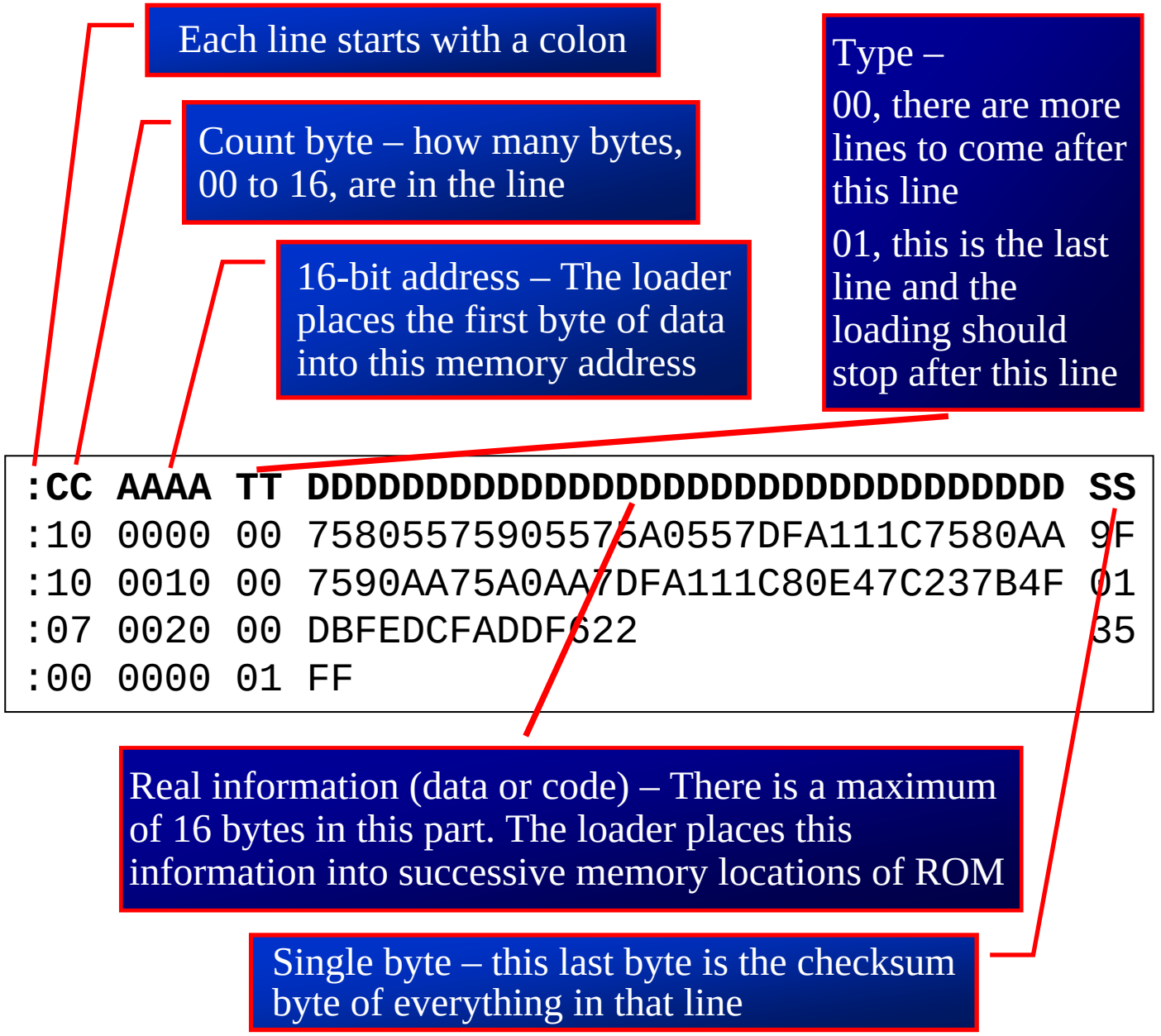
```
LOC  OBJ      LINE
0000          1             ORG 0H
0000 758055   2  MAIN:      MOV P0,#55H
0003 759055   3              MOV P1,#55H
0006 75A055   4              MOV P2,#55H
0009 7DFA     5              MOV R5,#250
000B 111C     6              ACALL MSDELAY
000D 7580AA   7              MOV P0,#0AAH
0010 7590AA   8              MOV P1,#0AAH
0013 75A0AA   9              MOV P2,#0AAH
0016 7DFA     10             MOV R5,#250
0018 111C     11             ACALL MSDELAY
001A 80E4     12             SJMP MAIN
              13 ;--- THE 250 MILLISECOND DELAY.
              14 MSDELAY:
001C 7C23     15 HERE3:      MOV R4,#35
001E 7B4F     16 HERE2:      MOV R3,#79
0020 DBFE     17 HERE1:      DJNZ R3,HERE1
0022 DCFA     18             DJNZ R4,HERE2
0024 DDF6     19             DJNZ R5,HERE3
0026 22       20             RET
              21             END
```

EXPLAINING INTEL HEX FILE (cont')

- The hex file provides the following:
  - The number of bytes of information to be loaded
  - The information itself
  - The starting address where the information must be placed

```
:10000000758055759055 75A0557DFA111C7580AA9F
:10001000 7590AA75A0AA7DFA111C80E47C237B4F01
:07002000 DBFEDCFADDF62235
:00000001 FF

:CC AAAA TT DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD SS
:10 0000 00 758055759055 75A0557DFA111C7580AA 9F
:10 0010 00 7590AA75A0AA7DFA111C80E47C237B4F 01
:07 0020 00 DBFEDCFADDF622                   35
:00 0000 01 FF
```

EXPLAINING INTEL HEX FILE (cont')



EXPLAINING INTEL HEX FILE (cont')

Example 8-4

Verify the checksum byte for line 3 of Figure 8-9. Verify also that the information is not corrupted.

Solution:

```
:07 0020 00 DBFEDCFADDF622                    35
```

07+00+20+00+DB+FE+DC+FA+DD+F6+22=5CBH

If we add all the information including the checksum byte, and drop the carries, we get 00.

5CBH + 35H = 600H

TIMER PROGRAMMING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

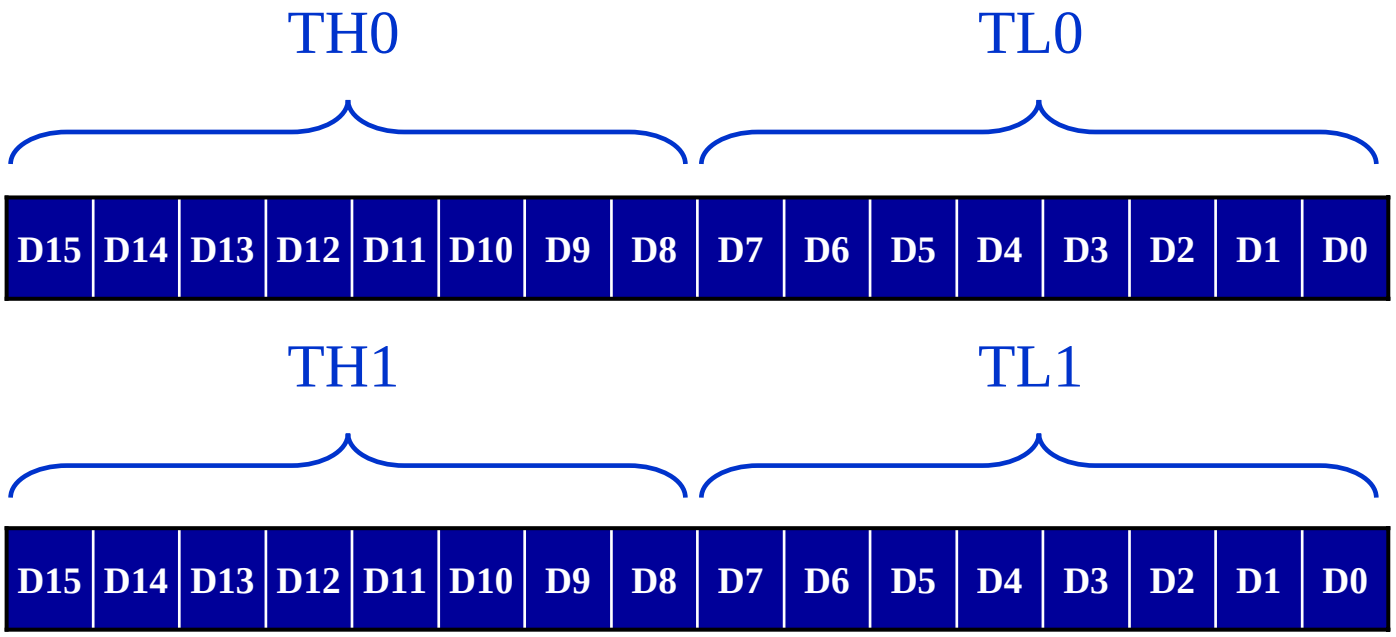
PROGRAMMING TIMERS

- The 8051 has two timers/counters, they can be used either as
  - Timers to generate a time delay or as
  - Event counters to count events happening outside the microcontroller
- Both Timer 0 and Timer 1 are 16 bits wide
  - Since 8051 has an 8-bit architecture, each 16-bits timer is accessed as two separate registers of low byte and high byte

PROGRAMMING TIMERS

Timer 0 & 1 Registers

- Accessed as low byte and high byte
  - The low byte register is called TL0/TL1 and
  - The high byte register is called TH0/TH1
  - Accessed like any other register
    - `MOV TL0,#4FH`
    - `MOV R5,TH0`



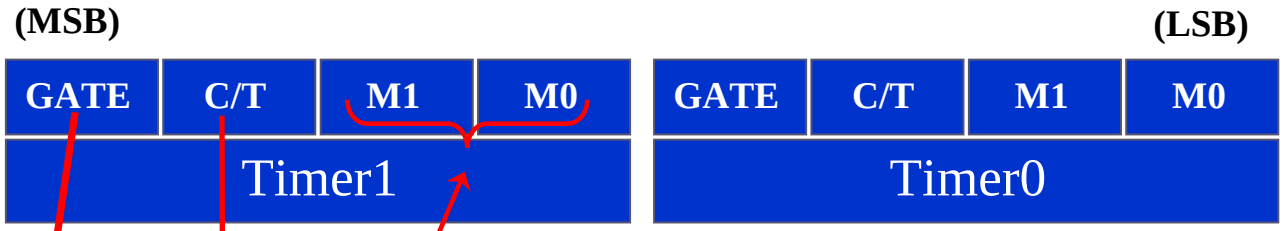
PROGRAMMING TIMERS

TMOD Register

- Both timers 0 and 1 use the same register, called TMOD (timer mode), to set the various timer operation modes
- TMOD is a 8-bit register
  - The lower 4 bits are for Timer 0
  - The upper 4 bits are for Timer 1
  - In each case,
    - The lower 2 bits are used to set the timer mode
    - The upper 2 bits to specify the operation

PROGRAMMING TIMERS

TMOD Register (cont')



M1	M0	Mode	Operating Mode
0	0	0	**13-bit timer mode** 8-bit timer/counter THx with TLx as 5-bit prescaler
0	1	1	**16-bit timer mode** 16-bit timer/counter THx and TLx are cascaded; there is no prescaler
1	0	2	**8-bit auto reload** 8-bit auto reload timer/counter; THx holds a value which is to be reloaded TLx each time it overfolws
1	1	3	**Split timer mode**

Gating control when set. Timer/counter is enable only while the INTx pin is high and the TRx control pin is set

When cleared, the timer is enabled whenever the TRx control bit is set

Timer or counter selected

Cleared for timer operation (input from internal system clock)

Set for counter operation (input from Tx input pin)

PROGRAMMING TIMERS

TMOD Register (cont')

If C/T = 0, it is used as a timer for time delay generation. The clock source for the time delay is the crystal frequency of the 8051

Example 9-1

Indicate which mode and which timer are selected for each of the following.
(a) MOV TMOD, #01H (b) MOV TMOD, #20H (c) MOV TMOD, #12H

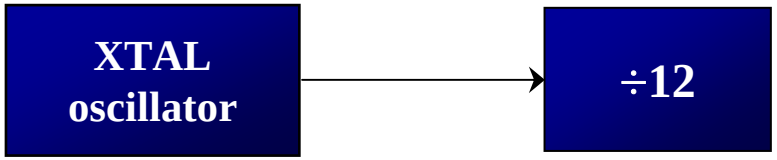
Solution:

We convert the value from hex to binary. From Figure 9-3 we have:
(a) TMOD = 00000001, mode 1 of timer 0 is selected.
(b) TMOD = 00100000, mode 2 of timer 1 is selected.
(c) TMOD = 00010010, mode 2 of timer 0, and mode 1 of timer 1 are selected.

Example 9-2

Find the timer's clock frequency and its period for various 8051-based system, with the crystal frequency 11.0592 MHz when C/T bit of TMOD is 0.

Solution:



1/12 × 11.0529 MHz = 921.6 MHz;
T = 1/921.6 kHz = 1.085 us

PROGRAMMING TIMERS

TMOD Register

GATE

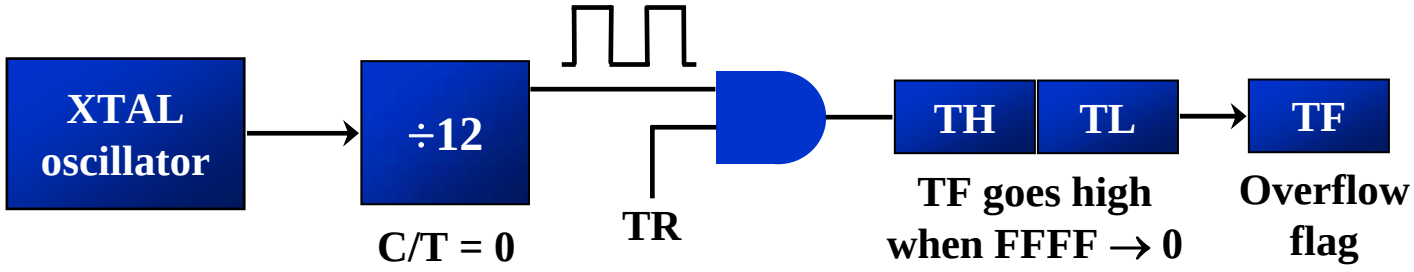
- Timers of 8051 do starting and stopping by either software or hardware control
  - In using software to start and stop the timer where GATE=0
    - The start and stop of the timer are controlled by way of software by the TR (timer start) bits TR0 and TR1
      - The SETB instruction starts it, and it is stopped by the CLR instruction
      - These instructions start and stop the timers as long as GATE=0 in the TMOD register
  - The hardware way of starting and stopping the timer by an external source is achieved by making GATE=1 in the TMOD register

- Timer 0, mode 2
- C/T = 0 to use XTAL clock source
- gate = 0 to use internal (software) start and stop method.

Find the value for TMOD if we want to program timer 0 in mode 2, use 8051 XTAL for the clock source, and use instructions to start and stop the timer.

TMOD = 0000 0010

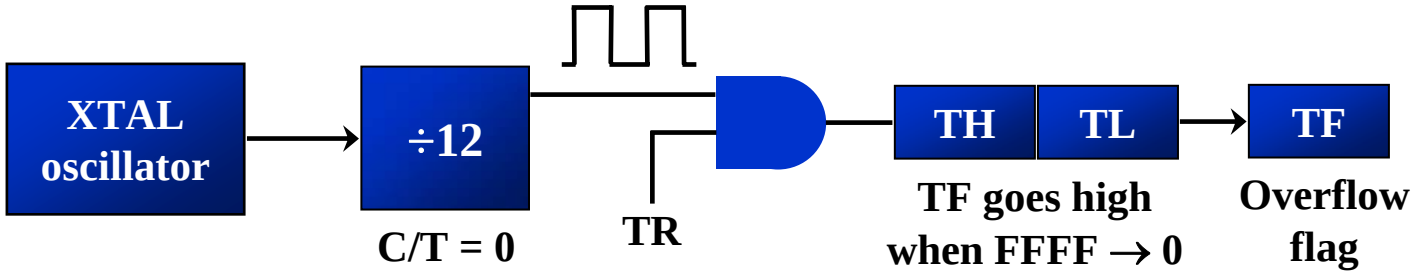
- The following are the characteristics and operations of mode1:
  1. It is a 16-bit timer; therefore, it allows value of 0000 to FFFFH to be loaded into the timer's register TL and TH
  2. After TH and TL are loaded with a 16-bit initial value, the timer must be started
     - This is done by `SETB TR0` for timer 0 and `SETB TR1` for timer 1
  3. After the timer is started, it starts to count up
     - It counts up until it reaches its limit of FFFFH



PROGRAMMING TIMERS

Mode 1 Programming (cont')

3. (cont')
   - When it rolls over from FFFFH to 0000, it sets high a flag bit called TF (timer flag)
     - Each timer has its own timer flag: TF0 for timer 0, and TF1 for timer 1
     - This timer flag can be monitored
   - When this timer flag is raised, one option would be to stop the timer with the instructions `CLR TR0` or `CLR TR1`, for timer 0 and timer 1, respectively
4. After the timer reaches its limit and rolls over, in order to repeat the process
   - TH and TL must be reloaded with the original value, and
   - TF must be reloaded to 0



- To generate a time delay
  1. Load the TMOD value register indicating which timer (timer 0 or timer 1) is to be used and which timer mode (0 or 1) is selected
  2. Load registers TL and TH with initial count value
  3. Start the timer
  4. Keep monitoring the timer flag (TF) with the `JNB TFx,target` instruction to see if it is raised
     - Get out of the loop when TF becomes high
  5. Stop the timer
  6. Clear the TF flag for the next round
  7. Go back to Step 2 to load TH and TL again

PROGRAMMING TIMERS

Mode 1 Programming

Steps to Mode 1 Program (cont')

Example 9-4

In the following program, we create a square wave of 50% duty cycle (with equal portions high and low) on the P1.5 bit. Timer 0 is used to generate the time delay. Analyze the program

```
        MOV  TMOD,#01   ;Timer 0, mode 1(16-bit mode)
HERE:   MOV  TL0,#0F2H  ;TL0=F2H, the low byte
        MOV  TH0,#0FFH  ;TH0=FFH, the high byte
        CPL  P1.5       ;toggle P1.5
        ACALL DELAY
        SJMP HERE
```

In the above program notice the following step.

1. TMOD is loaded.
2. FFF2H is loaded into TH0-TL0.
3. P1.5 is toggled for the high and low portions of the pulse.

...

Example 9-4 (cont')

```
DELAY:
        SETB TR0          ;start the timer 0
AGAIN:  JNB  TF0,AGAIN    ;monitor timer flag 0
                          ;until it rolls over
        CLR  TR0          ;stop timer 0
        CLR  TF0          ;clear timer 0 flag
        RET
```

4. The DELAY subroutine using the timer is called.
5. In the DELAY subroutine, timer 0 is started by the `SETB TR0` instruction.
6. Timer 0 counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of FFF3, FFF4, FFF5, FFF6, FFF7, FFF8, FFF9, FFFA, FFFB, and so on until it reaches FFFFH. One more clock rolls it to 0, raising the timer flag (TF0=1). At that point, the JNB instruction falls through.

7. Timer 0 is stopped by the instruction `CLR TR0`. The DELAY subroutine ends, and the process is repeated.

Notice that to repeat the process, we must reload the TL and TH registers, and start the process is repeated …

PROGRAMMING TIMERS

Mode 1 Programming

Steps to Mode 1 Program (cont')

Example 9-5

In Example 9-4, calculate the amount of time delay in the DELAY subroutine generated by the timer. Assume XTAL = 11.0592 MHz.

Solution:

The timer works with a clock frequency of 1/12 of the XTAL frequency; therefore, we have 11.0592 MHz / 12 = 921.6 kHz as the timer frequency. As a result, each clock has a period of T = 1/921.6kHz = 1.085us. In other words, Timer 0 counts up each 1.085 us resulting in delay = number of counts × 1.085us.

The number of counts for the roll over is FFFFH – FFF2H = 0DH (13 decimal). However, we add one to 13 because of the extra clock needed when it rolls over from FFFF to 0 and raise the TF flag. This gives 14 × 1.085us = 15.19us for half the pulse. For the entire period it is T = 2 × 15.19us = 30.38us as the time delay generated by the timer.

(a) in hex
(FFFF – YYXX + 1) × 1.085 us, where YYXX are TH, TL initial values respectively. Notice that value YYXX are in hex.

(b) in decimal
Convert YYXX values of the TH, TL register to decimal to get a NNNNN decimal, then (65536 - NNNN) × 1.085 us

Example 9-6

In Example 9-5, calculate the frequency of the square wave generated on pin P1.5.

Solution:

In the timer delay calculation of Example 9-5, we did not include the overhead due to instruction in the loop. To get a more accurate timing, we need to add clock cycles due to this instructions in the loop. To do that, we use the machine cycle from Table A-1 in Appendix A, as shown below.

```
                                        Cycles
HERE:  MOV  TL0,#0F2H                     2
       MOV  TH0,#0FFH                     2
       CPL  P1.5                          1
       ACALL DELAY                        2
       SJMP  HERE                         2
DELAY:
       SETB TR0                           1
AGAIN: JNB  TF0,AGAIN                    14
       CLR  TR0                           1
       CLR  TF0                           1
       RET                                2
                                 Total    28
```

T = 2 × 28 × 1.085 us = 60.76 us and F = 16458.2 Hz

Example 9-7

Find the delay generated by timer 0 in the following code, using both of the Methods of Figure 9-4. Do not include the overhead due to instruction.

```
        CLR  P2.3       ;Clear P2.3
        MOV  TMOD,#01   ;Timer 0, 16-bitmode
HERE:   MOV  TL0,#3EH   ;TL0=3Eh, the low byte
        MOV  TH0,#0B8H ;TH0=B8H, the high byte
        SETB P2.3       ;SET high timer 0
        SETB TR0        ;Start the timer 0
AGAIN: JNB  TF0,AGAIN ;Monitor timer flag 0
        CLR  TR0        ;Stop the timer 0
        CLR  TF0        ;Clear TF0 for next round
        CLR  P2.3
```

Solution:

(a) (FFFFH – B83E + 1) = 47C2H = 18370 in decimal and 18370 × 1.085 us = 19.93145 ms

(b) Since TH – TL = B83EH = 47166 (in decimal) we have 65536 – 47166 = 18370. This means that the timer counts from B38EH to FFFF. This plus Rolling over to 0 goes through a total of 18370 clock cycles, where each clock is 1.085 us in duration. Therefore, we have 18370 × 1.085 us = 19.93145 ms as the width of the pulse.

PROGRAMMING TIMERS

Mode 1 Programming

Steps to Mode 1 Program (cont')

Example 9-8

Modify TL and TH in Example 9-7 to get the largest time delay possible. Find the delay in ms. In your calculation, exclude the overhead due to the instructions in the loop.

Solution:

To get the largest delay we make TL and TH both 0. This will count up from 0000 to FFFFH and then roll over to zero.

```
        CLR  P2.3     ;Clear P2.3
        MOV  TMOD,#01 ;Timer 0, 16-bitmode
HERE:   MOV  TL0,#0   ;TL0=0, the low byte
        MOV  TH0,#0   ;TH0=0, the high byte
        SETB P2.3     ;SET high P2.3
        SETB TR0      ;Start timer 0
AGAIN:  JNB  TF0,AGAIN ;Monitor timer flag 0
        CLR  TR0      ;Stop the timer 0
        CLR  TF0      ;Clear timer 0 flag
        CLR  P2.3
```

Making TH and TL both zero means that the timer will count from 0000 to FFFF, and then roll over to raise the TF flag. As a result, it goes through a total Of 65536 states. Therefore, we have delay = (65536 - 0) × 1.085 us = 71.1065ms.

Example 9-9

The following program generates a square wave on P1.5 continuously using timer 1 for a time delay. Find the frequency of the square wave if XTAL = 11.0592 MHz. In your calculation do not include the overhead due to Instructions in the loop.

```
        MOV  TMOD,#10;Timer 1, mod 1 (16-bitmode)
AGAIN:  MOV  TL1,#34H ;TL1=34H, low byte of timer
        MOV  TH1,#76H ;TH1=76H, high byte timer
        SETB TR1      ;start the timer 1
BACK:   JNB  TF1,BACK ;till timer rolls over
        CLR  TR1      ;stop the timer 1
        CPL  P1.5     ;comp. p1. to get hi, lo
        CLR  TF1      ;clear timer flag 1
        SJMP AGAIN    ;is not auto-reload
```

Solution:

Since FFFFH – 7634H = 89CBH + 1 = 89CCH and 89CCH = 35276 clock count and 35276 × 1.085 us = 38.274 ms for half of the square wave. The frequency = 13.064Hz.

Also notice that the high portion and low portion of the square wave pulse are equal. In the above calculation, the overhead due to all the instruction in the loop is not included.

PROGRAMMING TIMERS

Mode 1 Programming

Finding the Loaded Timer Values

- To calculate the values to be loaded into the TL and TH registers, look at the following example
  - Assume XTAL = 11.0592 MHz, we can use the following steps for finding the TH, TL registers' values
    1. Divide the desired time delay by 1.085 us
    2. Perform 65536 – n, where n is the decimal value we got in Step1
    3. Convert the result of Step2 to hex, where yyxx is the initial hex value to be loaded into the timer's register
    4. Set TL = xx and TH = yy

PROGRAMMING TIMERS

Mode 1 Programming

Finding the Loaded Timer Values (cont')

Example 9-10

Assume that XTAL = 11.0592 MHz. What value do we need to load the timer's register if we want to have a time delay of 5 ms (milliseconds)? Show the program for timer 0 to create a pulse width of 5 ms on P2.3.

Solution:

Since XTAL = 11.0592 MHz, the counter counts up every 1.085 us. This means that out of many 1.085 us intervals we must make a 5 ms pulse. To get that, we divide one by the other. We need 5 ms / 1.085 us = 4608 clocks. To Achieve that we need to load into TL and TH the value 65536 – 4608 = EE00H. Therefore, we have TH = EE and TL = 00.

```
        CLR  P2.3     ;Clear P2.3
        MOV  TMOD,#01 ;Timer 0, 16-bitmode
HERE:   MOV  TL0,#0   ;TL0=0, the low byte
        MOV  TH0,#0EEH ;TH0=EE, the high byte
        SETB P2.3     ;SET high P2.3
        SETB TR0      ;Start timer 0
AGAIN:  JNB  TF0,AGAIN ;Monitor timer flag 0
        CLR  TR0      ;Stop the timer 0
        CLR  TF0      ;Clear timer 0 flag
```

PROGRAMMING TIMERS

Mode 1 Programming

Finding the Loaded Timer Values (cont')

Example 9-11

Assume that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 kHz frequency on pin P1.5.

Solution:

This is similar to Example 9-10, except that we must toggle the bit to generate the square wave. Look at the following steps.

(a) T = 1 / f = 1 / 2 kHz = 500 us the period of square wave.

(b) 1 / 2 of it for the high and low portion of the pulse is 250 us.

(c) 250 us / 1.085 us = 230 and 65536 – 230 = 65306 which in hex is FF1AH.

(d) TL = 1A and TH = FF, all in hex. The program is as follow.

```
        MOV  TMOD,#01 ;Timer 0, 16-bitmode
AGAIN:  MOV  TL1,#1AH ;TL1=1A, low byte of timer
        MOV  TH1,#0FFH ;TH1=FF, the high byte
        SETB TR1      ;Start timer 1
BACK:   JNB  TF1,BACK ;until timer rolls over
        CLR  TR1      ;Stop the timer 1
        CLR  P1.5     ;Clear timer flag 1
        CLR  TF1      ;Clear timer 1 flag
        SJMP AGAIN    ;Reload timer
```

PROGRAMMING TIMERS

Mode 1 Programming

Finding the Loaded Timer Values (cont')

Example 9-12

Assume XTAL = 11.0592 MHz, write a program to generate a square wave of 50 kHz frequency on pin P2.3.

Solution:

Look at the following steps.

(a) T = 1 / 50 = 20 ms, the period of square wave.

(b) 1 / 2 of it for the high and low portion of the pulse is 10 ms.

(c) 10 ms / 1.085 us = 9216 and 65536 – 9216 = 56320 in decimal, and in hex it is DC00H.

(d) TL = 00 and TH = DC (hex).

```
        MOV  TMOD,#10H  ;Timer 1, mod 1
AGAIN:  MOV  TL1,#00    ;TL1=00,low byte of timer
        MOV  TH1,#0DCH ;TH1=DC, the high byte
        SETB TR1        ;Start timer 1
BACK:   JNB  TF1,BACK   ;until timer rolls over
        CLR  TR1        ;Stop the timer 1
        CLR  P2.3       ;Comp. p2.3 to get hi, lo
        SJMP AGAIN      ;Reload timer
                        ;mode 1 isn't auto-reload
```

PROGRAMMING TIMERS

Mode 1 Programming

Generating Large Time Delay

Example 9-13

Examine the following program and find the time delay in seconds. Exclude the overhead due to the instructions in the loop.

```
        MOV  TMOD,#10H  ;Timer 1, mod 1
        MOV  R3,#200    ;cnter for multiple delay
AGAIN:  MOV  TL1,#08H   ;TL1=08,low byte of timer
        MOV  TH1,#01H   ;TH1=01,high byte
        SETB TR1        ;Start timer 1
BACK:   JNB  TF1,BACK   ;until timer rolls over
        CLR  TR1        ;Stop the timer 1
        CLR  TF1        ;clear Timer 1 flag
        DJNZ R3,AGAIN   ;if R3 not zero then
                        ;reload timer
```

Solution:

TH-TL = 0108H = 264 in decimal and 65536 – 264 = 65272. Now 65272 × 1.085 μs = 70.820 ms, and for 200 of them we have 200 ×70.820 ms = 14.164024 seconds.

PROGRAMMING TIMERS

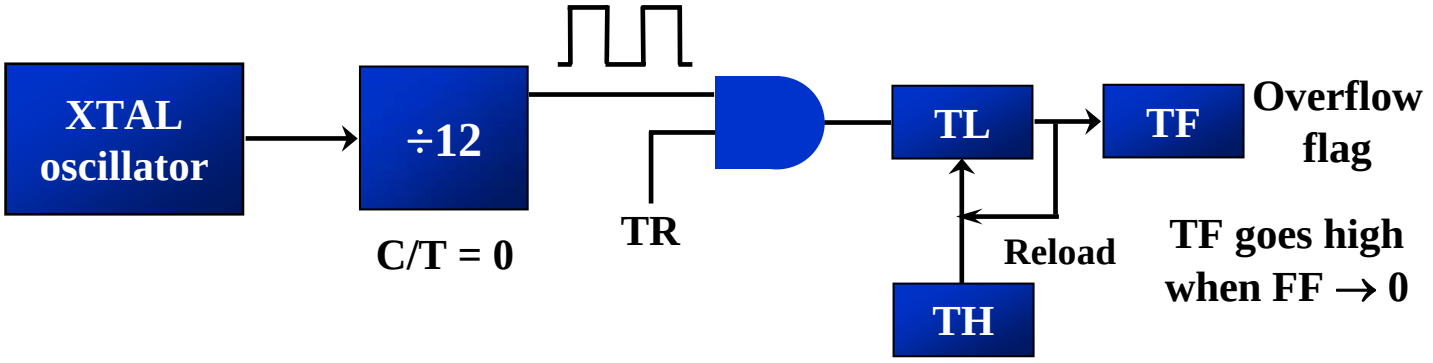
Mode 2 Programming

- The following are the characteristics and operations of mode 2:
  1. It is an 8-bit timer; therefore, it allows only values of 00 to FFH to be loaded into the timer's register TH
  2. After TH is loaded with the 8-bit value, the 8051 gives a copy of it to TL
     - Then the timer must be started
     - This is done by the instruction `SETB TR0` for timer 0 and `SETB TR1` for timer 1
  3. After the timer is started, it starts to count up by incrementing the TL register
     - It counts up until it reaches its limit of FFH
     - When it rolls over from FFH to 00, it sets high the TF (timer flag)

PROGRAMMING TIMERS

Mode 2 Programming (cont')

4. When the TL register rolls from FFH to 0 and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register
   - To repeat the process, we must simply clear TF and let it go without any need by the programmer to reload the original value
   - This makes mode 2 an auto-reload, in contrast with mode 1 in which the programmer has to reload TH and TL



PROGRAMMING TIMERS

Mode 2 Programming

Steps to Mode 2 Program

- To generate a time delay
  1. Load the TMOD value register indicating which timer (timer 0 or timer 1) is to be used, and the timer mode (mode 2) is selected
  2. Load the TH registers with the initial count value
  3. Start timer
  4. Keep monitoring the timer flag (TF) with the `JNB TFx,target` instruction to see whether it is raised
     - Get out of the loop when TF goes high
  5. Clear the TF flag
  6. Go back to Step4, since mode 2 is auto-reload

Example 9-14

Assume XTAL = 11.0592 MHz, find the frequency of the square wave generated on pin P1.0 in the following program

```
        MOV   TMOD,#20H ;T1/8-bit/auto reload
        MOV   TH1,#5    ;TH1 = 5
        SETB  TR1       ;start the timer 1
BACK:   JNB   TF1,BACK  ;till timer rolls over
        CPL   P1.0      ;P1.0 to hi, lo
        CLR   TF1       ;clear Timer 1 flag
        SJMP  BACK      ;mode 2 is auto-reload
```

Solution:

First notice the target address of SJMP. In mode 2 we do not need to reload TH since it is auto-reload. Now (256 - 05) × 1.085 us = 251 × 1.085 us = 272.33 us is the high portion of the pulse. Since it is a 50% duty cycle square wave, the period T is twice that; as a result T = 2 × 272.33 us = 544.67 us and the frequency = 1.83597 kHz

Example 9-15

Find the frequency of a square wave generated on pin P1.0.

Solution:

```
        MOV   TMOD,#2H   ;Timer 0, mod 2
                         ;(8-bit, auto reload)
        MOV   TH0,#0
AGAIN:  MOV   R5,#250     ;multiple delay count
        ACALL DELAY
        CPL   P1.0
        SJMP  AGAIN

DELAY:  SETB  TR0        ;start the timer 0
BACK:   JNB   TF0,BACK   ;stay timer rolls over
        CLR   TR0        ;stop timer
        CLR   TF0        ;clear TF for next round
        DJNZ  R5,DELAY
        RET
```

T = 2 (250 × 256 × 1.085 us) = 138.88ms, and frequency = 72 Hz

PROGRAMMING TIMERS

Mode 2 Programming

Steps to Mode 2 Program (cont')

Example 9-16

Assuming that we are programming the timers for mode 2, find the value (in hex) loaded into TH for each of the following cases.

(a) `MOV TH1,#-200`
(b) `MOV TH0,#-60`
(c) `MOV TH1,#-3`
(d) `MOV TH1,#-12`
(e) `MOV TH0,#-48`

Solution:

You can use the Windows scientific calculator to verify the result provided by the assembler. In Windows calculator, select decimal and enter 200. Then select hex, then +/- to get the TH value. Remember that we only use the right two digits and ignore the rest since our data is an 8-bit data.

Decimal	2's complement (TH value)
-3	FDH
-12	F4H
-48	D0H
-60	C4H
-200	38H

The number 200 is the timer count till the TF is set to 1

The advantage of using negative values is that you don't need to calculate the value loaded to THx

COUNTER PROGRAMMING

- Timers can also be used as counters counting events happening outside the 8051
  - When it is used as a counter, it is a pulse outside of the 8051 that increments the TH, TL registers
  - TMOD and TH, TL registers are the same as for the timer discussed previously
- Programming the timer in the last section also applies to programming it as a counter
  - Except the source of the frequency

COUNTER PROGRAMMING

C/T Bit in TMOD Register

- The C/T bit in the TMOD registers decides the source of the clock for the timer
  - When C/T = 1, the timer is used as a counter and gets its pulses from outside the 8051
    - The counter counts up as pulses are fed from pins 14 and 15, these pins are called T0 (timer 0 input) and T1 (timer 1 input)

Port 3 pins used for Timers 0 and 1

Pin	Port Pin	Function	Description
14	P3.4	T0	Timer/counter 0 external input
15	P3.5	T1	Timer/counter 1 external input

COUNTER PROGRAMMING

C/T Bit in TMOD Register (cont')

Example 9-18

Assuming that clock pulses are fed into pin T1, write a program for counter 1 in mode 2 to count the pulses and display the state of the TL1 count on P2, which connects to 8 LEDs.

Solution:

```
        MOV   TM0D,#01100000B ;counter 1, mode 2,
                              ;C/T=1 external pulses
        MOV   TH1,#0  ;clear TH1
        SETB  P3.5    ;make T1 input
AGAIN:  SETB  TR1     ;start the counter
BACK:   MOV   A,TL1   ;get copy of TL
        MOV   P2,A    ;display it on port 2
        JNB   TF1,Back ;keep doing, if TF = 0
        CLR   TR1     ;stop the counter 1
        CLR   TF1     ;make TF=0
        SJMP  AGAIN   ;keep doing it
```

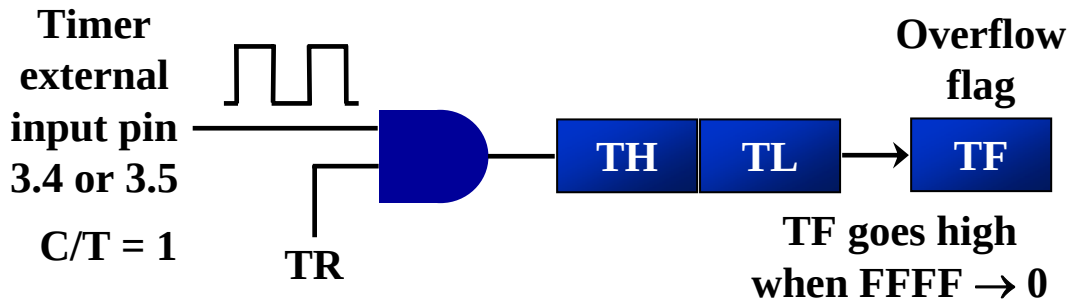
Notice in the above program the role of the instruction `SETB P3.5`.

Since ports are set up for output when the 8051 is powered up, we make P3.5 an input port by making it high. In other words, we must configure (set high) the T1 pin (pin P3.5) to allow pulses to be fed into it.

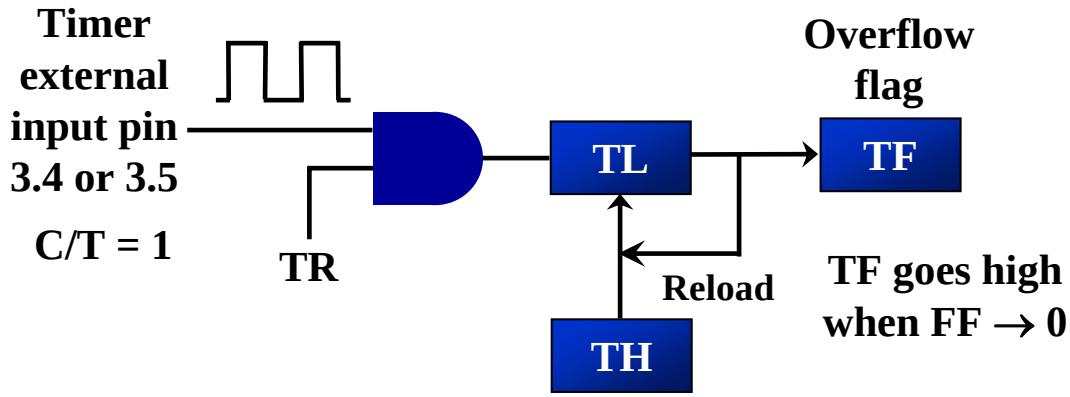
COUNTER PROGRAMMING

C/T Bit in TMOD Register (cont')

Timer with external input (Mode 1)



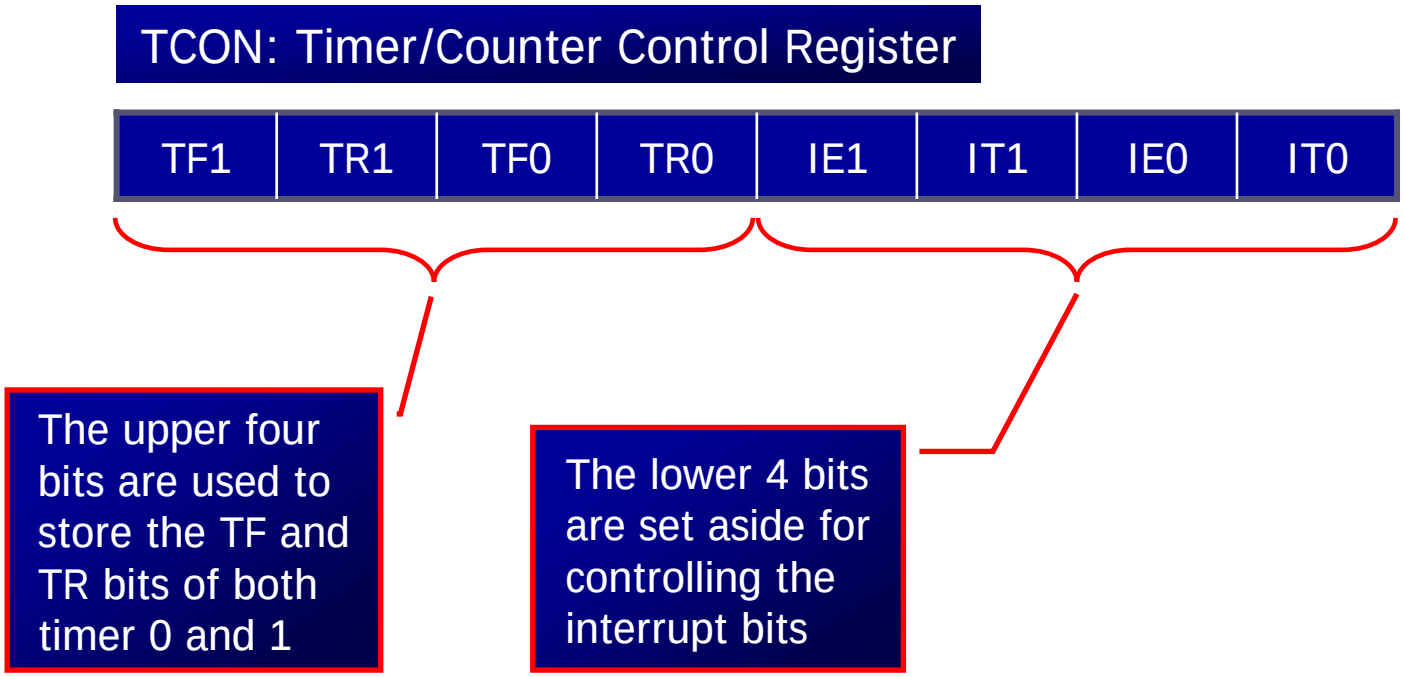
Timer with external input (Mode 2)



COUNTER PROGRAMMING

TCON Register

- TCON (timer control) register is an 8-bit register



- TCON register is a bit-addressable register

Equivalent instruction for the Timer Control Register

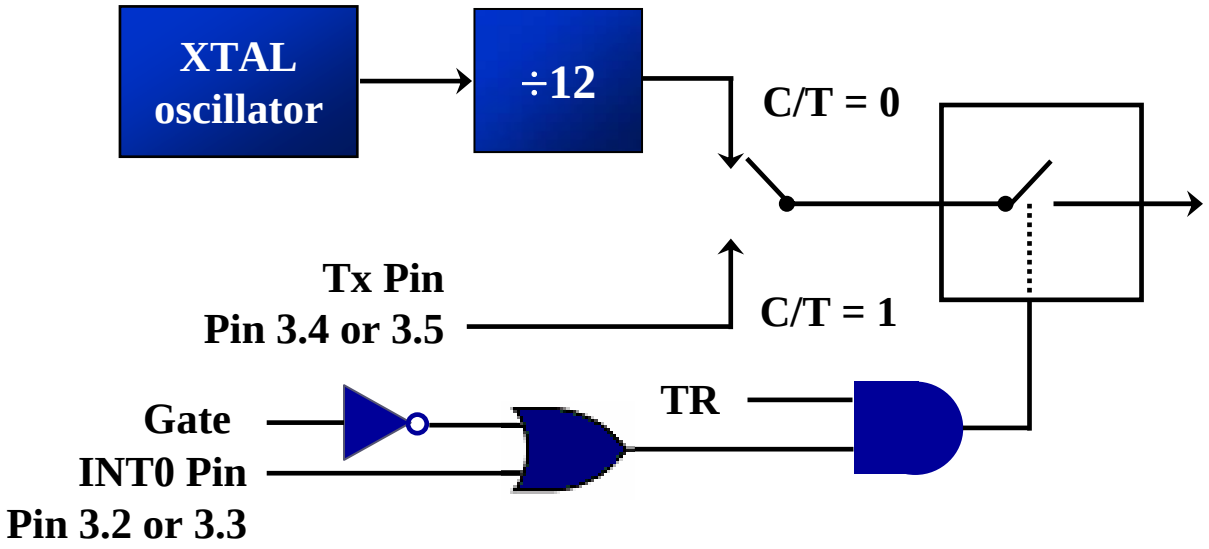
For timer 0			
SETB	TR0	=	SETB TCON.4
CLR	TR0	=	CLR TCON.4
SETB	TF0	=	SETB TCON.5
CLR	TF0	=	CLR TCON.5
For timer 1			
SETB	TR1	=	SETB TCON.6
CLR	TR1	=	CLR TCON.6
SETB	TF1	=	SETB TCON.7
CLR	TF1	=	CLR TCON.7

COUNTER PROGRAMMING

TCON Register

Case of GATE = 1

- If GATE = 1, the start and stop of the timer are done externally through pins P3.2 and P3.3 for timers 0 and 1, respectively
  - This hardware way allows to start or stop the timer externally at any time via a simple switch



Example 9-20

Write an 8051 C program to toggle all the bits of port P1 continuously with some delay in between. Use Timer 0, 16-bit mode to generate the delay.

Solution:

```
#include <reg51.h>
void T0Delay(void);
void main(void){
  while (1) {
    P1=0x55;
    T0Delay();
    P1=0xAA;
    T0Delay();
  }
}
void T0Delay(){
  TMOD=0x01;
  TL0=0x00;
  TH0=0x35;
  TR0=1;
  while (TF0==0);
  TR0=0;
  TF0=0;
}
```

FFFFH – 3500H = CAFFH

= 51967 + 1 = 51968

51968 × 1.085 μs = 56.384 ms is the approximate delay

PROGRAMMING TIMERS IN C

Calculating Delay Length Using Timers

- To speed up the 8051, many recent versions of the 8051 have reduced the number of clocks per machine cycle from 12 to four, or even one
- The frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051, regardless of the 8051 version

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 1 (16-bit Non Auto-reload)

Example 9-21

Write an 8051 C program to toggle only bit P1.5 continuously every 50 ms. Use Timer 0, mode 1 (16-bit) to create the delay. Test the program on the (a) AT89C51 and (b) DS89C420.

Solution:

```
#include <reg51.h>
void T0M1Delay(void);
sbit mybit=P1^5;
void main(void){
  while (1) {
    mybit=~mybit;
    T0M1Delay();
  }
}
void T0M1Delay(void){
  TMOD=0x01;
  TL0=0xFD;
  TH0=0x4B;
  TR0=1;
  while (TF0==0);
  TR0=0;
  TF0=0;
}
```

FFFFH – 4BFDH = B402H

= 46082 + 1 = 46083

46083 × 1.085 μs = 50 ms

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 1 (16-bit Non Auto-reload) (cont')

Example 9-22

Write an 8051 C program to toggle all bits of P2 continuously every 500 ms. Use Timer 1, mode 1 to create the delay.

Solution:

```
//tested for DS89C420, XTAL = 11.0592 MHz
#include <reg51.h>
void T1M1Delay(void);
void main(void){
  unsigned char x;
  P2=0x55;
  while (1) {
    P2=~P2;
    for (x=0;x<20;x++)
       T1M1Delay();
  }
}
void T1M1Delay(void){
  TMOD=0x10;
  TL1=0xFE;
  TH1=0xA5;
  TR1=1;
  while (TF1==0);
  TR1=0;
  TF1=0;
}
```

A5FEH = 42494 in decimal

65536 – 42494 = 23042

23042 × 1.085 μs = 25 ms and

20 × 25 ms = 500 ms

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 1 (16-bit Non Auto-reload) (cont')

Example 9-25
A switch is connected to pin P1.2. Write an 8051 C program to monitor SW and create the following frequencies on pin P1.7:
SW=0: 500Hz
SW=1: 750Hz, use Timer 0, mode 1 for both of them.

Solution:

```
#include <reg51.h>
sbit mybit=P1^5;
sbit SW=P1^7;
void T0M1Delay(unsigned char);
void main(void){
  SW=1;
  while (1) {
    mybit=~mybit;
    if (SW==0)
       T0M1Delay(0);
    else
       T0M1Delay(1);
  }
}

.....
```

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 1 (16-bit Non Auto-reload) (cont')

Example 9-25

```
.....

void T0M1Delay(unsigned char c){
  TMOD=0x01;
  if (c==0) {
     TL0=0x67;
     TH0=0xFC;
  }
  else {
     TL0=0x9A;
     TH0=0xFD;
  }
  TR0=1;
  while (TF0==0);
  TR0=0;
  TF0=0;
}
```

FC67H = 64615

65536 – 64615 = 921

921 × 1.085 μs = 999.285 μs

1 / (999.285 μs × 2) = 500 Hz

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 2 (8-bit Auto-reload)

Example 9-23

Write an 8051 C program to toggle only pin P1.5 continuously every 250 ms. Use Timer 0, mode 2 (8-bit auto-reload) to create the delay.

Solution:

```
#include <reg51.h>
void T0M2Delay(void);
sbit mybit=P1^5;
void main(void){
  unsigned char x,y;
  while (1) {
    mybit=~mybit;
    for (x=0;x<250;x++)
       for (y=0;y<36;y++) //we put 36, not 40
          T0M2Delay();
  }
}
void T0M2Delay(void){
  TMOD=0x02;
  TH0=-23;
  TR0=1;
  while (TF0==0);
  TR0=0;
  TF0=0;
}
```

Due to overhead of the for loop in C, we put 36 instead of 40

256 – 23 = 233

23 × 1.085 μs = 25 μs and

25 μs × 250 × 40 = 250 ms

PROGRAMMING TIMERS IN C

Times 0/1 Delay Using Mode 2 (8-bit Auto-reload) (cont')

Example 9-24

Write an 8051 C program to create a frequency of 2500 Hz on pin P2.7. Use Timer 1, mode 2 to create delay.

Solution:

```
#include <reg51.h>
void T1M2Delay(void);
sbit mybit=P2^7;
void main(void){
  unsigned char x;
  while (1) {
    mybit=~mybit;
    T1M2Delay();
  }
}
void T1M2Delay(void){
  TMOD=0x20;
  TH1=-184;
  TR1=1;
  while (TF1==0);
  TR1=0;
  TF1=0;
}
```

1/2500 Hz = 400 μs

400 μs /2 = 200 μs

200 μs / 1.085 μs = 184

Example 9-26
Assume that a 1-Hz external clock is being fed into pin T1 (P3.5). Write a C program for counter 1 in mode 2 (8-bit auto reload) to count up and display the state of the TL1 count on P1. Start the count at 0H.

Solution:

```
#include <reg51.h>
sbit T1=P3^5;
void main(void){
  T1=1;
  TMOD=0x60;
  TH1=0;
  while (1) {
    do {
       TR1=1;
       P1=TL1;
    }
    while (TF1==0);
    TR1=0;
    TF1=0;

  }
}
```

PROGRAMMING TIMERS IN C

C Programming of Timers as Counters (cont')

Example 9-27
Assume that a 1-Hz external clock is being fed into pin T0 (P3.4). Write a C program for counter 0 in mode 1 (16-bit) to count the pulses and display the state of the TH0 and TL0 registers on P2 and P1, respectively.

Solution:

```
#include <reg51.h>
void main(void){
  T0=1;
  TMOD=0x05;
  TL0=0
  TH0=0;
  while (1) {
    do {
       TR0=1;
       P1=TL0;
       P2=TH0;
    }
    while (TF0==0);
    TR0=0;
    TF0=0;

  }
}
```

SERIAL COMMUNICATION

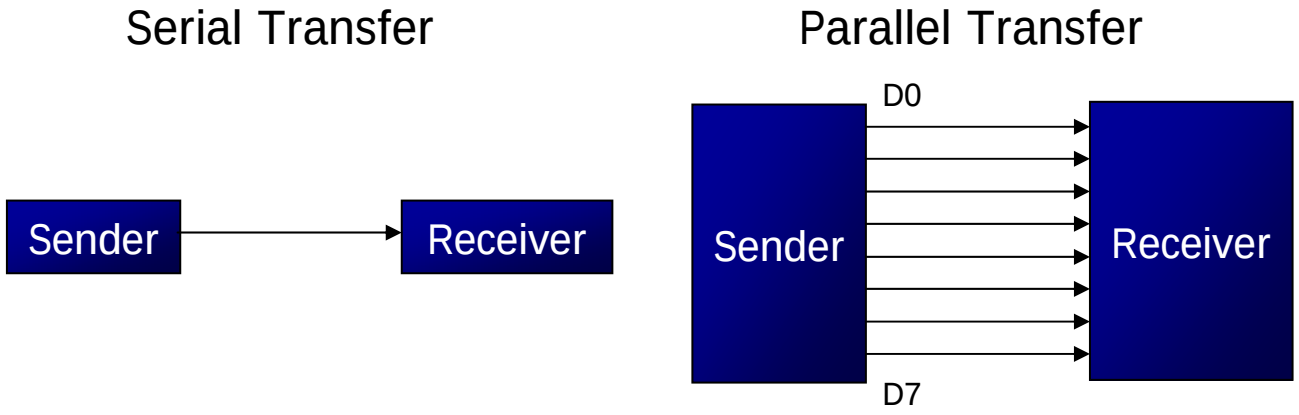
Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University

BASICS OF SERIAL COMMUNICA-TION

- Computers transfer data in two ways:
  - Parallel
    - Often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away
  - Serial
    - To transfer to a device located many meters away, the serial method is used
    - The data is sent one bit at a time



BASICS OF SERIAL COMMUNICA-TION (cont')

- At the transmitting end, the byte of data must be converted to serial bits using parallel-in-serial-out shift register
- At the receiving end, there is a serial-in-parallel-out shift register to receive the serial data and pack them into byte
- When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation
- If data is to be transferred on the telephone line, it must be converted from 0s and 1s to audio tones
  - This conversion is performed by a device called a *modem*, "Modulator/demodulator"

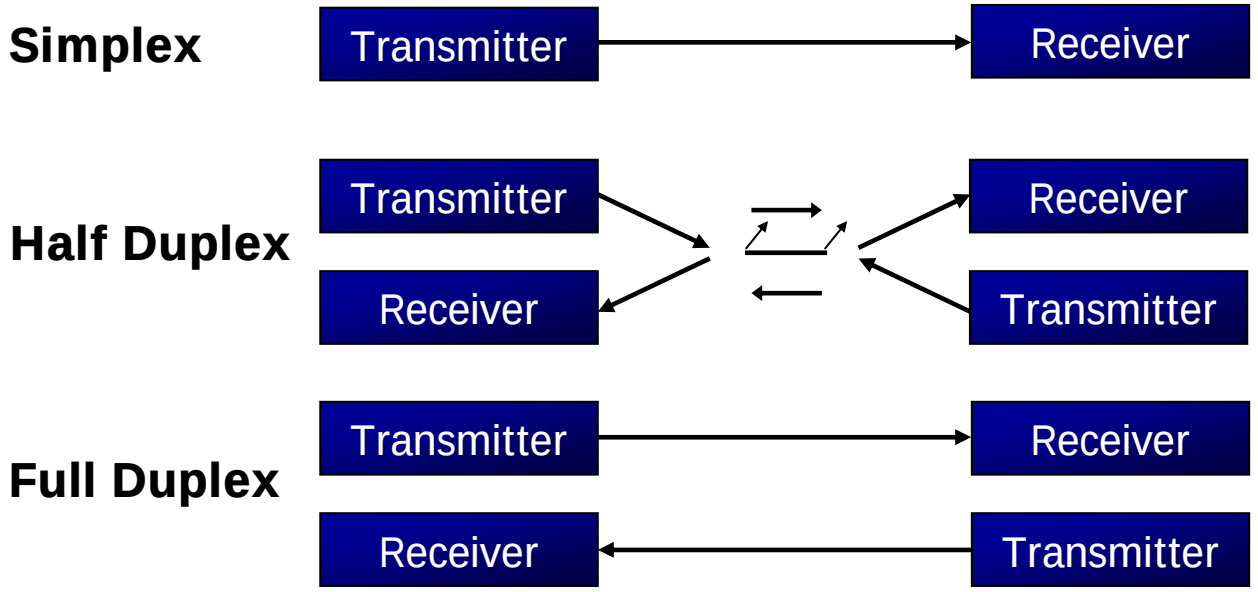
BASICS OF SERIAL COMMUNICA-TION (cont')

- Serial data communication uses two methods
  - *Synchronous* method transfers a block of data at a time
  - *Asynchronous* method transfers a single byte at a time
- It is possible to write software to use either of these methods, but the programs can be tedious and long
  - There are special IC chips made by many manufacturers for serial communications
    - UART (universal asynchronous Receiver-transmitter)
    - USART (universal synchronous-asynchronous Receiver-transmitter)

BASICS OF SERIAL COMMUNICA-TION

Half- and Full-Duplex Transmission

- If data can be transmitted and received, it is a *duplex* transmission
  - If data transmitted one way a time, it is referred to as *half duplex*
  - If data can go both ways at a time, it is *full duplex*
- This is contrast to *simplex* transmission



BASICS OF SERIAL COMMUNICA-TION

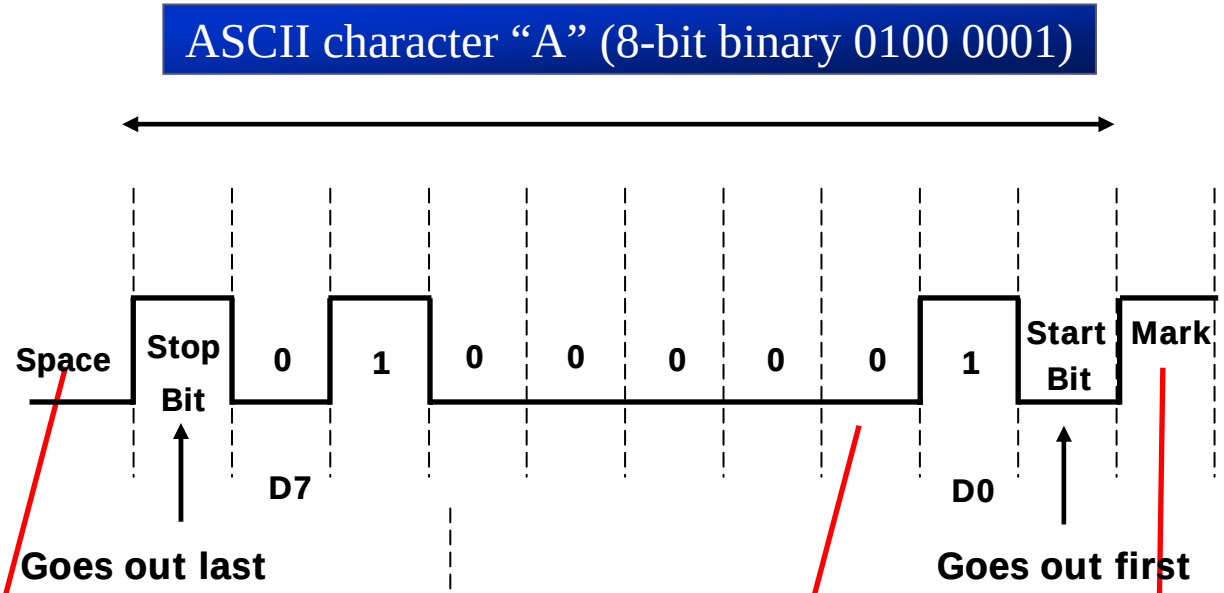
Start and Stop Bits

- A *protocol* is a set of rules agreed by both the sender and receiver on
  - How the data is packed
  - How many bits constitute a character
  - When the data begins and ends
- Asynchronous serial data communication is widely used for character-oriented transmissions
  - Each character is placed in between start and stop bits, this is called *framing*
  - Block-oriented data transfers use the synchronous method
- The start bit is always one bit, but the stop bit can be one or two bits

BASICS OF SERIAL COMMUNICA-TION

Start and Stop Bits (cont')

- The start bit is always a 0 (low) and the stop bit(s) is 1 (high)



The 0 (low) is referred to as *space*

The transmission begins with a start bit followed by D0, the LSB, then the rest of the bits until MSB (D7), and finally, the one stop bit indicating the end of the character

When there is no transfer, the signal is 1 (high), which is referred to as *mark*

BASICS OF SERIAL COMMUNICA-TION

Start and Stop Bits (cont')

- Due to the extended ASCII characters, 8-bit ASCII data is common
  - In older systems, ASCII characters were 7-bit
- In modern PCs the use of one stop bit is standard
  - In older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte

BASICS OF SERIAL COMMUNICA-TION

Start and Stop Bits (cont')

- Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character
  - This gives 25% overhead, i.e. each 8-bit character with an extra 2 bits
- In some systems in order to maintain data integrity, the parity bit of the character byte is included in the data frame
  - UART chips allow programming of the parity bit for odd-, even-, and no-parity options

BASICS OF SERIAL COMMUNICA-TION

Data Transfer Rate

- The rate of data transfer in serial data communication is stated in *bps* (bits per second)
- Another widely used terminology for bps is *baud rate*
  - It is modem terminology and is defined as the number of signal changes per second
  - In modems, there are occasions when a single change of signal transfers several bits of data
- As far as the conductor wire is concerned, the baud rate and bps are the same, and we use the terms interchangeably

BASICS OF SERIAL COMMUNICA-TION

Data Transfer Rate (cont')

- The data transfer rate of given computer system depends on communication ports incorporated into that system
  - IBM PC/XT could transfer data at the rate of 100 to 9600 bps
  - Pentium-based PCs transfer data at rates as high as 56K bps
  - In asynchronous serial data communication, the baud rate is limited to 100K bps

BASICS OF SERIAL COMMUNICA-TION

RS232 Standards

- An interfacing standard RS232 was set by the Electronics Industries Association (EIA) in 1960
- The standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible
  - In RS232, a 1 is represented by -3 ~ -25 V, while a 0 bit is +3 ~ +25 V, making -3 to +3 undefined

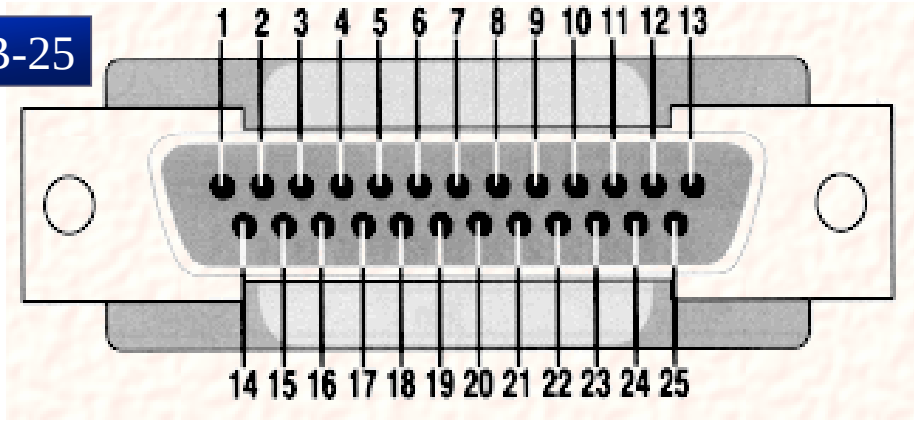
BASICS OF SERIAL COMMUNICA-TION

RS232 Standards (cont')

RS232 DB-25 Pins

Pin	Description	Pin	Description
1	Protective ground	14	Secondary transmitted data
2	Transmitted data (TxD)	15	Transmitted signal element timing
3	Received data (RxD)	16	Secondary receive data
4	Request to send (-RTS)	17	Receive signal element timing
5	Clear to send (-CTS)	18	Unassigned
6	Data set ready (-DSR)	19	Secondary receive data
7	Signal ground (GND)	20	Data terminal ready (-DTR)
8	Data carrier detect (-DCD)	21	Signal quality detector
9/10	Reserved for data testing	22	Ring indicator (RI)
11	Unassigned	23	Data signal rate select
12	Secondary data carrier detect	24	Transmit signal element timing
13	Secondary clear to send	25	Unassigned

RS232 Connector DB-25

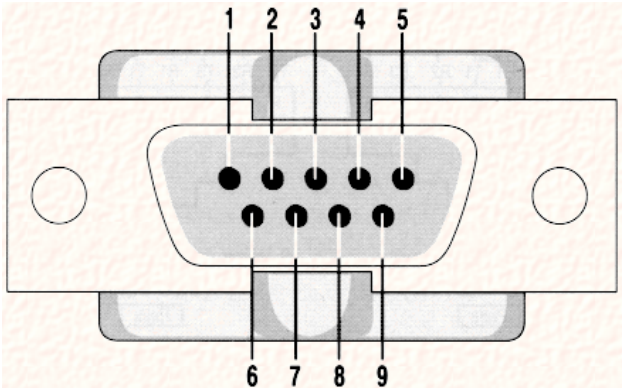


BASICS OF SERIAL COMMUNICA-TION

RS232 Standards (cont')

- Since not all pins are used in PC cables, IBM introduced the DB-9 version of the serial I/O standard

RS232 Connector DB-9



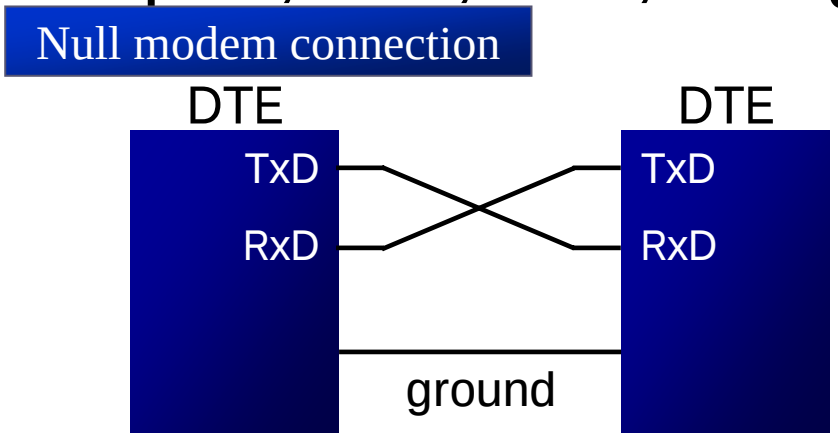
RS232 DB-9 Pins

Pin	Description
1	Data carrier detect (-DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (-DSR)
7	Request to send (-RTS)
8	Clear to send (-CTS)
9	Ring indicator (RI)

BASICS OF SERIAL COMMUNICA-TION

Data Communication Classification

- Current terminology classifies data communication equipment as
  - DTE (data terminal equipment) refers to terminal and computers that send and receive data
  - DCE (data communication equipment) refers to communication equipment, such as modems
- The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground



BASICS OF SERIAL COMMUNICA-TION

RS232 Pins

- DTR (data terminal ready)
  - When terminal is turned on, it sends out signal DTR to indicate that it is ready for communication
- DSR (data set ready)
  - When DCE is turned on and has gone through the self-test, it assert DSR to indicate that it is ready to communicate
- RTS (request to send)
  - When the DTE device has byte to transmit, it assert RTS to signal the modem that it has a byte of data to transmit
- CTS (clear to send)
  - When the modem has room for storing the data it is to receive, it sends out signal CTS to DTE to indicate that it can receive the data now

- DCD (data carrier detect)
  - The modem asserts signal DCD to inform the DTE that a valid carrier has been detected and that contact between it and the other modem is established
- RI (ring indicator)
  - An output from the modem and an input to a PC indicates that the telephone is ringing
  - It goes on and off in synchronous with the ringing sound

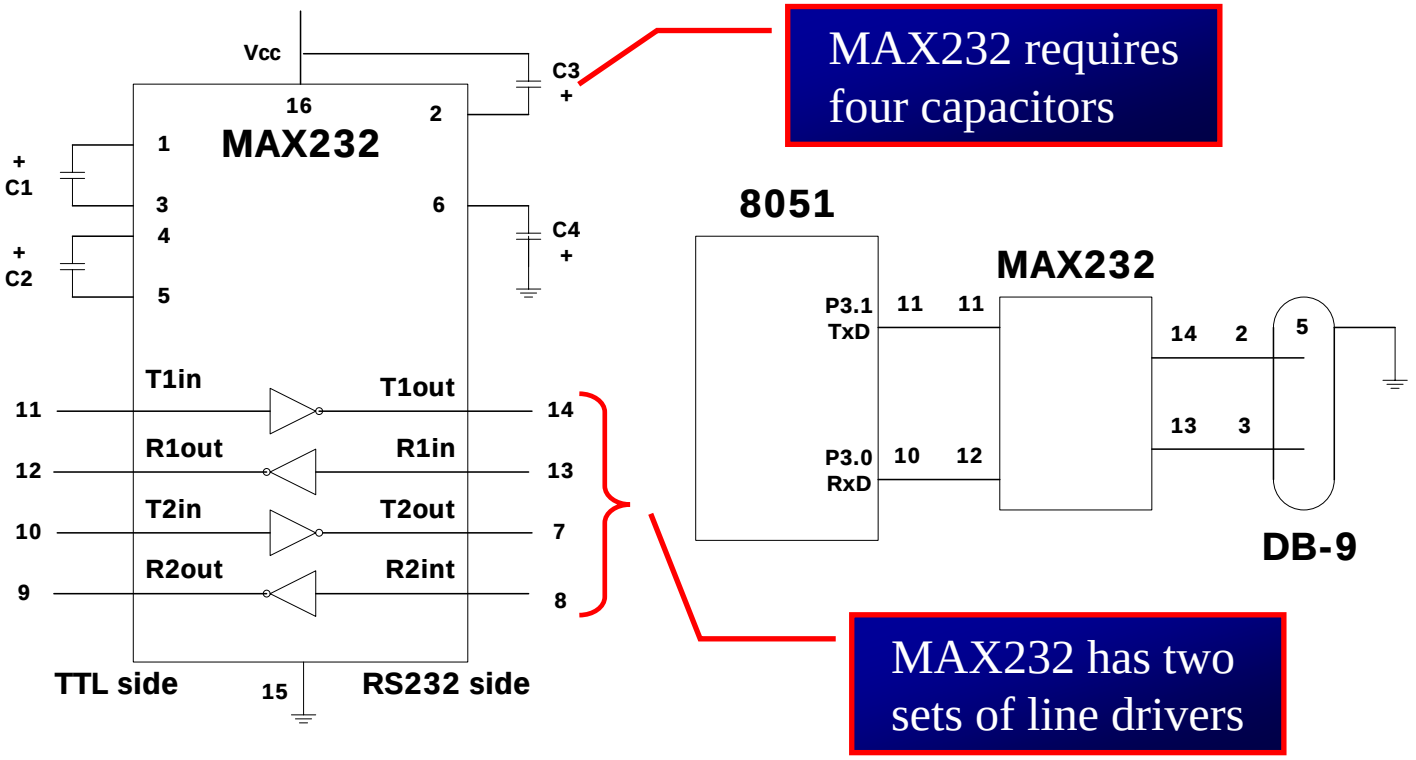
8051 CONNECTION TO RS232

- A line driver such as the MAX232 chip is required to convert RS232 voltage levels to TTL levels, and vice versa
- 8051 has two pins that are used specifically for transferring and receiving data serially
  - These two pins are called TxD and RxD and are part of the port 3 group (P3.0 and P3.1)
  - These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible

8051 CONNECTION TO RS232

MAX232

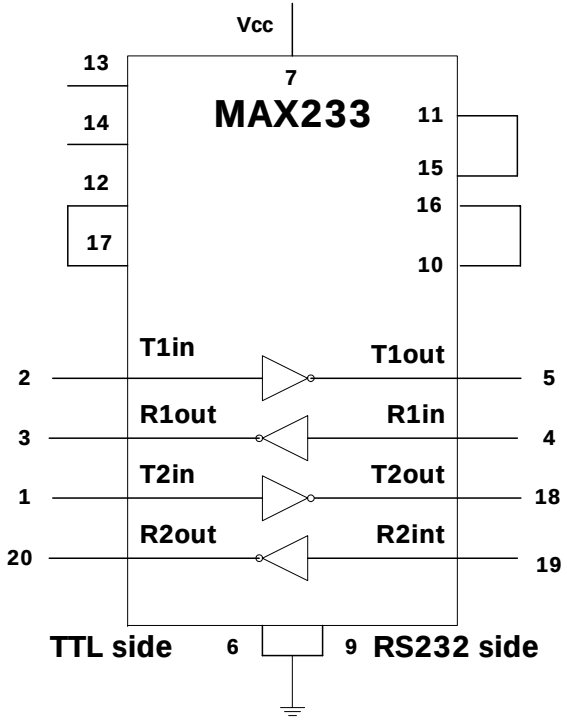
- We need a line driver (voltage converter) to convert the R232's signals to TTL voltage levels that will be acceptable to 8051's TxD and RxD pins

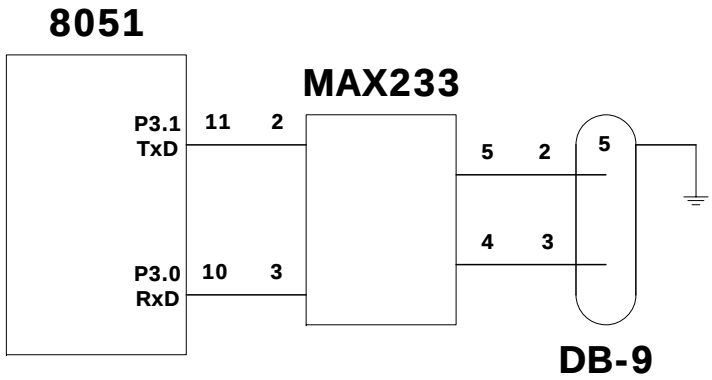


8051 CONNECTION TO RS232

MAX233

- To save board space, some designers use MAX233 chip from Maxim
  - MAX233 performs the same job as MAX232 but eliminates the need for capacitors
  - Notice that MAX233 and MAX232 are not pin compatible





- To allow data transfer between the PC and an 8051 system without any error, we must make sure that the baud rate of 8051 system matches the baud rate of the PC's COM port
- Hyperterminal function supports baud rates much higher than listed below

PC Baud Rates
110
150
300
600
1200
2400
4800
9600
19200

Baud rates supported by 486/Pentium IBM PC BIOS

SERIAL COMMUNICA-TION PROGRAMMING (cont')

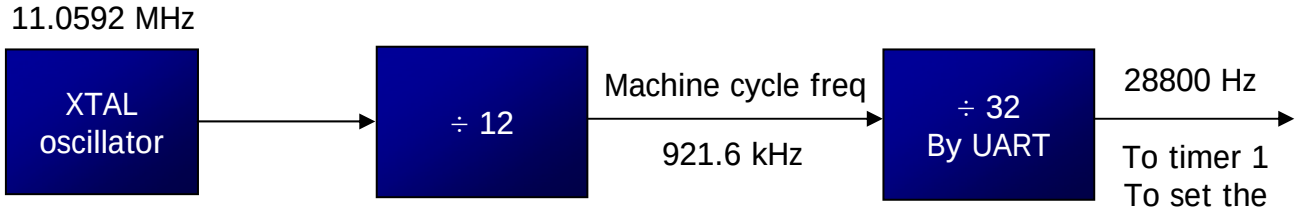
With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 2400 (c) 1200

Solution:

The machine cycle frequency of 8051 = 11.0592 / 12 = 921.6 kHz, and 921.6 kHz / 32 = 28,800 Hz is frequency by UART to timer 1 to set baud rate.

(a) 28,800 / 3 = 9600 where -3 = FD (hex) is loaded into TH1
(b) 28,800 / 12 = 2400 where -12 = F4 (hex) is loaded into TH1
(c) 28,800 / 24 = 1200 where -24 = E8 (hex) is loaded into TH1

Notice that dividing 1/12 of the crystal frequency by 32 is the default value upon activation of the 8051 RESET pin.



Baud Rate	TH1 (Decimal)	TH1 (Hex)
9600	-3	FD
4800	-6	FA
2400	-12	F4
1200	-24	E8

TF is set to 1 every 12 ticks, so it functions as a frequency divider

SERIAL COMMUNICA-TION PROGRAMMING

SBUF Register

- SBUF is an 8-bit register used solely for serial communication
  - For a byte data to be transferred via the TxD line, it must be placed in the SBUF register
    - The moment a byte is written into SBUF, it is framed with the start and stop bits and transferred serially via the TxD line
  - SBUF holds the byte of data when it is received by 8051 RxD line
    - When the bits are received serially via RxD, the 8051 deframes it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in SBUF

```
MOV SBUF,#'D'   ;load SBUF=44h, ASCII for 'D'
MOV SBUF,A      ;copy accumulator into SBUF
MOV A,SBUF      ;copy SBUF into accumulator
```

SERIAL COMMUNICA-TION PROGRAMMING

SCON Register

- SCON is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other things

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode specifier
SM1	SCON.6	Serial port mode specifier
SM2	SCON.5	Used for multiprocessor communication
REN	SCON.4	Set/cleared by software to enable/disable reception
TB8	SCON.3	Not widely used
RB8	SCON.2	Not widely used
TI	SCON.1	Transmit interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW
RI	SCON.0	Receive interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW

Note: *Make SM2, TB8, and RB8 =0*

SERIAL COMMUNICA-TION PROGRAMMING

SCON Register (cont')

- SM0, SM1
  - They determine the framing of data by specifying the number of bits per character, and the start and stop bits

SM0	SM1	
0	0	Serial Mode 0
0	**1**	**Serial Mode 1, 8-bit data, 1 stop bit, 1 start bit**
1	0	Serial Mode 2
1	1	Serial Mode 3

Only mode 1 is of interest to us

- SM2
  - This enables the multiprocessing capability of the 8051

SERIAL COMMUNICA-TION PROGRAMMING

SCON Register (cont')

- REN (receive enable)
  - It is a bit-adressable register
    - When it is high, it allows 8051 to receive data on RxD pin
    - If low, the receiver is disable
- TI (transmit interrupt)
  - When 8051 finishes the transfer of 8-bit character
    - It raises TI flag to indicate that it is ready to transfer another byte
    - TI bit is raised at the beginning of the stop bit
- RI (receive interrupt)
  - When 8051 receives data serially via RxD, it gets rid of the start and stop bits and places the byte in SBUF register
    - It raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost
    - RI is raised halfway through the stop bit

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Transmitting

- In programming the 8051 to transfer character bytes serially
  1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate
  2. The TH1 is loaded with one of the values to set baud rate for serial data transfer
  3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits
  4. TR1 is set to 1 to start timer 1
  5. TI is cleared by `CLR TI` instruction
  6. The character byte to be transferred serially is written into SBUF register
  7. The TI flag bit is monitored with the use of instruction `JNB TI,xx` to see if the character has been transferred completely
  8. To transfer the next byte, go to step 5

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Transmitting (cont')

Write a program for the 8051 to transfer letter "A" serially at 4800 baud, continuously.

Solution:

```
        MOV  TMOD,#20H  ;timer 1,mode 2(auto reload)
        MOV  TH1,#-6    ;4800 baud rate
        MOV  SCON,#50H  ;8-bit, 1 stop, REN enabled
        SETB TR1        ;start timer 1
AGAIN:  MOV  SBUF,#"A"  ;letter "A" to transfer
HERE:   JNB  TI,HERE    ;wait for the last bit
        CLR  TI         ;clear TI for next char
        SJMP AGAIN      ;keep sending A
```

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Transmitting (cont')

Write a program for the 8051 to transfer "YES" serially at 9600 baud, 8-bit data, 1 stop bit, do this continuously

Solution:

```
        MOV  TMOD,#20H  ;timer 1,mode 2(auto reload)
        MOV  TH1,#-3    ;9600 baud rate
        MOV  SCON,#50H  ;8-bit, 1 stop, REN enabled
        SETB TR1        ;start timer 1
AGAIN:  MOV  A,#"Y"     ;transfer "Y"
        ACALL TRANS
        MOV  A,#"E"     ;transfer "E"
        ACALL TRANS
        MOV  A,#"S"     ;transfer "S"
        ACALL TRANS
        SJMP AGAIN      ;keep doing it
;serial data transfer subroutine
TRANS:  MOV  SBUF,A     ;load SBUF
HERE:   JNB  TI,HERE    ;wait for the last bit
        CLR  TI         ;get ready for next byte
         RET
```

SERIAL COMMUNICA-TION PROGRAMMING

Importance of TI Flag

- The steps that 8051 goes through in transmitting a character via TxD
  1. The byte character to be transmitted is written into the SBUF register
  2. The start bit is transferred
  3. The 8-bit character is transferred on bit at a time
  4. The stop bit is transferred
     - It is during the transfer of the stop bit that 8051 raises the TI flag, indicating that the last character was transmitted
  5. By monitoring the TI flag, we make sure that we are not overloading the SBUF
     - If we write another byte into the SBUF before TI is raised, the untransmitted portion of the previous byte will be lost
  6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by `CLR TI` in order for this new byte to be transferred

SERIAL COMMUNICA-TION PROGRAMMING

Importance of TI Flag (cont')

- By checking the TI flag bit, we know whether or not the 8051 is ready to transfer another byte
  - It must be noted that TI flag bit is raised by 8051 itself when it finishes data transfer
  - It must be cleared by the programmer with instruction `CLR TI`
  - If we write a byte into SBUF before the TI flag bit is raised, we risk the loss of a portion of the byte being transferred
- The TI bit can be checked by
  - The instruction `JNB TI,xx`
  - Using an interrupt

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Receiving

- In programming the 8051 to receive character bytes serially
  1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate
  2. TH1 is loaded to set baud rate
  3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits
  4. TR1 is set to 1 to start timer 1
  5. RI is cleared by `CLR RI` instruction
  6. The RI flag bit is monitored with the use of instruction `JNB RI,xx` to see if an entire character has been received yet
  7. When RI is raised, SBUF has the byte, its contents are moved into a safe place
  8. To receive the next character, go to step 5

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Receiving (cont')

Write a program for the 8051 to receive bytes of data serially, and put them in P1, set the baud rate at 4800, 8-bit data, and 1 stop bit

Solution:

```
        MOV  TMOD,#20H  ;timer 1,mode 2(auto reload)
        MOV  TH1,#-6    ;4800 baud rate
        MOV  SCON,#50H  ;8-bit, 1 stop, REN enabled
        SETB TR1        ;start timer 1
HERE:   JNB  RI,HERE    ;wait for char to come in
        MOV  A,SBUF     ;saving incoming byte in A
        MOV  P1,A       ;send to port 1
        CLR  RI         ;get ready to receive next
                        ;byte
        SJMP HERE       ;keep getting data
```

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Receiving (cont')

Example 10-5

Assume that the 8051 serial port is connected to the COM port of IBM PC, and on the PC, we are using the terminal.exe program to send and receive data serially. P1 and P2 of the 8051 are connected to LEDs and switches, respectively. Write an 8051 program to (a) send to PC the message "We Are Ready", (b) receive any data send by PC and put it on LEDs connected to P1, and (c) get data on switches connected to P2 and send it to PC serially. The program should perform part (a) once, but parts (b) and (c) continuously, use 4800 baud rate.

Solution:

```
        ORG   0
        MOV   P2,#0FFH    ;make P2 an input port
        MOV   TMOD,#20H   ;timer 1, mode 2
        MOV   TH1,#0FAH   ;4800 baud rate
        MOV   SCON,#50H   ;8-bit, 1 stop, REN enabled
        SETB  TR1         ;start timer 1
        MOV   DPTR,#MYDATA ;load pointer for message
H_1:    CLR   A
        MOV   A,@A+DPTR   ;get the character
...
```

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Receiving (cont')

Example 10-5 (cont')

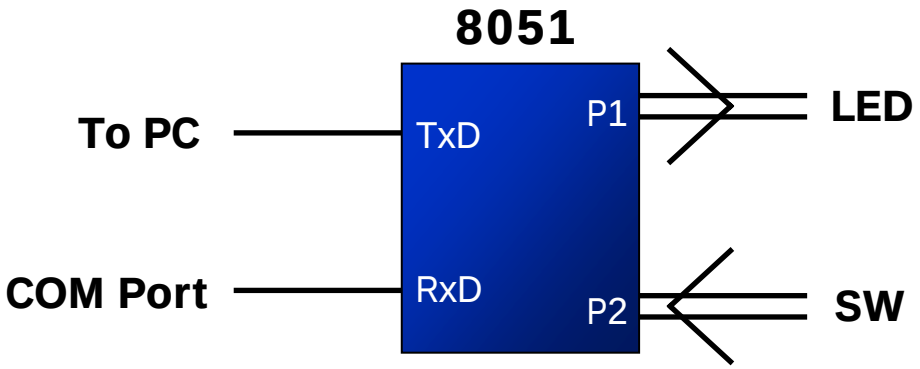
```
        JZ   B_1          ;if last character get out
        ACALL SEND        ;otherwise call transfer
        INC  DPTR         ;next one
        SJMP H_1          ;stay in loop
B_1:    MOV  a,P2         ;read data on P2
        ACALL SEND        ;transfer it serially
        ACALL RECV        ;get the serial data
        MOV  P1,A         ;display it on LEDs
        SJMP B_1          ;stay in loop indefinitely
;----serial data transfer. ACC has the data------
SEND:   MOV  SBUF,A       ;load the data
H_2:    JNB  TI,H_2       ;stay here until last bit
                          ;gone
        CLR  TI           ;get ready for next char
        RET               ;return to caller
;----Receive data serially in ACC----------------
RECV:   JNB  RI,RECV      ;wait here for char
        MOV  A,SBUF       ;save it in ACC
        CLR  RI           ;get ready for next char
        RET               ;return to caller
...
```

SERIAL COMMUNICA-TION PROGRAMMING

Programming Serial Data Receiving (cont')

Example 10-5 (cont')

```
;-----The message---------------
MYDATA: DB   "We Are Ready",0
        END
```



- In receiving bit via its RxD pin, 8051 goes through the following steps
  1. It receives the start bit
     - Indicating that the next bit is the first bit of the character byte it is about to receive
  2. The 8-bit character is received one bit at time
  3. The stop bit is received
     - When receiving the stop bit 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character

SERIAL COMMUNICA-TION PROGRAMMING

Importance of RI Flag (cont')

(cont')

4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register
   - We copy the SBUF contents to a safe place in some other register or memory before it is lost
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by `CLR RI` in order to allow the next received character byte to be placed in SBUF
   - Failure to do this causes loss of the received character

SERIAL COMMUNICA-TION PROGRAMMING

Importance of RI Flag (cont')

- By checking the RI flag bit, we know whether or not the 8051 received a character byte
  - If we failed to copy SBUF into a safe place, we risk the loss of the received byte
  - It must be noted that RI flag bit is raised by 8051 when it finish receive data
  - It must be cleared by the programmer with instruction `CLR RI`
  - If we copy SBUF into a safe place before the RI flag bit is raised, we risk copying garbage
- The RI bit can be checked by
  - The instruction `JNB RI,xx`
  - Using an interrupt

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate

- There are two ways to increase the baud rate of data transfer
  - To use a higher frequency crystal
  - To change a bit in the PCON register

The system crystal is fixed

- PCON register is an 8-bit register
  - When 8051 is powered up, SMOD is zero
  - We can set it to high by software and thereby double the baud rate

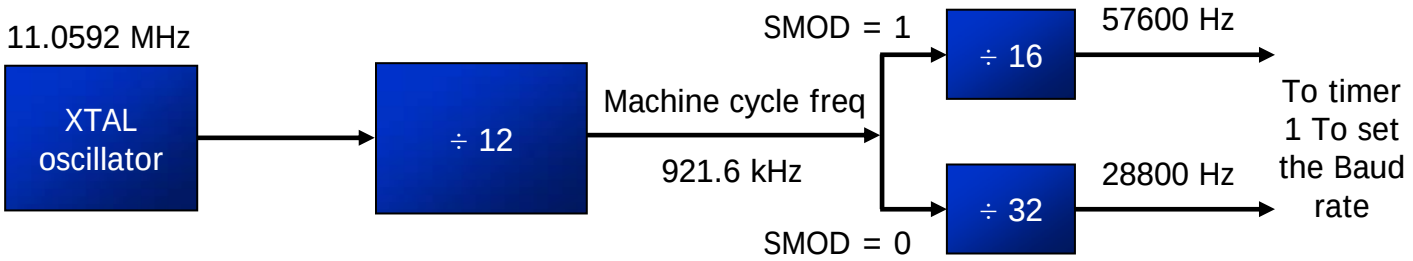
SMOD	--	--	--	GF1	GF0	PD	IDL

It is not a bit-addressable register

```
MOV  A,PCON     ;place a copy of PCON in ACC
SETB ACC.7      ;make D7=1
MOV  PCON,A     ;changing any other bits
```

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')



Baud Rate comparison for SMOD=0 and SMOD=1

TH1	(Decimal)	(Hex)	SMOD=0	SMOD=1
	-3	FD	9600	19200
	-6	FA	4800	9600
	-12	F4	2400	4800
	-24	E8	1200	2400

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')

Example 10-6

Assume that XTAL = 11.0592 MHz for the following program, state (a) what this program does, (b) compute the frequency used by timer 1 to set the baud rate, and (c) find the baud rate of the data transfer.

```
        MOV   A,PCON      ;A=PCON
        MOV   ACC.7       ;make D7=1
        MOV   PCON,A      ;SMOD=1, double baud rate
                          ;with same XTAL freq.
        MOV   TMOD,#20H   ;timer 1, mode 2
        MOV   TH1,-3      ;19200 (57600/3 =19200)
        MOV   SCON,#50H   ;8-bit data, 1 stop bit, RI
                          ;enabled
        SETB  TR1         ;start timer 1
        MOV   A,#"B"      ;transfer letter B
A_1:    CLR   TI          ;make sure TI=0
        MOV   SBUF,A      ;transfer it
H_1:    JNB   TI,H_1      ;stay here until the last
                          ;bit is gone
        SJMP  A_1         ;keep sending "B" again
```

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')

Example 10-6 (cont')

Solution:

```
(a) This program transfers ASCII letter B (01000010
    binary) continuously
(b) With XTAL = 11.0592 MHz and SMOD = 1 in the
    above program, we have:

11.0592 / 12 = 921.6 kHz machine cycle frequency.
921.6 / 16 = 57,600 Hz frequency used by timer 1
to set the baud rate.
57600 / 3 = 19,200, the baud rate.
```

Find the TH1 value (in both decimal and hex) to set the baud rate to each of the following. (a) 9600 (b) 4800 if SMOD=1. Assume that XTAL 11.0592 MHz

Solution:

With XTAL = 11.0592 and SMOD = 1, we have timer frequency = 57,600 Hz.

(a) 57600 / 9600 = 6; so TH1 = -6 or TH1 = FAH

(b) 57600 / 4800 = 12; so TH1 = -12 or TH1 = F4H

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')

Example 10-8

Find the baud rate if TH1 = -2, SMOD = 1, and XTAL = 11.0592 MHz. Is this baud rate supported by IBM compatible PCs?

Solution:

With XTAL = 11.0592 and SMOD = 1, we have timer frequency = 57,600 Hz. The baud rate is 57,600/2 = 28,800. This baud rate is not supported by the BIOS of the PCs; however, the PC can be programmed to do data transfer at such a speed. Also, HyperTerminal in Windows supports this and other baud rates.

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')

Example 10-10

Write a program to send the message "The Earth is but One Country" to serial port. Assume a SW is connected to pin P1.2. Monitor its status and set the baud rate as follows:

SW = 0, 4800 baud rate

SW = 1, 9600 baud rate

Assume XTAL = 11.0592 MHz, 8-bit data, and 1 stop bit.

Solution:

```
        SW    BIT P1.2
        ORG   0H          ;starting position
MAIN:
        MOV   TMOD,#20H
        MOV   TH1,#-6     ;4800 baud rate (default)
        MOV   SCON,#50H
        SETB  TR1
        SETB  SW          ;make SW an input
S1:     JNB   SW,SLOWSP   ;check SW status
        MOV   A,PCON      ;read PCON
        SETB  ACC.7       ;set SMOD high for 9600
        MOV   PCON,A      ;write PCON
        SJMP  OVER        ;send message
.....
```

SERIAL COMMUNICA-TION PROGRAMMING

Doubling Baud Rate (cont')

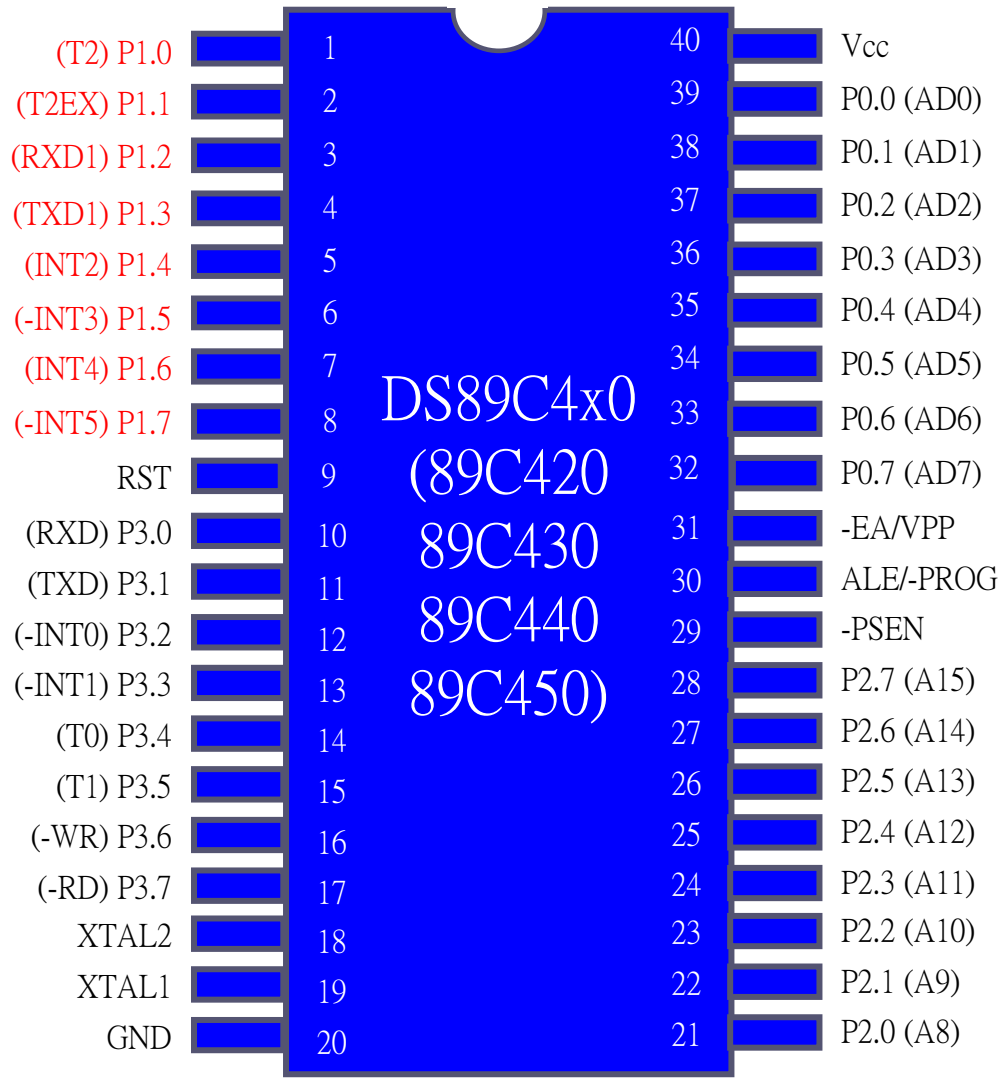
```
.....

SLOWSP:
        MOV   A,PCON      ;read PCON
        SETB  ACC.7       ;set SMOD low for 4800
        MOV   PCON,A      ;write PCON
OVER:   MOV   DPTR,#MESS1 ;load address to message
FN:     CLR   A
        MOVC  A,@A+DPTR   ;read value
        JZ    S1          ;check for end of line
        ACALL SENDCOM     ;send value to serial port
        INC   DPTR        ;move to next value
        SJMP  FN          ;repeat
;------------
SENDCOM:
        MOV   SBUF,A      ;place value in buffer
HERE:   JNB   TI,HERE     ;wait until transmitted
        CLR   TI          ;clear
        RET               ;return

;------------
MESS1: DB "The Earth is but One Country",0
        END
```

PROGRAMMING THE SECOND SERIAL PORT

- Many new generations of 8051 microcontroller come with two serial ports, like DS89C4x0 and DS80C320
  - The second serial port of DS89C4x0 uses pins P1.2 and P1.3 for the Rx and Tx lines
  - The second serial port uses some reserved SFR addresses for the SCON and SBUF
    - There is no universal agreement among the makers as to which addresses should be used
      - The SFR addresses of C0H and C1H are set aside for SBUF and SCON of DS89C4x0
    - The DS89C4x0 technical documentation refers to these registers as SCON1 and SBUF1
      - The first ones are designated as SCON0 and SBUF0

PROGRAMMING THE SECOND SERIAL PORT (cont')

DS89C4x0 pin diagram



PROGRAMMING THE SECOND SERIAL PORT (cont')

SFR Byte Addresses for DS89C4x0 Serial Ports

SFR (byte address)	First Serial Port	Second Serial Port
SCON	SCON0 = 98H	SCON1 = C0H
SBUF	SBUF0 = 99H	SBUF1 = C1H
TL	TL1 = 8BH	TL1 = 8BH
TH	TH1 = 8DH	TH1 = 8DH
TCON	TCON0 = 88H	TCON0 = 88H
PCON	PCON = 87H	PCON = 87H

PROGRAMMING THE SECOND SERIAL PORT (cont')

- Upon reset, DS89c4x0 uses Timer 1 for setting baud rate of both serial ports
  - While each serial port has its own SCON and SBUF registers, both ports can use Timer1 for setting the baud rate
  - SBUF and SCON refer to the SFR registers of the first serial port
    - Since the older 8051 assemblers do not support this new second serial port, we need to define them in program
    - To avoid confusion, in DS89C4x0 programs we use SCON0 and SBUF0 for the first and SCON1 and SBUF1for the second serial ports

PROGRAMMING THE SECOND SERIAL PORT (cont')

Example 10-11

Write a program for the second serial port of the DS89C4x0 to continuously transfer the letter "A" serially at 4800 baud. Use 8-bit data and 1 stop bit. Use Timer 1.

Solution:

```
      SBUF1 EQU 0C1H   ;2nd serial SBUF addr
      SCON1 EQU 0C0H   ;2nd serial SCON addr
      TI1   BIT 0C1H   ;2nd serial TI bit addr
      RI1   BIT 0C0H   ;2nd serial RI bit addr
      ORG   0H         ;starting position
MAIN:
      MOV  TMOD,#20H   ;COM2 uses Timer 1 on reset
      MOV  TH1,#-6     ;4800 baud rate
      MOV  SCON1,#50H  ;8-bit, 1 stop, REN enabled
      SETB TR1         ;start timer 1
AGAIN:MOV  A,#"A"      ;send char 'A'
      ACALL SENDCOM2
      SJMP AGAIN
SENDCOM2:
      MOV  SBUF1,A     ;COM2 has its own SBUF
HERE: JNB  TI1,HERE    ;COM2 has its own TI flag
      CLR  TI1
      RET
      END
```

PROGRAMMING THE SECOND SERIAL PORT (cont')

Example 10-14

Assume that a switch is connected to pin P2.0. Write a program to monitor the switch and perform the following:
(a) If SW = 0, send the message "Hello" to the Serial #0 port
(b) If SW = 1, send the message "Goodbye" to the Serial #1 port.

Solution:

```
        SCON1 EQU 0C0H
        TI1   BIT 0C1H
        SW1   BIT P2.0
        ORG   0H           ;starting position
        MOV   TMOD,#20H
        MOV   TH1,#-3      ;9600 baud rate
        MOV   SCON,#50H
        MOV   SCON1,#50H
        SETB  TR1
        SETB  SW1          ;make SW1 an input
S1:     JB    SW1,NEXT     ;check SW1 status
        MOV   DPTR,#MESS1;if SW1=0 display "Hello"
FN:     CLR   A
        MOVC  A,@A+DPTR    ;read value
        JZ    S1           ;check for end of line
        ACALL SENDCOM1     ;send to serial port
        INC   DPTR         ;move to next value
        SJM   FN
.....
```

PROGRAMMING THE SECOND SERIAL PORT (cont')

```
.....

NEXT:   MOV  DPTR,#MESS2;if SW1=1 display “Goodbye”
LN:     CLR  A
        MOVC A,@A+DPTR  ;read value
        JZ   S1         ;check for end of line
        ACALL SENDCOM2  ;send to serial port
        INC  DPTR       ;move to next value
        SJM  LN

SENDCOM1:
        MOV  SBUF,A     ;place value in buffer
HERE:   JNB  TI,HERE    ;wait until transmitted
        CLR  TI         ;clear
        RET
;------------
SENDCOM2:
        MOV  SBUF1,A    ;place value in buffer
HERE1:  JNB  TI1,HERE1  ;wait until transmitted
        CLR  TI1        ;clear
        RET

MESS1:  DB “Hello”,0
MESS2:  DB “Goodbye”,0
        END
```

SERIAL PORT PROGRAMMING IN C

Transmitting and Receiving Data

Example 10-15
Write a C program for 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
  TMOD=0x20;          //use Timer 1, mode 2
  TH1=0xFA;           //4800 baud rate
  SCON=0x50;
  TR1=1;
  while (1) {
    SBUF='A';         //place value in buffer
    while (TI==0);
    TI=0;
  }
}
```

SERIAL PORT PROGRAMMING IN C

Transmitting and Receiving Data (cont')

Example 10-16
Write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
#include <reg51.h>
void SerTx(unsigned char);
void main(void){
  TMOD=0x20;          //use Timer 1, mode 2
  TH1=0xFD;           //9600 baud rate
  SCON=0x50;
  TR1=1;              //start timer
  while (1) {
    SerTx('Y');
    SerTx('E');
    SerTx('S');
  }
}
void SerTx(unsigned char x){
  SBUF=x;             //place value in buffer
  while (TI==0);      //wait until transmitted
  TI=0;
}
```

SERIAL PORT PROGRAMMING IN C

Transmitting and Receiving Data (cont')

Example 10-17
Program the 8051 in C to receive bytes of data serially and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
  unsigned char mybyte;
  TMOD=0x20;          //use Timer 1, mode 2
  TH1=0xFA;           //4800 baud rate
  SCON=0x50;
  TR1=1;              //start timer
  while (1) {         //repeat forever
    while (RI==0);    //wait to receive
    mybyte=SBUF;      //save value
    P1=mybyte;        //write value to port
    RI=0;
  }
}
```

SERIAL PORT PROGRAMMING IN C

Transmitting and Receiving Data (cont')

Example 10-19
Write an 8051 C Program to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and set the baud rate as follows:
SW = 0, 28,800 baud rate
SW = 1, 56K baud rate
Assume that XTAL = 11.0592 MHz for both cases.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0;      //input switch
void main(void){
  unsigned char z;
  unsigned char Mess1[]="Normal Speed";
  unsigned char Mess2[]="High Speed";
  TMOD=0x20;         //use Timer 1, mode 2
  TH1=0xFF;          //28800 for normal
  SCON=0x50;
  TR1=1;             //start timer
.....
```

SERIAL PORT PROGRAMMING IN C

Transmitting and Receiving Data (cont')

```
.....
  if(MYSW==0) {
    for (z=0;z<12;z++) {
       SBUF=Mess1[z]; //place value in buffer
       while(TI==0);  //wait for transmit
       TI=0;
    }
  }
  else {
    PCON=PCON|0x80;   //for high speed of 56K
    for (z=0;z<10;z++) {
       SBUF=Mess2[z]; //place value in buffer
       while(TI==0);  //wait for transmit
       TI=0;
    }
  }
}
```

SERIAL PORT PROGRAMMING IN C

C Compilers and the Second Serial Port

Example 10-20
Write a C program for the DS89C4x0 to transfer the letter "A" serially at 4800 baud continuously. Use the second serial port with 8-bit data and 1 stop bit. We can only use Timer 1 to set the baud rate.

Solution:

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit TI1=0xC1;
void main(void){
  TMOD=0x20;          //use Timer 1, mode 2
  TH1=0xFA;           //4800 baud rate
  SCON=0x50;          //use 2nd serial port SCON1
  TR1=1;              //start timer
  while (1) {
    SBUF1='A';        //use 2nd serial port SBUF1
    while (TI1==0); //wait for transmit
    TI1=0;
  }
}
```

SERIAL PORT PROGRAMMING IN C

C Compilers and the Second Serial Port

Example 10-21
Program the DS89C4x0 in C to receive bytes of data serially via the second serial port and put them in P1. Set the baud rate at 9600, 8-bit data and 1 stop bit. Use Timer 1 for baud rate generation.

Solution:

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit RI1=0xC0;
void main(void){
  unsigned char mybyte;
  TMOD=0x20;          //use Timer 1, mode 2
  TH1=0xFD;           //9600 baud rate
  SCON1=0x50;         //use 2nd serial port SCON1
  TR1=1;              //start timer
  while (1) {
    while (RI1==0); //monitor RI1
    mybyte=SBUF1;   //use SBUF1
    P2=mybyte;      //place value on port
    RI1=0;
  }
}
```

INTERRUPTS PROGRAMMING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

INTERRUPTS

Interrupts vs. Polling

- An *interrupt* is an external or internal event that interrupts the microcontroller to inform it that a device needs its service
- A single microcontroller can serve several devices by two ways
  - Interrupts
    - Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal
    - Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device
    - The program which is associated with the interrupt is called the *interrupt service routine* (ISR) or *interrupt handler*

INTERRUPTS

Interrupts vs. Polling (cont')

- (cont')
  - Polling
    - The microcontroller continuously monitors the status of a given device
    - When the conditions met, it performs the service
    - After that, it moves on to monitor the next device until every one is serviced
- Polling can monitor the status of several devices and serve each of them as certain conditions are met
  - The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service
  - ex. `JNB TF,target`

INTERRUPTS

Interrupts vs. Polling (cont')

- The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time)
  - Each devices can get the attention of the microcontroller based on the assigned priority
  - For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion
- The microcontroller can also ignore (mask) a device request for service
  - This is not possible for the polling method

- For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler
  - When an interrupt is invoked, the micro-controller runs the interrupt service routine
  - For every interrupt, there is a fixed location in memory that holds the address of its ISR
  - The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table

INTERRUPTS

Steps in Executing an Interrupt

- Upon activation of an interrupt, the microcontroller goes through the following steps
  1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack
  2. It also saves the current status of all the interrupts internally (i.e: not on the stack)
  3. It jumps to a fixed location in memory, called the interrupt vector table, that holds the address of the ISR

INTERRUPTS

Steps in Executing an Interrupt (cont')

(cont')

4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it
   - It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt)
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted
   - First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC
   - Then it starts to execute from that address

- Six interrupts are allocated as follows
  - Reset – power-up reset
  - Two interrupts are set aside for the timers: one for timer 0 and one for timer 1
  - Two interrupts are set aside for hardware external interrupts
    - P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2)
  - Serial communication has a single interrupt that belongs to both receive and transfer

INTERRUPTS

Six Interrupts in 8051 (cont')

Interrupt vector table

Interrupt	ROM Location (hex)	Pin
Reset	0000	9
External HW (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
External HW (INT1)	0013	P3.3 (13)
Timer 1 (TF1)	001B	
Serial COM (RI and TI)	0023	

```
        ORG   0      ;wake-up ROM reset location
        LJMP MAIN    ;by-pass int. vector table
;---- the wake-up program
        ORG   30H
MAIN:
        ....
        END
```

Only three bytes of ROM space assigned to the reset pin. We put the LJMP as the first instruction and redirect the processor away from the interrupt vector table.

INTERRUPTS

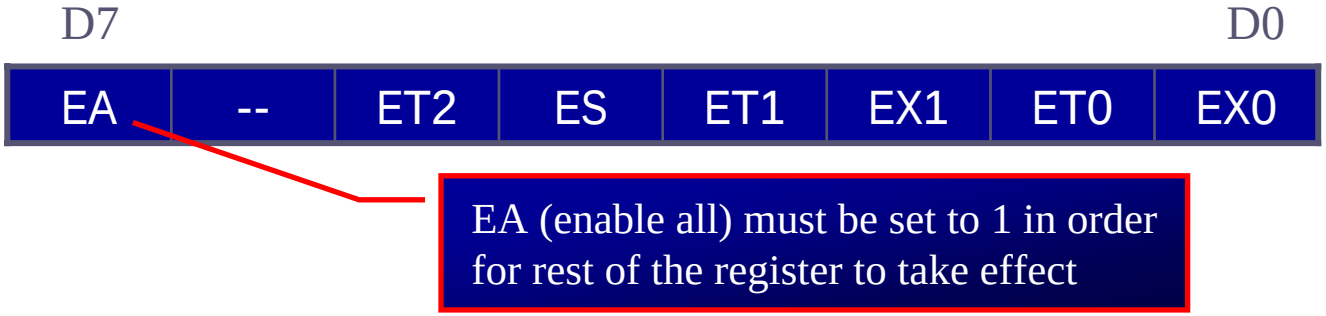
Enabling and Disabling an Interrupt

- Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated
- The interrupts must be enabled by software in order for the microcontroller to respond to them
  - There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts

INTERRUPTS

Enabling and Disabling an Interrupt (cont')

IE (Interrupt Enable) Register



EA	IE.7	Disables all interrupts
--	IE.6	Not implemented, reserved for future use
ET2	IE.5	Enables or disables timer 2 overflow or capture interrupt (8952)
ES	IE.4	Enables or disables the serial port interrupt
ET1	IE.3	Enables or disables timer 1 overflow interrupt
EX1	IE.2	Enables or disables external interrupt 1
ET0	IE.1	Enables or disables timer 0 overflow interrupt
EX0	IE.0	Enables or disables external interrupt 0

- To enable an interrupt, we take the following steps:
  1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect
  2. The value of EA
     - If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high
     - If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high

INTERRUPTS

Enabling and Disabling an Interrupt (cont')

Example 11-1

Show the instructions to (a) enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 (EX1),and (b) disable (mask) the timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

Solution:

```
(a) MOV  IE,#10010110B ;enable serial,
                        ;timer 0, EX1
```

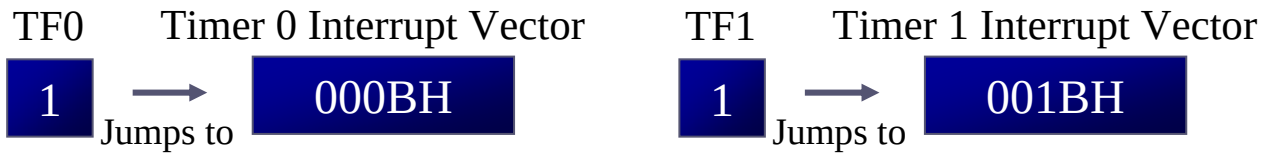
Another way to perform the same manipulation is

```
SETB IE.7  ;EA=1, global enable
SETB IE.4  ;enable serial interrupt
SETB IE.1  ;enable Timer 0 interrupt
SETB IE.2  ;enable EX1
```

```
(b) CLR  IE.1   ;mask (disable) timer 0
                ;interrupt only
```

```
(c) CLR  IE.7   ;disable all interrupts
```

- The timer flag (TF) is raised when the timer rolls over
  - In polling TF, we have to wait until the TF is raised
    - The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and can not do anything else
  - Using interrupts solves this problem and, avoids tying down the controller
    - If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR
    - In this way, the microcontroller can do other until it is notified that the timer has rolled over



TIMER INTERRUPTS (cont')

Example 11-2

Write a program that continuously get 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 μs period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use timer 0 in mode 2 (auto reload). TH0 = 100/1.085 us = 92

```
;--upon wake-up go to main, avoid using
;memory allocated to Interrupt Vector Table
     ORG  0000H
     LJMP MAIN  ;by-pass interrupt vector table
;
;--ISR for timer 0 to generate square wave
     ORG  000BH ;Timer 0 interrupt vector table
     CPL  P2.1  ;toggle P2.1 pin
     RETI       ;return from ISR
...
```

TIMER INTERRUPTS (cont')

```
...
;--The main program for initialization
       ORG  0030H    ;after vector table space
MAIN:  MOV  TMOD,#02H ;Timer 0, mode 2
       MOV  P0,#0FFH ;make P0 an input port
       MOV  TH0,#-92 ;TH0=A4H for -92
       MOV  IE,#82H  ;IE=10000010 (bin) enable
                     ;Timer 0
       SETB TR0      ;Start Timer 0
BACK:  MOV  A,P0     ;get data from P0
       MOV  P1,A     ;issue it to P1
       SJMP BACK     ;keep doing it loop
                     ;unless interrupted by TF0
       END
```

TIMER INTERRUPTS (cont')

Example 11-3

Rewrite Example 11-2 to create a square wave that has a high portion of 1085 us and a low portion of 15 us. Assume XTAL=11.0592MHz. Use timer 1.

Solution:

Since 1085 us is 1000 × 1.085 we need to use mode 1 of timer 1.

```
;--upon wake-up go to main, avoid using
;memory allocated to Interrupt Vector Table
       ORG  0000H
       LJMP MAIN      ;by-pass int. vector table
;--ISR for timer 1 to generate square wave
       ORG  001BH     ;Timer 1 int. vector table
       LJMP ISR_T1    ;jump to ISR
...
```

TIMER INTERRUPTS (cont')

```
...
;--The main program for initialization
        ORG   0030H      ;after vector table space
MAIN:   MOV   TMOD,#10H  ;Timer 1, mode 1
        MOV   P0,#0FFH   ;make P0 an input port
        MOV   TL1,#018H  ;TL1=18 low byte of -1000
        MOV   TH1,#0FCH  ;TH1=FC high byte of -1000
        MOV   IE,#88H    ;10001000 enable Timer 1 int
        SETB  TR1        ;Start Timer 1
BACK:   MOV   A,P0       ;get da
        MOV   P1,A       ;issue
        SJMP  BACK       ;keep c
;Timer 1 ISR. Must be reloaded, not auto-reload
ISR_T1: CLR TR1          ;stop Timer 1
        MOV R2,#4        ;                          2MC
        CLR P2.1        ;P2.1=0, start of low portion
HERE:   DJNZ R2,HERE     ;4x2 machine cycle         8MC
        MOV TL1,#18H     ;load T1 low byte value    2MC
        MOV TH1,#0FCH    ;load T1 high byte value   2MC
        SETB TR1         ;starts timer1             1MC
        SETB P2.1        ;P2.1=1,back to high       1MC
        RETI             ;return to main
        END
```

Low portion of the pulse is created by 14 MC
14 x 1.085 us = 15.19 us

TIMER INTERRUPTS (cont')

Example 11-4

Write a program to generate a square wave if 50Hz frequency on pin P1.2. This is similar to Example 9-12 except that it uses an interrupt for timer 0. Assume that XTAL=11.0592 MHz

Solution:

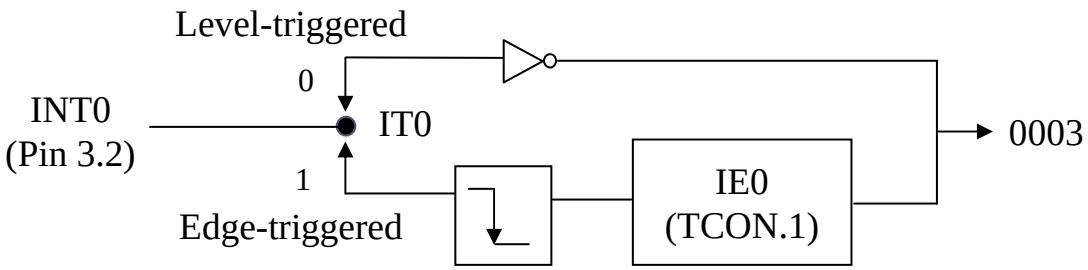
```
      ORG  0
      LJMP MAIN
      ORG  000BH  ;ISR for Timer 0
      CPL  P1.2
      MOV  TL0,#00
      MOV  TH0,#0DCH
      RETI
      ORG  30H
;--------main program for initialization
MAIN:MOV  TMOD,#00000001B  ;Timer 0, Mode 1
      MOV  TL0,#00
      MOV  TH0,#0DCH
      MOV  IE,#82H   ;enable Timer 0 interrupt
      SETB TR0
HERE:SJMP HERE
      END
```

EXTERNAL HARDWARE INTERRUPTS

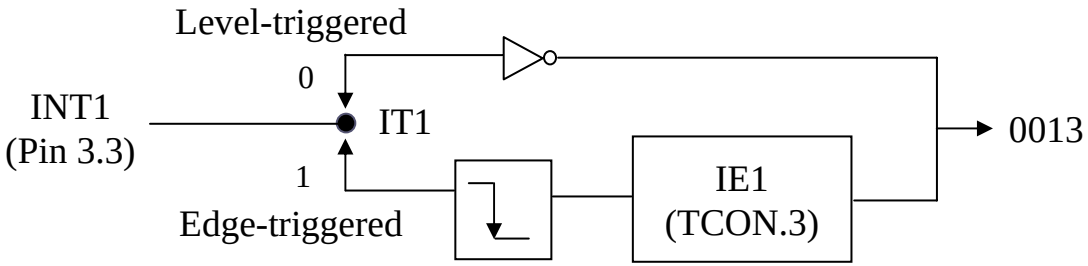
- The 8051 has two external hardware interrupts
  - Pin 12 (P3.2) and pin 13 (P3.3) of the 8051, designated as INT0 and INT1, are used as external hardware interrupts
    - The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1
  - There are two activation levels for the external hardware interrupts
    - Level trigged
    - Edge trigged

EXTERNAL HARDWARE INTERRUPTS (cont')

Activation of INT0



Activation of INT1



EXTERNAL HARDWARE INTERRUPTS

Level-Triggered Interrupt

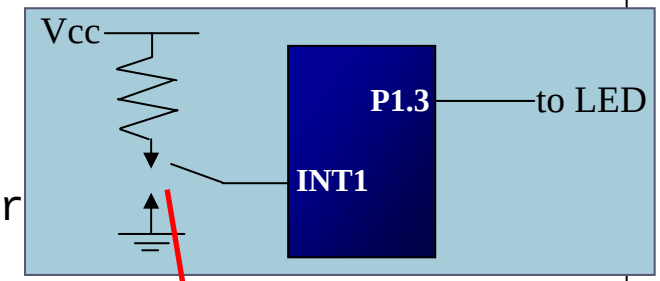
- In the level-triggered mode, INT0 and INT1 pins are normally high
  - If a low-level signal is applied to them, it triggers the interrupt
  - Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt
  - The low-level signal at the INT pin must be removed before the execution of the last instruction of the ISR, RETI; otherwise, another interrupt will be generated
- This is called a *level-triggered* or *level-activated* interrupt and is the default mode upon reset of the 8051

EXTERNAL HARDWARE INTERRUPTS

Level-Triggered Interrupt (cont')

Example 11-5

Assume that the INT1 pin is connected to a switch that is normally high. Whenever it goes low, it should turn on an LED. The LED is connected to P1.3 and is normally off. When it is turned on it should stay on for a fraction of a second. As long as the switch is pressed low, the LED should stay on.



Solution:

```
        ORG   0000H
        LJMP  MAIN ;by-pass inter
                   ;vector table
;--ISR for INT1 to turn on LED
        ORG   0013H      ;INT1 ISR
        SETB  P1.3       ;turn on LED
        MOV   R3,#255
BACK:   DJNZ  R3,BACK    ;keep LED on for a
        CLR   P1.3       ;turn off the LED
        RETI             ;return from ISR

;--MAIN program for initialization
        ORG   30H
MAIN:   MOV   IE,#10000100B ;enable external INT 1
HERE:   SJMP  HERE       ;stay here until get interrupted
        END
```

Pressing the switch will cause the LED to be turned on. If it is kept activated, the LED stays on

EXTERNAL HARDWARE INTERRUPTS

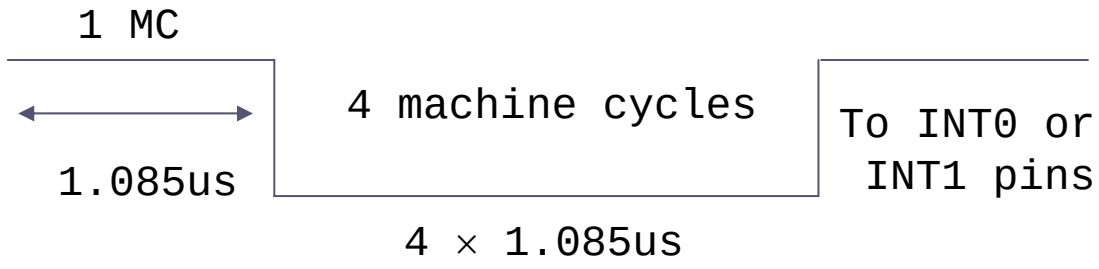
Sampling Low Level-Triggered Interrupt

- Pins P3.2 and P3.3 are used for normal I/O unless the INT0 and INT1 bits in the IE register are enabled
  - After the hardware interrupts in the IE register are enabled, the controller keeps sampling the INTn pin for a low-level signal once each machine cycle
  - According to one manufacturer's data sheet,
    - The pin must be held in a low state until the start of the execution of ISR
    - If the INTn pin is brought back to a logic high before the start of the execution of ISR there will be no interrupt
    - If INTn pin is left at a logic low after the RETI instruction of the ISR, another interrupt will be activated after one instruction is executed

EXTERNAL HARDWARE INTERRUPTS

Sampling Low Level-Triggered Interrupt (cont')

- To ensure the activation of the hardware interrupt at the INTn pin, make sure that the duration of the low-level signal is around 4 machine cycles, but no more
  - This is due to the fact that the level-triggered interrupt is not latched
  - Thus the pin must be held in a low state until the start of the ISR execution



note: On reset, IT0 (TCON.0) and IT1 (TCON.2) are both low, making external interrupt level-triggered

EXTERNAL HARDWARE INTERRUPTS

Edge-Triggered Interrupt

- To make INT0 and INT1 edge-triggered interrupts, we must program the bits of the TCON register
  - The TCON register holds, among other bits, the IT0 and IT1 flag bits that determine level- or edge-triggered mode of the hardware interrupt
    - IT0 and IT1 are bits D0 and D2 of the TCON register
    - They are also referred to as TCON.0 and TCON.2 since the TCON register is bit-addressable

EXTERNAL HARDWARE INTERRUPTS

Edge-Triggered Interrupt (cont')

TCON (Timer/Counter) Register (Bit-addressable)

D7							D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off

EXTERNAL HARDWARE INTERRUPTS

Edge-Triggered Interrupt (cont')

TCON (Timer/Counter) Register (Bit-addressable) (cont')

IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt

EXTERNAL HARDWARE INTERRUPTS

Edge-Triggered Interrupt (cont')

Assume that pin 3.3 (INT1) is connected to a pulse generator, write a program in which the falling edge of the pulse will send a high to P1.3, which is connected to an LED (or buzzer). In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT1 pin.

When the falling edge of the signal is applied to pin INT1, the LED will be turned on momentarily.

Solution:

```
        ORG   0000H
        LJMP  MAIN
;--ISR for hardware interrupt INT1 to turn on LED
        ORG   0013H   ;INT1 ISR
        SETB  P1.3    ;turn on LED
        MOV   R3,#255
BACK:   DJNZ  R3,BACK ;keep the buzzer on for a while
        CLR   P1.3    ;turn off the buzzer
        RETI          ;return from ISR
;------MAIN program for initialization
        ORG   30H
MAIN:   SETB  TCON.2 ;make INT1 edge-triggered int.
        MOV   IE,#10000100B ;enable External INT 1
HERE:   SJMP  HERE    ;stay here until get interrupted
        END
```

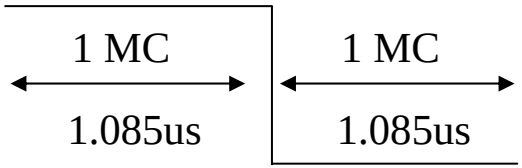
The on-state duration depends on the time delay inside the ISR for INT1

EXTERNAL HARDWARE INTERRUPTS

Sampling Edge-Triggered Interrupt

- In edge-triggered interrupts
  - The external source must be held high for at least one machine cycle, and then held low for at least one machine cycle
  - The falling edge of pins INT0 and INT1 are latched by the 8051 and are held by the TCON.1 and TCON.3 bits of TCON register
    - Function as interrupt-in-service flags
    - It indicates that the interrupt is being serviced now and on this INTn pin, and no new interrupt will be responded to until this service is finished

```
Minimum pulse duration to
detect edge-triggered
interrupts XTAL=11.0592MHz
```



EXTERNAL HARDWARE INTERRUPTS

Sampling Edge-Triggered Interrupt (cont')

- Regarding the IT0 and IT1 bits in the TCON register, the following two points must be emphasized
  - When the ISRs are finished (that is, upon execution of RETI), these bits (TCON.1 and TCON.3) are cleared, indicating that the interrupt is finished and the 8051 is ready to respond to another interrupt on that pin
  - During the time that the interrupt service routine is being executed, the INTn pin is ignored, no matter how many times it makes a high-to-low transition
    - RETI clears the corresponding bit in TCON register (TCON.1 or TCON.3)
    - There is no need for instruction `CLR TCON.1` before RETI in the ISR associated with INT0

EXTERNAL HARDWARE INTERRUPTS

Sampling Edge-Triggered Interrupt (cont')

Example 11-7
What is the difference between the RET and RETI instructions? Explain why we can not use RET instead of RETI as the last instruction of an ISR.

Solution:
Both perform the same actions of popping off the top two bytes of the stack into the program counter, and marking the 8051 return to where it left off.

However, RETI also performs an additional task of clearing the interrupt-in-service flag, indicating that the servicing of the interrupt is over and the 8051 now can accept a new interrupt on that pin. If you use RET instead of RETI as the last instruction of the interrupt service routine, you simply block any new interrupt on that pin after the first interrupt, since the pin status would indicate that the interrupt is still being serviced. In the cases of TF0, TF1, TCON.1, and TCON.3, they are cleared due to the execution of RETI.

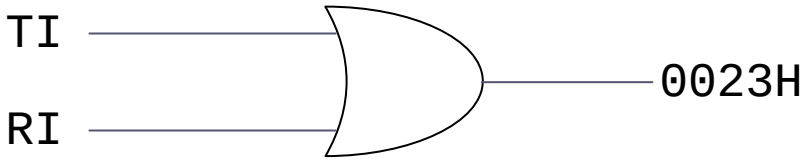
SERIAL COMMUNI-CATION INTERRUPT

- TI (transfer interrupt) is raised when the last bit of the framed data, the stop bit, is transferred, indicating that the SBUF register is ready to transfer the next byte
- RI (received interrupt) is raised when the entire frame of data, including the stop bit, is received
  - In other words, when the SBUF register has a byte, RI is raised to indicate that the received byte needs to be picked up before it is lost (overrun) by new incoming serial data

SERIAL COMMUNI-CATION INTERRUPT

RI and TI Flags and Interrupts

- In the 8051 there is only one interrupt set aside for serial communication
  - This interrupt is used to both send and receive data
  - If the interrupt bit in the IE register (IE.4) is enabled, when RI or TI is raised the 8051 gets interrupted and jumps to memory location 0023H to execute the ISR
  - In that ISR we must examine the TI and RI flags to see which one caused the interrupt and respond accordingly



Serial interrupt is invoked by TI or RI flags

SERIAL COMMUNI-CATION INTERRUPT

Use of Serial COM in 8051

- The serial interrupt is used mainly for receiving data and is never used for sending data serially
  - This is like getting a telephone call in which we need a ring to be notified
  - If we need to make a phone call there are other ways to remind ourselves and there is no need for ringing
  - However in receiving the phone call, we must respond immediately no matter what we are doing or we will miss the call

SERIAL COMMUNI-CATION INTERRUPT

Use of Serial COM in 8051 (cont')

Example 11-8

Write a program in which the 8051 reads data from P1 and writes it to P2 continuously while giving a copy of it to the serial COM port to be transferred serially. Assume that XTAL=11.0592. Set the baud rate at 9600.

Solution:

```
        ORG   0000H
        LJMP  MAIN
        ORG   23H
        LJMP  SERIAL     ;jump to serial int ISR
        ORG   30H
MAIN:   MOV   P1,#0FFH   ;make P1 an input port
        MOV   TMOD,#20H  ;timer 1, auto reload
        MOV   TH1,#0FDH  ;9600 baud rate
        MOV   SCON,#50H  ;8-bit,1 stop, ren enabled
        MOV   IE,10010000B ;enable serial int.
        SETB  TR1        ;start timer 1
BACK:   MOV   A,P1       ;read data from port 1
        MOV   SBUF,A     ;give a copy to SBUF
        MOV   P2,A       ;send it to P2
        SJMP  BACK       ;stay in loop indefinitely
...
```

SERIAL COMMUNI-CATION INTERRUPT

Use of Serial COM in 8051 (cont')

```
...
;------------------SERIAL PORT ISR
        ORG 100H
SERIAL: JB  TI,TRANS;jump if TI is high
        MOV A,SBUF  ;otherwise due to receive
        CLR RI      ;clear RI since CPU doesn't
        RETI        ;return from ISR
TRANS:  CLR TI      ;clear TI since CPU doesn't
        RETI        ;return from ISR
        END
```

The moment a byte is written into SBUF it is framed and transferred serially. As a result, when the last bit (stop bit) is transferred the TI is raised, and that causes the serial interrupt to be invoked since the corresponding bit in the IE register is high. In the serial ISR, we check for both TI and RI since both could have invoked interrupt.

SERIAL COMMUNI-CATION INTERRUPT

Use of Serial COM in 8051 (cont')

Example 11-9

Write a program in which the 8051 gets data from P1 and sends it to P2 continuously while incoming data from the serial port is sent to P0. Assume that XTAL=11.0592. Set the baud rata at 9600.

Solution:

```
        ORG   0000H
        LJMP  MAIN
        ORG   23H
        LJMP  SERIAL     ;jump to serial int ISR
        ORG   30H
MAIN:   MOV   P1,#0FFH   ;make P1 an input port
        MOV   TMOD,#20H  ;timer 1, auto reload
        MOV   TH1,#0FDH  ;9600 baud rate
        MOV   SCON,#50H  ;8-bit,1 stop, ren enabled
        MOV   IE,10010000B ;enable serial int.
        SETB  TR1        ;start timer 1
BACK:   MOV   A,P1       ;read data from port 1
        MOV   P2,A       ;send it to P2
        SJMP  BACK       ;stay in loop indefinitely
...
```

SERIAL COMMUNI-CATION INTERRUPT

Use of Serial COM in 8051 (cont')

```
...
;------------------SERIAL PORT ISR
         ORG 100H
SERIAL:  JB  TI,TRANS;jump if TI is high
         MOV A,SBUF  ;otherwise due to receive
         MOV P0,A    ;send incoming data to P0
         CLR RI      ;clear RI since CPU doesn't
         RETI        ;return from ISR
TRANS:   CLR TI      ;clear TI since CPU doesn't
         RETI        ;return from ISR
         END
```

SERIAL COMMUNI-CATION INTERRUPT

Clearing RI and TI before RETI

Example 11-10

Write a program using interrupts to do the following:

(a) Receive data serially and sent it to P0,

(b) Have P1 port read and transmitted serially, and a copy given to P2,

(c) Make timer 0 generate a square wave of 5kHz frequency on P0.1.

Assume that XTAL-11,0592. Set the baud rate at 4800.

Solution:

```
        ORG   0
        LJMP  MAIN
        ORG   000BH   ;ISR for timer 0
        CPL   P0.1    ;toggle P0.1
        RETI          ;return from ISR
        ORG   23H     ;
        LJMP  SERIAL  ;jump to serial interrupt ISR
        ORG   30H
MAIN:   MOV   P1,#0FFH ;make P1 an input port
        MOV   TMOD,#22H;timer 1,mode 2(auto reload)
        MOV   TH1,#0F6H;4800 baud rate
        MOV   SCON,#50H;8-bit, 1 stop, ren enabled
        MOV   TH0,#-92 ;for 5kHZ wave
...
```

SERIAL COMMUNI-CATION INTERRUPT

Clearing RI and TI before RETI (cont')

```
...
        MOV  IE,10010010B ;enable serial int.
        SETB TR1          ;start timer 1
        SETB TR0          ;start timer 0
BACK:   MOV  A,P1         ;read data from port 1
        MOV  SBUF,A       ;give a copy to SBUF
        MOV  P2,A         ;send it to P2
        SJMP BACK         ;stay in loop indefinitely
;-----------------SERIAL PORT ISR
        ORG  100H
SERIAL:JB    TI,TRANS;jump if TI is high
        MOV  A,SBUF       ;otherwise due to receive
        MOV  P0,A         ;send serial data to P0
        CLR  RI           ;clear RI since CPU doesn't
        RETI              ;return from ISR
TRANS: CLR  TI            ;clear TI since CPU doesn't
        RETI              ;return from ISR
        END
```

SERIAL COMMUNI-CATION INTERRUPT

Interrupt Flag Bits

- The TCON register holds four of the interrupt flags, in the 8051 the SCON register has the RI and TI flags

Interrupt Flag Bits

Interrupt	Flag	SFR Register Bit
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial Port	T1	SCON.1
Timer 2	TF2	T2CON.7 (AT89C52)
Timer 2	EXF2	T2CON.6 (AT89C52)

INTERRUPT PRIORITY

- When the 8051 is powered up, the priorities are assigned according to the following
  - In reality, the priority scheme is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed and responds accordingly

Interrupt Priority Upon Reset

Highest To Lowest Priority	
External Interrupt 0	(INT0)
Timer Interrupt 0	(TF0)
External Interrupt 1	(INT1)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)

INTERRUPT PRIORITY (cont')

Example 11-11
Discuss what happens if interrupts INT0, TF0, and INT1 are activated at the same time. Assume priority levels were set by the power-up reset and the external hardware interrupts are edge-triggered.

Solution:
If these three interrupts are activated at the same time, they are latched and kept internally. Then the 8051 checks all five interrupts according to the sequence listed in Table 11-3. If any is activated, it services it in sequence. Therefore, when the above three interrupts are activated, IE0 (external interrupt 0) is serviced first, then timer 0 (TF0), and finally IE1 (external interrupt 1).

INTERRUPT PRIORITY (cont')

- We can alter the sequence of interrupt priority by assigning a higher priority to any one of the interrupts by programming a register called IP (interrupt priority)
  - To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high
  - When two or more interrupt bits in the IP register are set to high
    - While these interrupts have a higher priority than others, they are serviced according to the sequence of Table 11-13

INTERRUPT PRIORITY (cont')

Interrupt Priority Register (Bit-addressable)

D7							D0
--	--	PT2	PS	PT1	PX1	PT0	PX0

--	IP.7	Reserved
--	IP.6	Reserved
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)
PS	IP.4	Serial port interrupt priority bit
PT1	IP.3	Timer 1 interrupt priority bit
PX1	IP.2	External interrupt 1 priority bit
PT0	IP.1	Timer 0 interrupt priority bit
PX0	IP.0	External interrupt 0 priority bit

Priority bit=1 assigns high priority

Priority bit=0 assigns low priority

INTERRUPT PRIORITY (cont')

Example 11-12

(a) Program the IP register to assign the highest priority to INT1(external interrupt 1), then

(b) discuss what happens if INT0, INT1, and TF0 are activated at the same time. Assume the interrupts are both edge-triggered.

Solution:

(a) `MOV IP,#00000100B ;IP.2=1` assign INT1 higher priority. The instruction `SETB IP.2` also will do the same thing as the above line since IP is bit-addressable.

(b) The instruction in Step (a) assigned a higher priority to INT1 than the others; therefore, when INT0, INT1, and TF0 interrupts are activated at the same time, the 8051 services INT1 first, then it services INT0, then TF0. This is due to the fact that INT1 has a higher priority than the other two because of the instruction in Step (a). The instruction in Step (a) makes both the INT0 and TF0 bits in the IP register 0. As a result, the sequence in Table 11-3 is followed which gives a higher priority to INT0 over TF0

INTERRUPT PRIORITY (cont')

Example 11-13

Assume that after reset, the interrupt priority is set the instruction `MOV IP,#00001100B`. Discuss the sequence in which the interrupts are serviced.

Solution:

The instruction "MOV IP #00001100B" (B is for binary) and timer 1 (TF1)to a higher priority level compared with the reset of the interrupts. However, since they are polled according to Table, they will have the following priority.

Highest Priority	External Interrupt 1	(INT1)
	Timer Interrupt 1	(TF1)
	External Interrupt 0	(INT0)
	Timer Interrupt 0	(TF0)
Lowest Priority	Serial Communication	(RI+TI)

INTERRUPT PRIORITY

Interrupt inside an Interrupt

- In the 8051 a low-priority interrupt can be interrupted by a higher-priority interrupt but not by another low-priority interrupt
  - Although all the interrupts are latched and kept internally, no low-priority interrupt can get the immediate attention of the CPU until the 8051 has finished servicing the high-priority interrupts

INTERRUPT PRIORITY

Triggering Interrupt by Software

- To test an ISR by way of simulation can be done with simple instructions to set the interrupts high and thereby cause the 8051 to jump to the interrupt vector table
  - ex. If the IE bit for timer 1 is set, an instruction such as SETB TF1 will interrupt the 8051 in whatever it is doing and will force it to jump to the interrupt vector table
    - We do not need to wait for timer 1 go roll over to have an interrupt

- The 8051 compiler have extensive support for the interrupts
  - They assign a unique number to each of the 8051 interrupts

Interrupt	Name	Numbers
External Interrupt 0	(INT0)	0
Timer Interrupt 0	(TF0)	1
External Interrupt 1	(INT1)	2
Timer Interrupt 1	(TF1)	3
Serial Communication	(RI + TI)	4
Timer 2 (8052 only)	(TF2)	5

  - It can assign a register bank to an ISR
    - This avoids code overhead due to the pushes and pops of the R0 – R7 registers

PROGRAMMING IN C
(cont')

Example 11-14

Write a C program that continuously gets a single bit of data from P1.7 and sends it to P1.0, while simultaneously creating a square wave of 200 μs period on pin P2.5. Use Timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use timer 0 mode 2 (auto-reload). One half of the period is 100 μs. 100/1.085 μs = 92, and TH0 = 256 - 92 = 164 or A4H

```
#include <reg51.h>
sbit SW   =P1^7;
sbit IND  =P1^0;
sbit WAVE =P2^5;
void timer0(void) interrupt 1 {
  WAVE=~WAVE;  //toggle pin
}
void main() {
  SW=1;          //make switch input
  TMOD=0x02;
  TH0=0xA4;      //TH0=-92
  IE=0x82;       //enable interrupt for timer 0
  while (1) {
    IND=SW;      //send switch to LED
  }
}
```

PROGRAMMING IN C
(cont')

Example 11-16

Write a C program using interrupts to do the following:
(a) Receive data serially and send it to P0
(b) Read port P1, transmit data serially, and give a copy to P2
(c) Make timer 0 generate a square wave of 5 kHz frequency on P0.1
Assume that XTAL = 11.0592 MHz. Set the baud rate at 4800.

Solution:

```
#include <reg51.h>
sbit WAVE =P0^1;

void timer0() interrupt 1 {
  WAVE=~WAVE;   //toggle pin
}

void serial0() interrupt 4 {
  if (TI==1) {
    TI=0;       //clear interrupt
  }
  else {
    P0=SBUF;    //put value on pins
    RI=0;       //clear interrupt
  }
}
.....
```

PROGRAMMING IN C (cont')

```
.....

void main() {
  unsigned char x;
  P1=0xFF;      //make P1 an input
  TMOD=0x22;
  TH1=0xF6;     //4800 baud rate
  SCON=0x50;
  TH0=0xA4;     //5 kHz has T=200us
  IE=0x92;      //enable interrupts
  TR1=1;        //start timer 1
  TR0=1;        //start timer 0
  while (1) {
    x=P1;       //read value from pins
    SBUF=x;     //put value in buffer
    P2=x;       //write value to pins
  }
}
```

PROGRAMMING IN C (cont')

Example 11-17

Write a C program using interrupts to do the following:
(a) Generate a 10 KHz frequency on P2.1 using T0 8-bit auto-reload
(b) Use timer 1 as an event counter to count up a 1-Hz pulse and display it on P0. The pulse is connected to EX1.
Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

```
#include <reg51.h>
sbit WAVE =P2^1;
Unsigned char cnt;

void timer0() interrupt 1 {
  WAVE=~WAVE;   //toggle pin
}

void timer1() interrupt 3 {
  cnt++;          //increment counter
  P0=cnt;         //display value on pins
}
.....
```

PROGRAMMING IN C (cont')

```
.....

void main() {
  cnt=0;          //set counter to 0
  TMOD=0x42;
  TH0=0x-46;      //10 KHz
  IE=0x86;        //enable interrupts
  TR0=1;          //start timer 0
  while (1);      //wait until interrupted
}
```

8031/51 INTERFACING TO EXTERNAL MEMORY

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University

SEMI-CONDUCTOR MEMORY

Memory Capacity

- The number of bits that a semiconductor memory chip can store is called chip *capacity*
  - It can be in units of Kbits (kilobits), Mbits (megabits), and so on
- This must be distinguished from the storage capacity of computer systems
  - While the memory capacity of a memory IC chip is always given bits, the memory capacity of a computer system is given in bytes
    - 16M memory chip – 16 megabits
    - A computer comes with 16M memory – 16 megabytes

SEMI-CONDUCTOR MEMORY

Memory Organization

- Memory chips are organized into a number of locations within the IC
  - Each location can hold 1 bit, 4 bits, 8 bits, or even 16 bits, depending on how it is designed internally
    - The number of locations within a memory IC depends on the address pins
    - The number of bits that each location can hold is always equal to the number of data pins
- To summarize
  - A memory chip contain 2^x location, where x is the number of address pins
  - Each location contains y bits, where y is the number of data pins on the chip
  - The entire chip will contain $2^x \times y$ bits

SEMI-CONDUCTOR MEMORY

Speed

- One of the most important characteristics of a memory chip is the speed at which its data can be accessed
  - To access the data, the address is presented to the address pins, the READ pin is activated, and after a certain amount of time has elapsed, the data shows up at the data pins
  - The shorter this elapsed time, the better, and consequently, the more expensive the memory chip
  - The speed of the memory chip is commonly referred to as its *access time*

SEMI-CONDUCTOR MEMORY

Speed (cont')

Example

A given memory chip has 12 address pins and 4 data pins. Find: (a) The organization, and (b) the capacity.

Solution:

(a) This memory chip has 4096 locations ($2^{12} = 4096$), and each location can hold 4 bits of data. This gives an organization of 4096×4, often represented as $4K \times 4$.

(b) The capacity is equal to 16K bits since there is a total of 4K locations and each location can hold 4 bits of data.

Example

A 512K memory chip has 8 pins for data. Find:

(a) The organization, and (b) the number of address pins for this memory chip.

Solution:

(a) A memory chip with 8 data pins means that each location within the chip can hold 8 bits of data. To find the number of locations within this memory chip, divide the capacity by the number of data pins. 512K/8 = 64K; therefore, the organization for this memory chip is $64K \times 8$

(b) The chip has 16 address lines since $2^{16} = 64K$

SEMI-CONDUCTOR MEMORY

ROM (Read-only Memory)

- ROM is a type of memory that does not lose its contents when the power is turned off
  - ROM is also called *nonvolatile* memory
- There are different types of read-only memory
  - PROM
  - EPROM
  - EEPROM
  - Flash EPROM
  - Mask ROM

SEMI-CONDUCTOR MEMORY

ROM

PROM (Programmable ROM)

- PROM refers to the kind of ROM that the user can burn information into
  - PROM is a user-programmable memory
  - For every bit of the PROM, there exists a fuse
- If the information burned into PROM is wrong, that PROM must be discarded since its internal fuses are blown permanently
  - PROM is also referred to as OTP (one-time programmable)
  - Programming ROM, also called *burning* ROM, requires special equipment called a ROM burner or ROM programmer

SEMI-CONDUCTOR MEMORY

ROM

EPROM (Erasable Programmable ROM)

- EPROM was invented to allow making changes in the contents of PROM after it is burned
  - In EPROM, one can program the memory chip and erase it thousands of times
- A widely used EPROM is called UV-EPROM
  - UV stands for ultra-violet
  - The only problem with UV-EPROM is that erasing its contents can take up to 20 minutes
  - All UV-EPROM chips have a window that is used to shine ultraviolet (UV) radiation to erase its contents

SEMI-CONDUCTOR MEMORY

ROM

EPROM (Erasable Programmable ROM) (cont')

- To program a UV-EPROM chip, the following steps must be taken:
  - Its contents must be erased
    - To erase a chip, it is removed from its socket on the system board and placed in EPROM erasure equipment to expose it to UV radiation for 15-20 minutes
  - Program the chip
    - To program a UV-EPROM chip, place it in the ROM burner
    - To burn code or data into EPROM, the ROM burner uses 12.5 volts, V_{pp} in the UV-EPROM data sheet or higher, depending on the EPROM type
    - Place the chip back into its system board socket

SEMI-CONDUCTOR MEMORY

ROM

EPROM (Erasable Programmable ROM) (cont')

- There is an EPROM programmer (burner), and there is also separate EPROM erasure equipment
- The major disadvantage of UV-EPROM, is that it cannot be programmed while in the system board
- Notice the pattern of the IC numbers
  - Ex. 27128-25 refers to UV-EPROM that has a capacity of 128K bits and access time of 250 nanoseconds
    - 27xx always refers to UV-EPROM chips

For ROM chip 27128, find the number of data and address pins.

Solution:
The 27128 has a capacity of 128K bits. It has 16K × 8 organization (all ROMs have 8 data pins), which indicates that there are 8 pins for data, and 14 pins for address (2^{14} = 16K)

SEMI-CONDUCTOR MEMORY

ROM

EEPROM (Electrically Erasable Programmable ROM)

- EEPROM has several advantage over EPROM
  - Its method of erasure is electrical and therefore instant, as opposed to the 20-minute erasure time required for UV-EPROM
  - One can select which byte to be erased, in contrast to UV-EPROM, in which the entire contents of ROM are erased
  - One can program and erase its contents while it is still in the system board
    - EEPROM does not require an external erasure and programming device
    - The designer incorporate into the system board the circuitry to program the EEPROM

SEMI-CONDUCTOR MEMORY

ROM

Flash Memory EPROM

- Flash EPROM has become a popular user-programmable memory chip since the early 1990s
  - The process of erasure of the entire contents takes less than a second, or might say in a flash
    - The erasure method is electrical
    - It is commonly called flash memory
  - The major difference between EEPROM and flash memory is
    - Flash memory's contents are erased, then the entire device is erased
      - There are some flash memories are recently made so that the erasure can be done block by block
    - One can erase a desired section or byte on EEPROM

SEMI-CONDUCTOR MEMORY

ROM

Flash Memory EPROM (cont')

- It is believed that flash memory will replace part of the hard disk as a mass storage medium
  - The flash memory can be programmed while it is in its socket on the system board
    - Widely used as a way to upgrade PC BIOS ROM
  - Flash memory is semiconductor memory with access time in the range of 100 ns compared with disk access time in the range of tens of milliseconds
  - Flash memory's program/erase cycles must become infinite, like hard disks
    - Program/erase cycle refers to the number of times that a chip can be erased and programmed before it becomes unusable
    - The program/erase cycle is 100,000 for flash and EEPROM, 1000 for UV-EPROM

SEMI-CONDUCTOR MEMORY

ROM

Mask ROM

- Mask ROM refers to a kind of ROM in which the contents are programmed by the IC manufacturer, not user-programmable
  - The terminology mask is used in IC fabrication
  - Since the process is costly, mask ROM is used when the needed volume is high and it is absolutely certain that the contents will not change
  - The main advantage of mask ROM is its cost, since it is significantly cheaper than other kinds of ROM, but if an error in the data/code is found, the entire batch must be thrown away

SEMI-CONDUCTOR MEMORY

RAM (Random Access Memory)

- RAM memory is called *volatile* memory since cutting off the power to the IC will result in the loss of data
  - Sometimes RAM is also referred to as RAWM (read and write memory), in contrast to ROM, which cannot be written to
- There are three types of RAM
  - Static RAM (SRAM)
  - NV-RAM (nonvolatile RAM)
  - Dynamic RAM (DRAM)

SEMI-CONDUCTOR MEMORY

RAM

SRAM (Static RAM)

- Storage cells in static RAM memory are made of flip-flops and therefore do not require refreshing in order to keep their data
- The problem with the use of flip-flops for storage cells is that each cell require at least 6 transistors to build, and the cell holds only 1 bit of data
  - In recent years, the cells have been made of 4 transistors, which still is too many
  - The use of 4-transistor cells plus the use of CMOS technology has given birth to a high-capacity SRAM, but its capacity is far below DRAM

SEMI-CONDUCTOR MEMORY

RAM

NV-RAM (Nonvolatile RAM)

- NV-RAM combines the best of RAM and ROM
  - The read and write ability of RAM, plus the nonvolatility of ROM
- NV-RAM chip internally is made of the following components
  - It uses extremely power-efficient SRAM cells built out of CMOS
  - It uses an internal lithium battery as a backup energy source
  - It uses an intelligent control circuitry
    - The main job of this control circuitry is to monitor the V_{cc} pin constantly to detect loss of the external power supply

SEMI-CONDUCTOR MEMORY

RAM

Checksum Byte 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

SEMI-CONDUCTOR MEMORY

RAM

Checksum Byte 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 that is the checksum byte, which 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

SEMI-CONDUCTOR MEMORY

RAM

Checksum Byte 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.

SEMI-CONDUCTOR MEMORY

RAM

DRAM (Dynamic RAM)

- Dynamic RAM uses a capacitor to store each bit
  - It cuts down the number of transistors needed to build the cell
  - It requires constant refreshing due to leakage
- The advantages and disadvantages of DRAM memory
  - The major advantages are high density (capacity), cheaper cost per bit, and lower power consumption per bit
  - The disadvantages is that
    - it must be refreshed periodically, due to the fact that the capacitor cell loses its charge;
    - While it is being refreshed, the data cannot be accessed

- In DRAM there is a problem of packing a large number of cells into a single chip with the normal number of pins assigned to addresses
  - Using conventional method of data access, large number of pins defeats the purpose of high density and small packaging
    - For example, a 64K-bit chip (64K×1) must have 16 address lines and 1 data line, requiring 16 pins to send in the address
  - The method used is to split the address in half and send in each half of the address through the same pins, thereby requiring fewer address pins

SEMI-CONDUCTOR MEMORY

RAM

Packing Issue in DRAM (cont')

- Internally, the DRAM structure is divided into a square of rows and columns
- The first half of the address is called the row and the second half is called column
  - The first half of the address is sent in through the address pins, and by activating RAS (row address strobe), the internal latches inside DRAM grab the first half of the address
  - After that, the second half of the address is sent in through the same pins, and by activating CAS (column address strobe), the internal latches inside DRAM latch the second half of the address

- In the discussion of ROM, we noted that all of them have 8 pins for data
  - This is not the case for DRAM memory chips, which can have any of the x1, x4, x8, x16 organizations

Discuss the number of pins set aside for address in each of the following memory chips. (a) 16K×4 DRAM (b) 16K×4 SRAM

Solution :

Since 2^{14} = 16K :

(a) For DRAM we have 7 pins (A0-A6) for the address pins and 2 pins for RAS and CAS

(b) For SRAM we have 14 pins for address and no pins for RAS and CAS since they are associated only with DRAM. In both cases we have 4 pins for the data bus.

MEMORY ADDRESS DECODING

- The CPU provides the address of the data desired, but it is the job of the decoding circuitry to locate the selected memory block
  - Memory chips have one or more pins called CS (chip select), which must be activated for the memory's contents to be accessed
  - Sometimes the chip select is also referred to as chip enable (CE)

MEMORY ADDRESS DECODING (cont')

- In connecting a memory chip to the CPU, note the following points
  - The data bus of the CPU is connected directly to the data pins of the memory chip
  - Control signals RD (read) and WR (memory write) from the CPU are connected to the OE (output enable) and WE (write enable) pins of the memory chip
  - In the case of the address buses, while the lower bits of the address from the CPU go directly to the memory chip address pins, the upper ones are used to activate the CS pin of the memory chip

MEMORY ADDRESS DECODING (cont')

- Normally memories are divided into blocks and the output of the decoder selects a given memory block
  - Using simple logic gates
  - Using the 74LS138
  - Using programmable logics

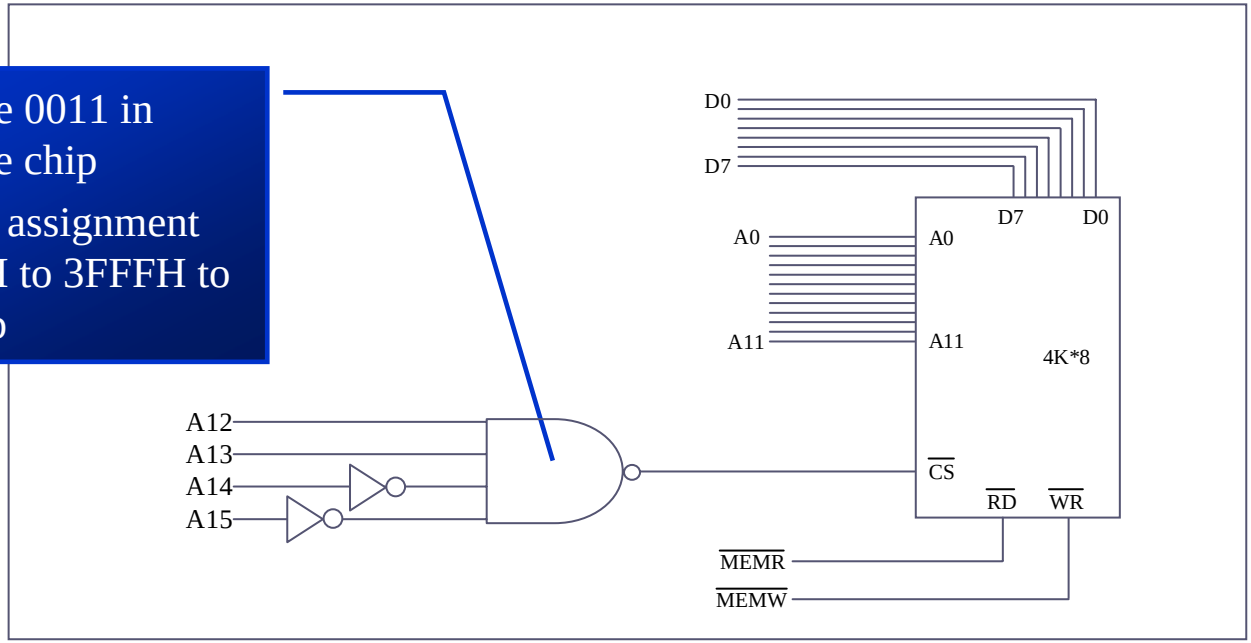
MEMORY ADDRESS DECODING

Simple Logic Gate Address Decoder

- The simplest way of decoding circuitry is the use of NAND or other gates
  - The fact that the output of a NAND gate is active low, and that the CS pin is also active low makes them a perfect match

A15-A12 must be 0011 in order to select the chip

This result in the assignment of address 3000H to 3FFFH to this memory chip



MEMORY ADDRESS DECODING

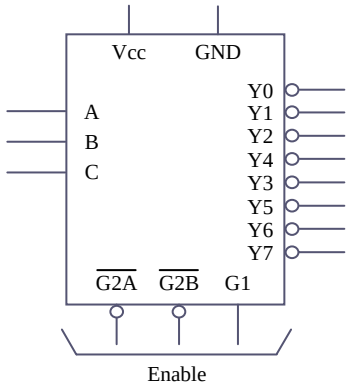
Using 74LS138 3-8 Decoder

- This is one of the most widely used address decoders
  - The 3 inputs A, B, and C generate 8 active-low outputs Y0 – Y7
    - Each Y output is connected to CS of a memory chip, allowing control of 8 memory blocks by a single 74LS138
  - In the 74LS138, where A, B, and C select which output is activated, there are three additional inputs, G2A, G2B, and G1
    - G2A and G2B are both active low, and G1 is active high
    - If any one of the inputs G1, G2A, or G2B is not connected to an address signal, they must be activated permanently either by V_{cc} or ground, depending on the activation level

MEMORY ADDRESS DECODING

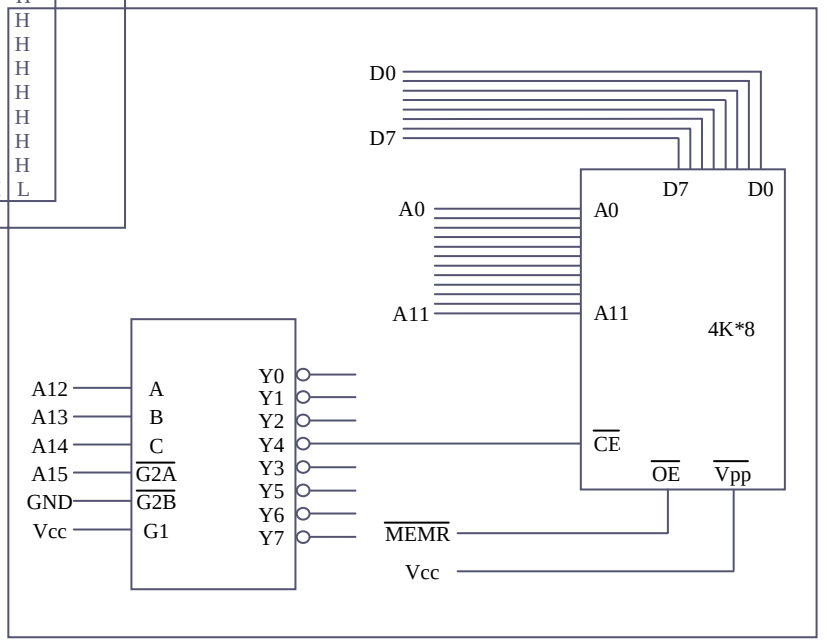
Using 74LS138 3-8 Decoder (cont')

74LS138 Decoder



Function Table

Inputs			Outputs							
Enable		Select								
G1	G2	C B A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X X X	H	H	H	H	H	H	H	H
L	X	X X X	H	H	H	H	H	H	H	H
H	L	L L L	L	H	H	H	H	H	H	H
H	L	L L H	H	L	H	H	H	H	H	H
H	L	L H L	H	H	L	H	H	H	H	H
H	L	L H H	H	H	H	L	H	H	H	H
H	L	H L L	H	H	H	H	L	H	H	H
H	L	H L H	H	H	H	H	H	L	H	H
H	L	H H L	H	H	H	H	H	H	L	H
H	L	H H H	H	H	H	H	H	H	H	L



MEMORY ADDRESS DECODING

Using 74LS138 3-8 Decoder (cont')

Looking at the design in Figure 14-6, find the address range for the Following. (a) Y4, (b) Y2, and (c) Y7.

Solution :

(a) The address range for Y4 is calculated as follows.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1

The above shows that the range for Y4 is 4000H to 4FFFH. In Figure 14-6, notice that A15 must be 0 for the decoder to be activated. Y4 will be selected when A14 A13 A12 = 100 (4 in binary). The remaining A11-A0 will be 0 for the lowest address and 1 for the highest address.

(b) The address range for Y2 is 2000H to 2FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1

(c) The address range for Y7 is 7000H to 7FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

MEMORY ADDRESS DECODING

Using Programmable Logic

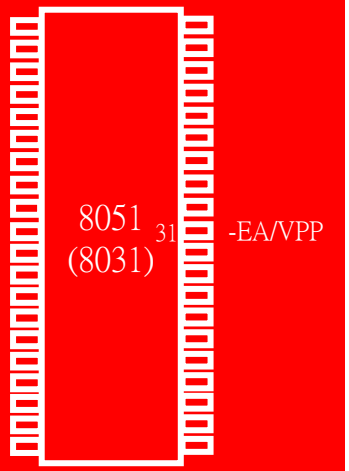
- Other widely used decoders are programmable logic chips such as PAL and GAL chips
  - One disadvantage of these chips is that one must have access to a PAL/GAL software and burner, whereas the 74LS138 needs neither of these
  - The advantage of these chips is that they are much more versatile since they can be programmed for any combination of address ranges

INTERFACING EXTERNAL ROM

- The 8031 chip is a ROMless version of the 8051
  - It is exactly like any member of the 8051 family as far as executing the instructions and features are concerned, but it has no on-chip ROM
  - To make the 8031 execute 8051 code, it must be connected to external ROM memory containing the program code
- 8031 is ideal for many systems where the on-chip ROM of 8051 is not sufficient, since it allows the program size to be as large as 64K bytes

INTERFACING EXTERNAL ROM

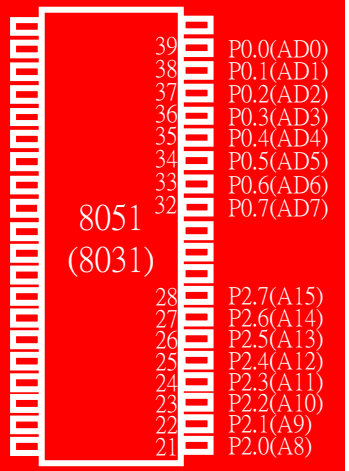
EA Pin



- For 8751/89C51/DS5000-based system, we connected the EA pin to V_{CC} to indicate that the program code is stored in the microcontroller's on-chip ROM
  - To indicate that the program code is stored in external ROM, this pin must be connected to GND

INTERFACING EXTERNAL ROM

P0 and P2 in Providing Address

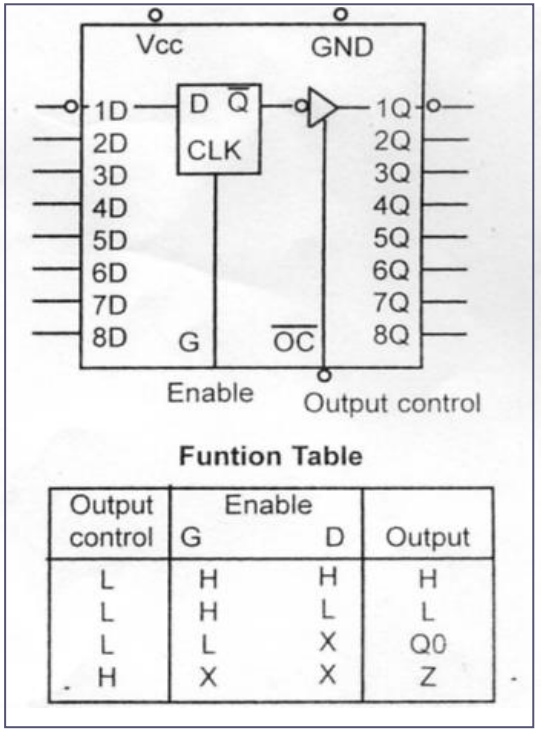


- Since the PC (program counter) of the 8031/51 is 16-bit, it is capable of accessing up to 64K bytes of program code
  - In the 8031/51, port 0 and port 2 provide the 16-bit address to access external memory
    - P0 provides the lower 8 bit address A0 – A7, and P2 provides the upper 8 bit address A8 – A15
    - P0 is also used to provide the 8-bit data bus D0 – D7
  - P0.0 – P0.7 are used for both the address and data paths
    - address/data multiplexing

INTERFACING EXTERNAL ROM

P0 and P2 in Providing Address (cont')

- ALE (address latch enable) pin is an output pin for 8031/51
  - ALE = 0, P0 is used for data path
  - ALE = 1, P0 is used for address path
- To extract the address from the P0 pins we connect P0 to a 74LS373 and use the ALE pin to latch the address



Funtion Table

Output control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

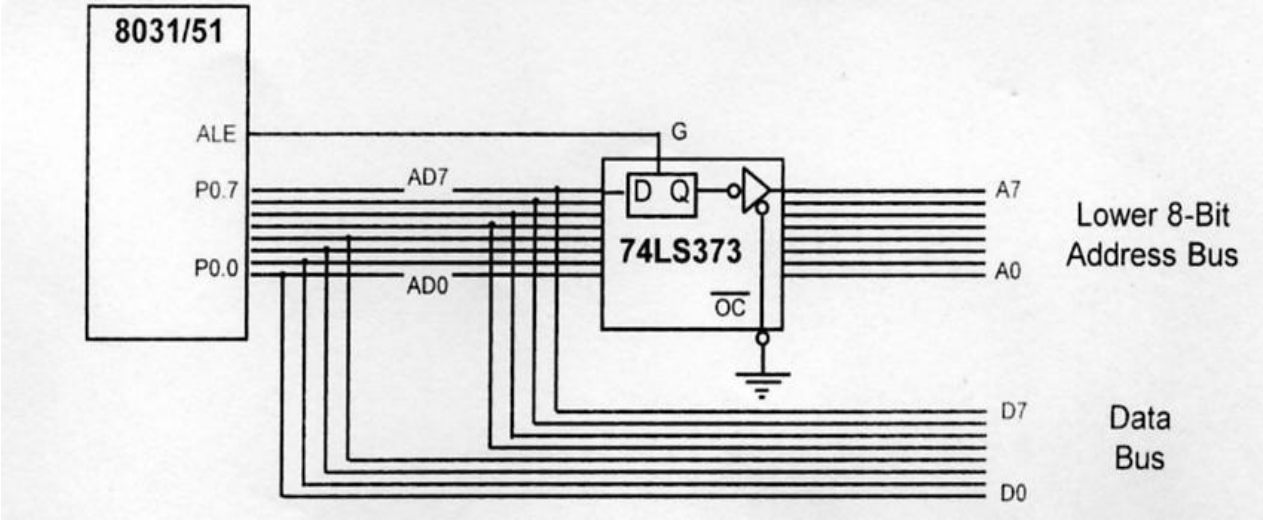
74LS373 D Latch

INTERFACING EXTERNAL ROM

P0 and P2 in Providing Address (cont')

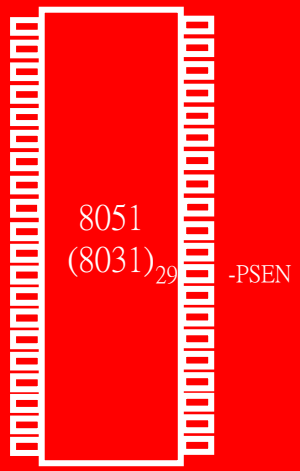
- Normally ALE = 0, and P0 is used as a data bus, sending data out or bringing data in
- Whenever the 8031/51 wants to use P0 as an address bus, it puts the addresses A0 – A7 on the P0 pins and activates ALE = 1

Address/Data Multiplexing



INTERFACING EXTERNAL ROM

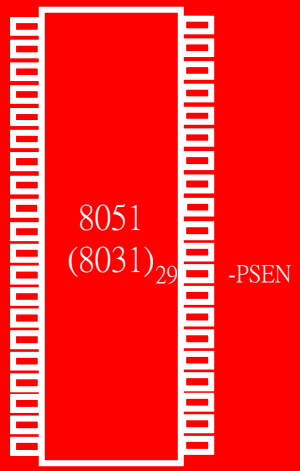
PSEN



- PSEN (program store enable) signal is an output signal for the 8031/51 microcontroller and must be connected to the OE pin of a ROM containing the program code
- It is important to emphasize the role of EA and PSEN when connecting the 8031/51 to external ROM
  - When the EA pin is connected to GND, the 8031/51 fetches opcode from external ROM by using PSEN

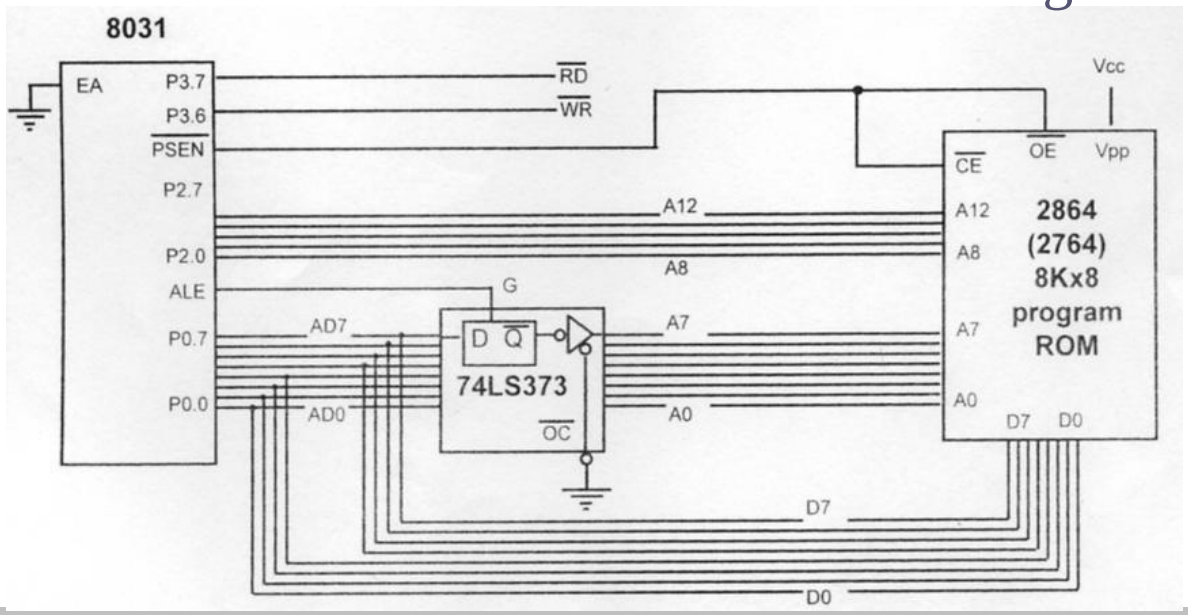
INTERFACING EXTERNAL ROM

PSEN (cont')



- The connection of the PSEN pin to the OE pin of ROM
  - In systems based on the 8751/89C51/ DS5000 where EA is connected to V_{CC}, these chips do not activate the PSEN pin
    - This indicates that the on-chip ROM contains program code

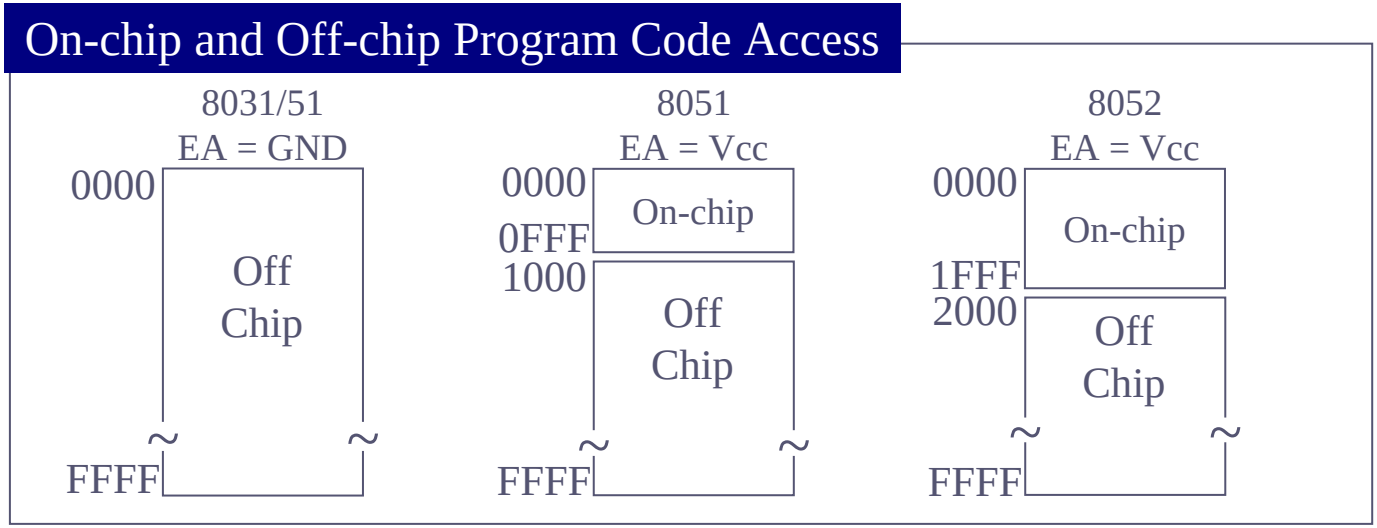
Connection to External Program ROM



INTERFACING EXTERNAL ROM

On-Chip and Off-Chip Code ROM

- In an 8751 system we could use on-chip ROM for boot code and an external ROM will contain the user's program
  - We still have EA = V_{cc},
    - Upon reset 8051 executes the on-chip program first, then
    - When it reaches the end of the on-chip ROM, it switches to external ROM for rest of program



INTERFACING EXTERNAL ROM

On-Chip and Off-Chip Code ROM (cont')

Discuss the program ROM space allocation for each of the following cases.

(a) EA = 0 for the 8751 (89C51) chip.

(b) EA = V_{cc} with both on-chip and off-chip ROM for the 8751.

(c) EA = V_{cc} with both on-chip and off-chip ROM for the 8752.

Solution:

(a) When EA = 0, the EA pin is strapped to GND, and all program fetches are directed to external memory regardless of whether or not the 8751 has some on-chip ROM for program code. This external ROM can be as high as 64K bytes with address space of 0000 – FFFFH. In this case an 8751(89C51) is the same as the 8031 system.

(b) With the 8751 (89C51) system where EA=V_{cc}, it fetches the program code of address 0000 – 0FFFH from on-chip ROM since it has 4K bytes of on-chip program ROM and any fetches from addresses 1000H – FFFFH are directed to external ROM.

(c) With the 8752 (89C52) system where EA=V_{cc}, it fetches the program code of addresses 0000 – 1FFFH from on-chip ROM since it has 8K bytes of on-chip program ROM and any fetches from addresses 2000H – FFFFH are directed to external ROM

8051 DATA MEMORY SPACE

Data Memory Space

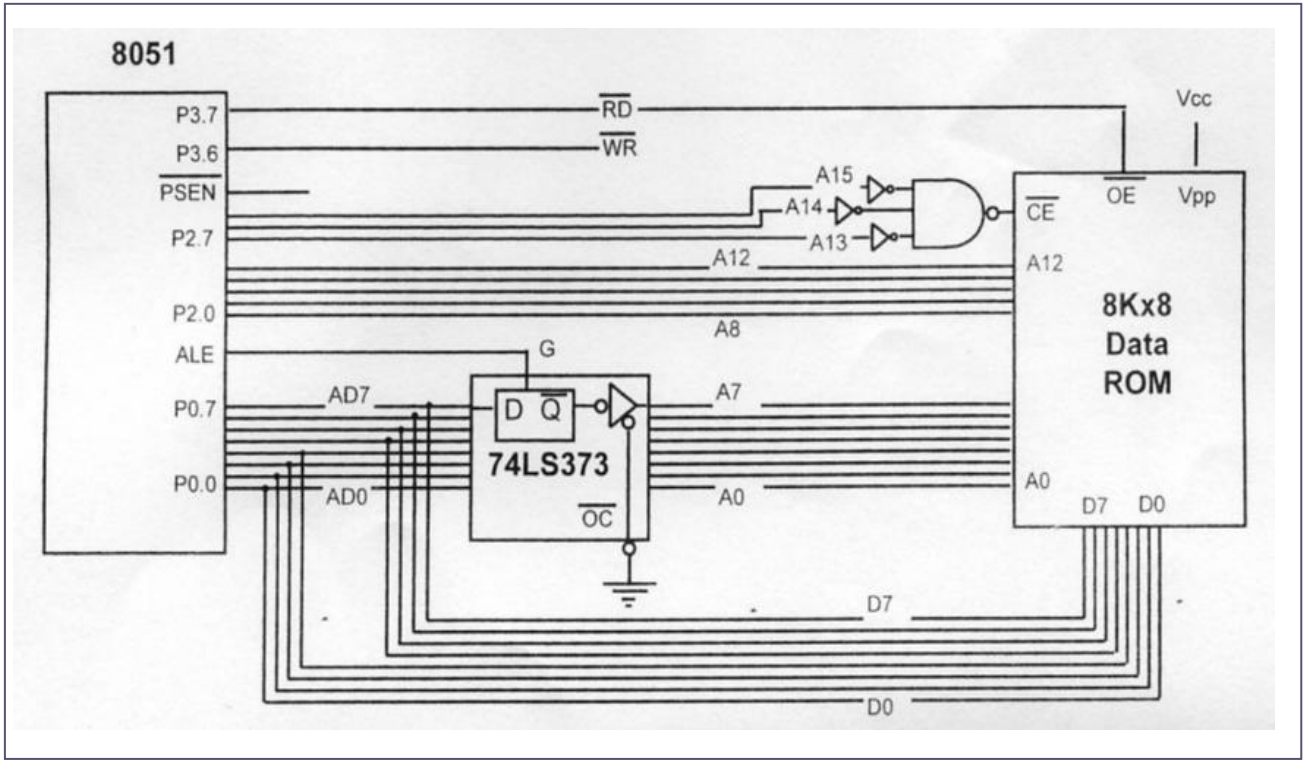
- The 8051 has 128K bytes of address space
  - 64K bytes are set aside for program code
    - Program space is accessed using the program counter (PC) to locate and fetch instructions
    - In some example we placed data in the code space and used the instruction

      `MOVC A,@A+DPTR` to get data, where C stands for code
  - The other 64K bytes are set aside for data
    - The data memory space is accessed using the DPTR register and an instruction called `MOVX`, where X stands for external
      - The data memory space must be implemented externally

8051 DATA MEMORY SPACE

External ROM for Data

- We use RD to connect the 8031/51 to external ROM containing data
  - For the ROM containing the program code, PSEN is used to fetch the code



8051 Connection to External Data ROM

8051 DATA MEMORY SPACE

MOVX Instruction

- MOVX is a widely used instruction allowing access to external data memory space
  - To bring externally stored data into the CPU, we use the instruction

    `MOVX A,@DPTR`

An external ROM uses the 8051 data space to store the look-up table (starting at 1000H) for DAC data. Write a program to read 30 Bytes of these data and send it to P1.

Solution:

```
MYXDATA  EQU   1000H
COUNT    EQU   30
          …
         MOV   DPTR,#MYXDATA
         MOV   R2,#COUNT
AGAIN:   MOVX  A,@DPTR
         MOV   P1,A
         INC   DPTR
         DJNZ  R2,AGAIN
```

Although both `MOVC A,@A+DPTR` and `MOVX A,@DPTR` look very similar, one is used to get data in the code space and the other is used to get data in the data space of the microcontroller

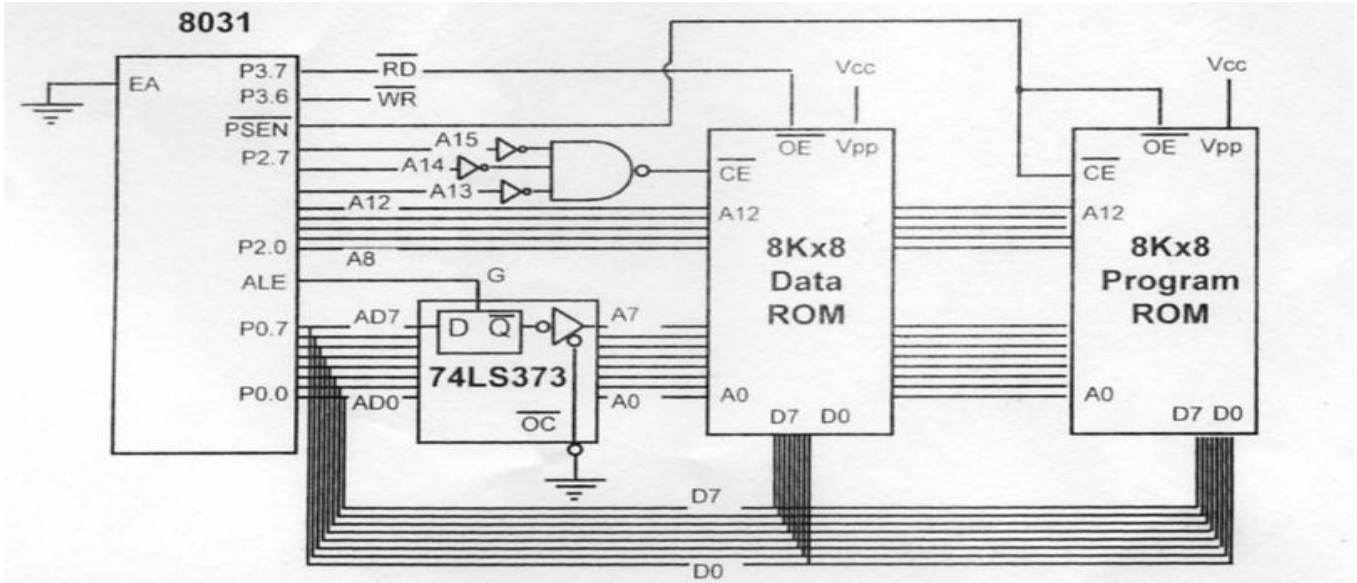
8051 DATA MEMORY SPACE

MOVX Instruction (cont')

Show the design of an 8031-based system with 8K bytes of program ROM and 8K bytes of data ROM.

Solution:

Figure 14-14 shows the design. Notice the role of PSEN and RD in each ROM. For program ROM, PSEN is used to activate both OE and CE. For data ROM, we use RD to active OE, while CE is activated by a Simple decoder.

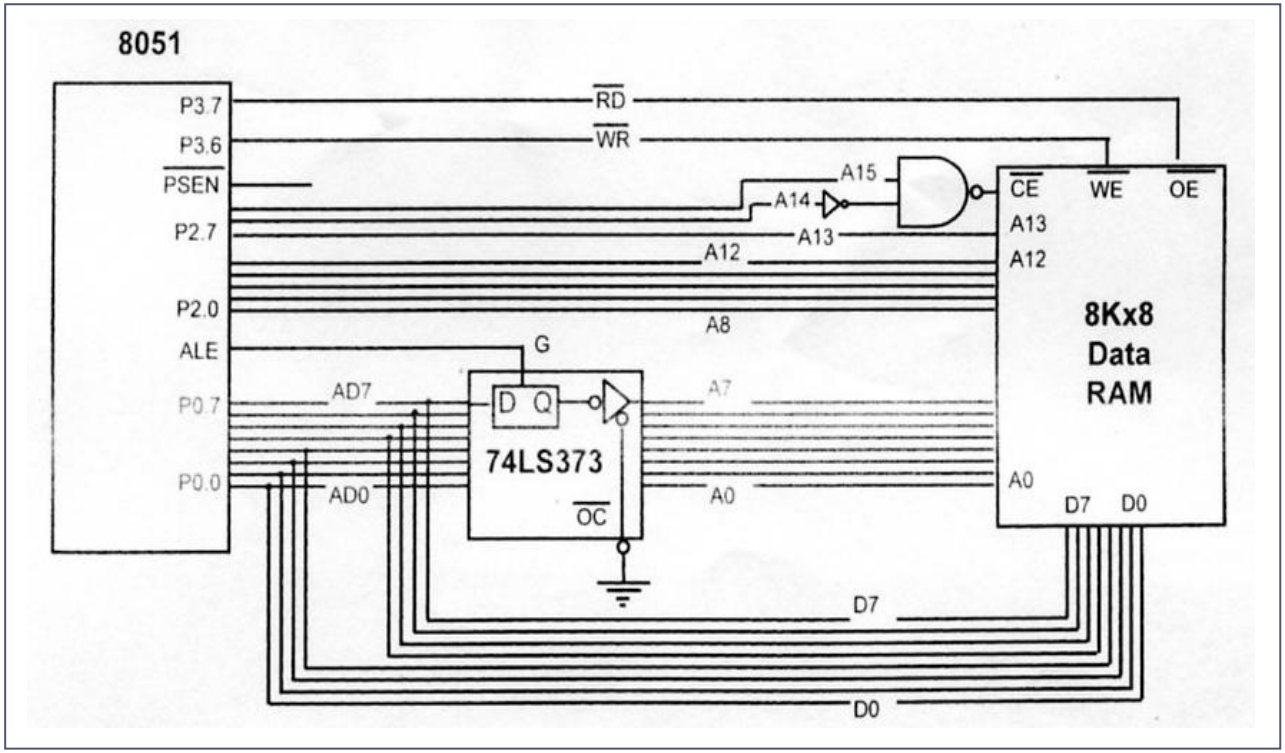


8031 Connection to External Data ROM and External Program ROM

8051 DATA MEMORY SPACE

External Data RAM

- To connect the 8051 to an external SRAM, we must use both RD (P3.7) and WR (P3.6)



8051 Connection to External Data RAM

8051 DATA MEMORY SPACE

External Data RAM (cont')

- In writing data to external data RAM, we use the instruction
  `MOVX @DPTR,A`

(a) Write a program to read 200 bytes of data from P1 and save the data in external RAM starting at RAM location 5000H.
(b) What is the address space allocated to data RAM in Figure 14-15?

Solution:
(a)

```
RAMDATA         EQU     5000H
COUNT           EQU     200

                MOV     DPTR,#RAMDATA
                MOV     R3,#COUNT
AGAIN:          MOV     A,P1
                MOVX    @DPTR,A
                ACALL   DELAY
                INC     DPTR
                DJNZ    R3,AGAIN
HERE:           SJMP    HERE
```

(b) The data address space is 8000H to BFFFH.

8051 DATA MEMORY SPACE

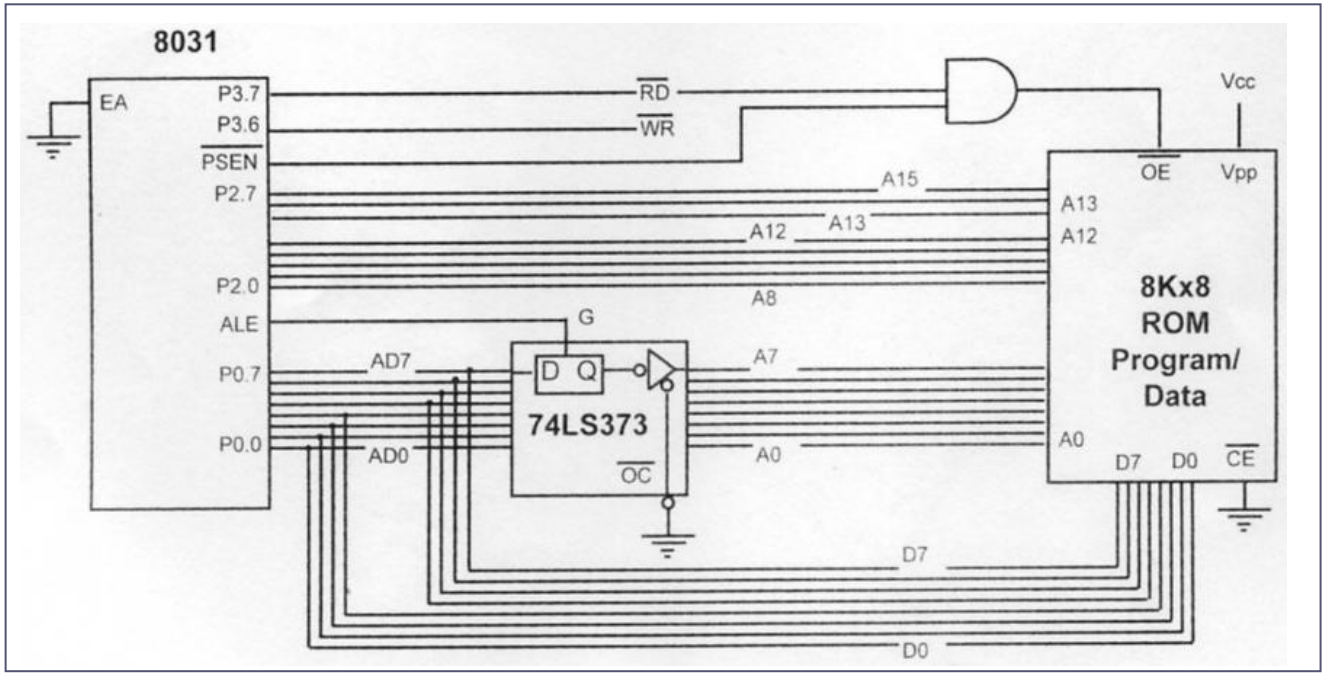
Single External ROM for Code and Data

- Assume that we have an 8031-based system connected to a single 64K×8 (27512) external ROM chip
  - The single external ROM chip is used for both program code and data storage
    - For example, the space 0000 – 7FFFH is allocated to program code, and address space 8000H – FFFFH is set aside for data
  - In accessing the data, we use the MOVX instruction

8051 DATA MEMORY SPACE

Single External ROM for Code and Data (cont')

- To allow a single ROM chip to provide both program code space and data space, we use an AND gate to signal the OE pin of the ROM chip



A Single ROM for BOTH Program and Data

8051 DATA MEMORY SPACE

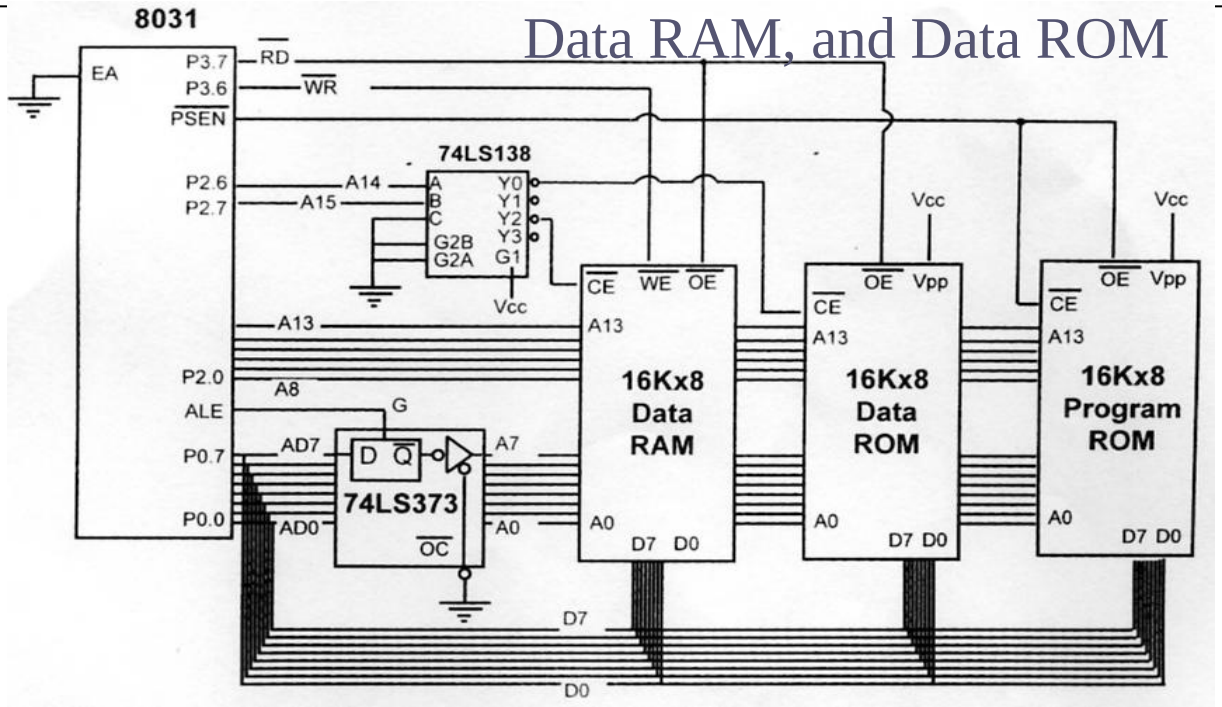
8031 System with ROM and RAM

Assume that we need an 8031 system with 16KB of program space, 16KB of data ROM starting at 0000, and 16K of NV-RAM starting at 8000H. Show the design using a 74LS138 for the address decoder.

Solution:

The solution is diagrammed in Figure 14-17. Notice that there is no need for a decoder for program ROM, but we need a 74LS138 decoder For data ROM and RAM. Also notice that G1 = V_{cc}, G2A = GND, G2B = GND, and the C input of the 74LS138 is also grounded since we Use Y0 – Y3 only.

8031 Connection to External Program ROM, Data RAM, and Data ROM



8051 DATA MEMORY SPACE

Interfacing to Large External Memory

- In some applications we need a large amount of memory to store data
  - The 8051 can support only 64K bytes of external data memory since DPTR is 16-bit
- To solve this problem, we connect A0 – A15 of the 8051 directly to the external memory's A0 – A15 pins, and use some of the P1 pins to access the 64K bytes blocks inside the single 256K×8 memory chip

8051 DATA MEMORY SPACE

Interfacing to Large External Memory (cont')

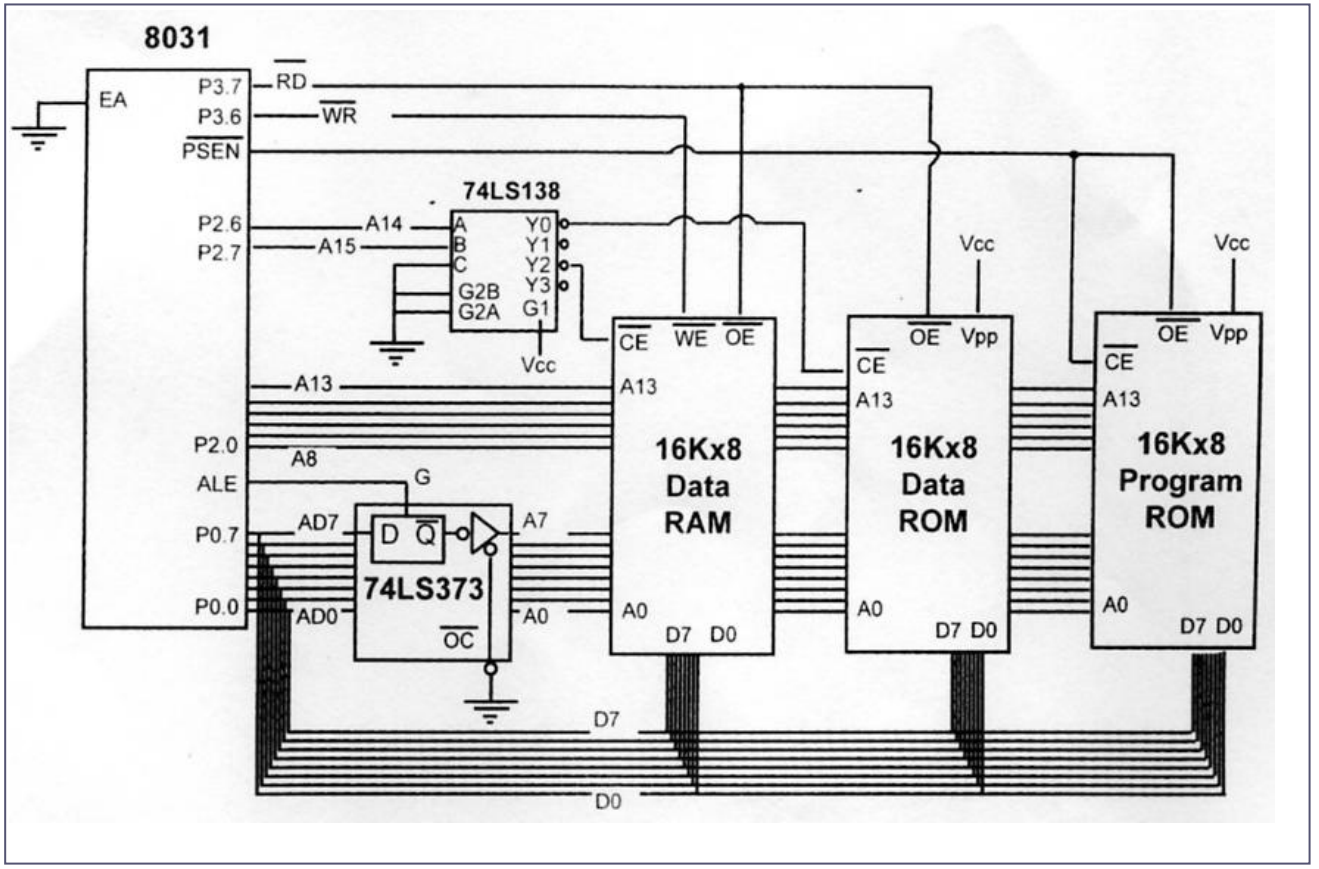


Figure 14-18. 8051 Accessing 256K*8 External NV-RAM

8051 DATA MEMORY SPACE

Interfacing to Large External Memory (cont')

In a certain application, we need 256K bytes of NV-RAM to store data collected by an 8051 microcontroller. (a) Show the connection of an 8051 to a single 256K×8 NV-RAM chip. (b) Show how various blocks of this single chip are accessed

Solution:

(a) The 256K×8 NV-RAM has 18 address pins (A0 – A17) and 8 data lines. As shown in Figure 14-18, A0 – A15 go directly to the memory chip while A16 and A17 are controlled by P1.0 and P1.1, respectively. Also notice that chip select of external RAM is connected to P1.2 of the 8051.

(b) The 256K bytes of memory are divided into four blocks, and each block is accessed as follows :

Chip select P1.2	A17 P1.1	A16 P1.0	Block address space
0	0	0	00000H - 0FFFFH
0	0	1	10000H - 1FFFFH
0	1	0	20000H - 2FFFFH
0	1	1	30000H - 3FFFFH
1	x	x	External RAM disabled

….

8051 DATA MEMORY SPACE

Interfacing to Large External Memory (cont')

....

For example, to access the 20000H – 2FFFFH address space we need the following :

```
CLR       P1.2      ;enable external RAM
MOV       DPTR,#0   ;start of 64K memory block
CLR       P1.0      ;A16 = 0
SETB      P1.1      ;A17 = 1 for 20000H block
MOV       A,SBUF    ;get data from serial port
MOVX      @DPTR,A
INC       DPTR      ;next location
...
```

REAL-WORLD INTERFACING I
LCD, ADC, AND SENSORS

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab
Dept. of Computer Science and Information Engineering
National Cheng Kung University

INTERFACING LCD TO 8051

LCD Operation

- LCD is finding widespread use replacing LEDs
  - The declining prices of LCD
  - The ability to display numbers, characters, and graphics
  - Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD
  - Ease of programming for characters and graphics

INTERFACING LCD TO 8051

LCD Pin Descriptions

Pin Descriptions for LCD

Pin	Symbol	I/O	Descriptions
1	VSS	--	Ground
2	VCC	--	+5V power supply
3	VEE	--	Power supply to control contrast
4	RS	I	RS=0 to select command register, RS=1 to select data register
5	R/W	I	R/W=0 for write, R/W=1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

used by the LCD to latch information presented to its data bus

- Send displayed information or instruction command codes to the LCD
- Read the contents of the LCD's internal registers

INTERFACING LCD TO 8051

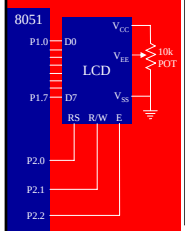
LCD Command Codes

LCD Command Codes

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning to 1st line
C0	Force cursor to beginning to 2nd line
38	2 lines and 5x7 matrix

INTERFACING LCD TO 8051

Sending Codes and Data to LCDs w/ Time Delay

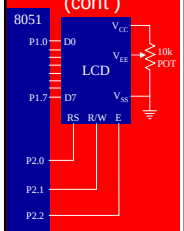


To send any of the commands to the LCD, make pin RS=0. For data, make RS=1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD. This is shown in the code below.

```
;calls a time delay before sending next data/command
;P1.0-P1.7 are connected to LCD data pins D0-D7
;P2.0 is connected to RS pin of LCD
;P2.1 is connected to R/W pin of LCD
;P2.2 is connected to E pin of LCD
        ORG
        MOV   A,#38H   ;INIT. LCD 2 LINES, 5X7 MATRIX
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#0EH   ;display on, cursor on
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#01    ;clear LCD
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#06H   ;shift cursor right
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#84H   ;cursor at line 1, pos. 4
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
```


INTERFACING LCD TO 8051

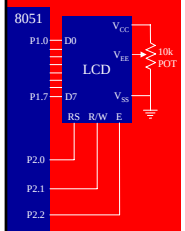
Sending Codes and Data to LCDs w/ Time Delay (cont')




```
        MOV   A,#'N'   ;display letter N
        ACALL DATAWRT  ;call display subroutine
        ACALL DELAY    ;give LCD some time
        MOV   A,#'O'    ;display letter O
        ACALL DATAWRT  ;call display subroutine
AGAIN:  SJMP  AGAIN    ;stay here
COMNWRT:               ;send command to LCD
        MOV   P1,A     ;copy reg A to port 1
        CLR   P2.0     ;RS=0 for command
        CLR   P2.1     ;R/W=0 for write
        SETB  P2.2     ;E=1 for high pulse
        CLR   P2.2     ;E=0 for H-to-L pulse
        RET
DATAWRT:               ;write data to LCD
        MOV   P1,A     ;copy reg A to port 1
        CLR   P2.0     ;RS=0 for command
        CLR   P2.1     ;R/W=0 for write
        SETB  P2.2     ;E=1 for high pulse
        CLR   P2.2     ;E=0 for H-to-L pulse
        RET
DELAY:  MOV   R3,#50   ;50 or higher for fast CPUs
HERE2:  MOV   R4,#255  ;R4 = 255
HERE:   DJNZ  R4,HERE  ;stay until R4 becomes 0
        DJNZ  R3,HERE2
        RET
        END
```


INTERFACING LCD TO 8051

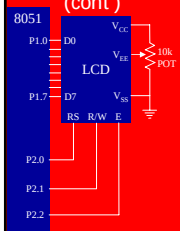
Sending Codes and Data to LCDs w/ Busy Flag




```
;Check busy flag before sending data, command to LCD
;p1=data pin
;P2.0 connected to RS pin
;P2.1 connected to R/W pin
;P2.2 connected to E pin
      ORG
      MOV   A,#38H      ;init. LCD 2 lines ,5x7 matrix
      ACALL COMMAND     ;issue command
      MOV   A,#0EH      ;LCD on, cursor on
      ACALL COMMAND     ;issue command
      MOV   A,#01H      ;clear LCD command
      ACALL COMMAND     ;issue command
      MOV   A,#06H      ;shift cursor right
      ACALL COMMAND     ;issue command
      MOV   A,#86H      ;cursor: line 1, pos. 6
      ACALL COMMAND     ;command subroutine
      MOV   A,#'N'      ;display letter N
      ACALL DATA_DISPLAY
      MOV   A,#'O'      ;display letter O
      ACALL DATA_DISPLAY
HERE:SJMP  HERE         ;STAY HERE
```


INTERFACING LCD TO 8051

Sending Codes and Data to LCDs w/ Busy Flag (cont')




```
COMMAND:
      ACALL READY       ;is LCD ready?
      MOV   P1,A        ;issue command code
      CLR   P2.0        ;RS=0 for command
      CLR   P2.1        ;R/W=0 to write to LCD
      SETB  P2.2        ;E=1 for H-to-L pulse
      CLR   P2.2        ;E=0,latch in
      RET
DATA_DISPLAY:
      ACALL READY       ;is LCD
      MOV   P1,A        ;issue
      SETB  P2.0        ;RS=1 for data
      CLR   P2.1        ;R/W =0 to write to LCD
      SETB  P2.2        ;E=1 for H-to-L pulse
      CLR   P2.2        ;E=0,latch in
      RET
READY:
      SETB  P1.7        ;make P1.7 input port
      CLR   P2.0        ;RS=0 access command reg
      SETB  P2.1        ;R/W=1 read command reg
;read command reg and check busy flag
BACK:SETB  P2.2         ;E=1 for H-to-L pulse
      CLR   P2.2        ;E=0 H-to-L pulse
      JB    P1.7,BACK   ;stay until busy flag=0
      RET
      END
```

To read the command register, we make R/W=1, RS=0, and a H-to-L pulse for the E pin.

If bit 7 (busy flag) is high, the LCD is busy and no information should be issued to it.

INTERFACING LCD TO 8051

LCD Data Sheet

- One can put data at any location in the LCD and the following shows address locations and how they are accessed

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

  - AAAAAAA=000_0000 to 010_0111 for line1
  - AAAAAAA=100_0000 to 110_0111 for line2

The upper address range can go as high as 0100111 for the 40-character-wide LCD, which corresponds to locations 0 to 39

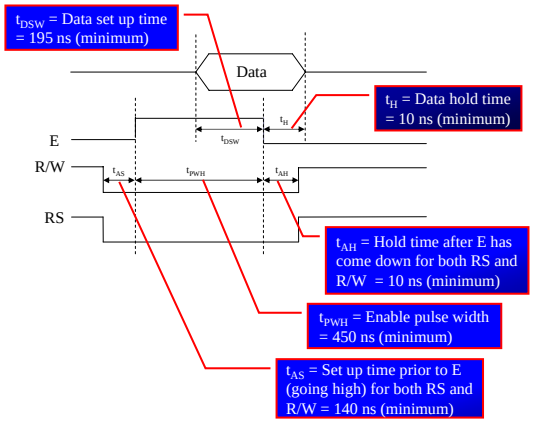
LCD Addressing for the LCDs of 40×2 size

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Line1 (min)	1	0	0	0	0	0	0	0
Line1 (max)	1	0	1	0	0	1	1	1
Line2 (min)	1	1	0	0	0	0	0	0
Line2 (max)	1	1	1	0	0	1	1	1

INTERFACING LCD TO 8051

LCD Data Sheet (cont')

LCD Timing



INTERFACING TO ADC AND SENSORS

ADC Devices

- ADCs (analog-to-digital converters) are among the most widely used devices for data acquisition
  - A physical quantity, like temperature, pressure, humidity, and velocity, etc., is converted to electrical (voltage, current) signals using a device called a *transducer*, or *sensor*
- We need an analog-to-digital converter to translate the analog signals to digital numbers, so microcontroller can read them

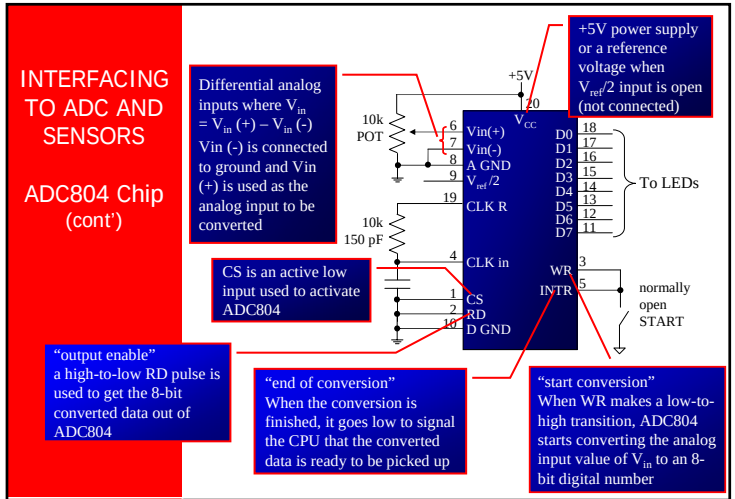
INTERFACING TO ADC AND SENSORS

ADC804 Chip

- ADC804 IC is an analog-to-digital converter
  - It works with +5 volts and has a resolution of 8 bits
  - *Conversion time* is another major factor in judging an ADC
    - Conversion time is defined as the time it takes the ADC to convert the analog input to a digital (binary) number
    - In ADC804 conversion time varies depending on the clocking signals applied to CLK R and CLK IN pins, but it cannot be faster than 110 μs

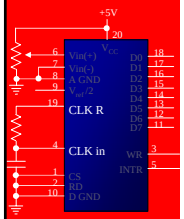
INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')



INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')



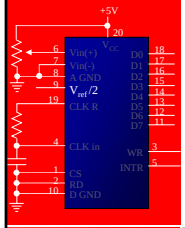
- CLK IN and CLK R
  - CLK IN is an input pin connected to an external clock source
  - To use the internal clock generator (also called self-clocking), CLK IN and CLK R pins are connected to a capacitor and a resistor, and the clock frequency is determined by

$$f = \frac{1}{1.1RC}$$

    - Typical values are R = 10K ohms and C = 150 pF
    - We get f = 606 kHz and the conversion time is 110 μs

INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')



- $V_{ref}/2$
  - It is used for the reference voltage
    - If this pin is open (not connected), the analog input voltage is in the range of 0 to 5 volts (the same as the Vcc pin)
    - If the analog input range needs to be 0 to 4 volts, $V_{ref}/2$ is connected to 2 volts

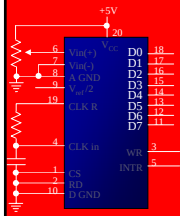
$V_{ref}/2$ Relation to V_{in} Range

$V_{ref}/2$(v)	V_{in}(V)	Step Size (mV)
Not connected*	0 to 5	5/256=19.53
2.0	0 to 4	4/255=15.62
1.5	0 to 3	3/256=11.71
1.28	0 to 2.56	2.56/256=10
1.0	0 to 2	2/256=7.81
0.5	0 to 1	1/256=3.90

Step size is the smallest change can be discerned by an ADC

INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')



- D0-D7
  - The digital data output pins
  - These are tri-state buffered
    - The converted data is accessed only when CS = 0 and RD is forced low
  - To calculate the output voltage, use the following formula

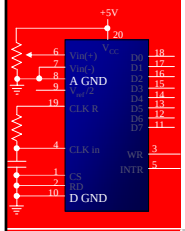
$$D_{out} = \frac{V_{in}}{step\ size}$$

    - *Dout* = digital data output (in decimal),
    - *Vin* = analog voltage, and
    - *step size* (resolution) is the smallest change

INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')

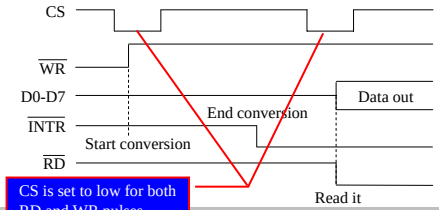
- Analog ground and digital ground
  - Analog ground is connected to the ground of the analog V_{in}
  - Digital ground is connected to the ground of the V_{cc} pin
- To isolate the analog V_{in} signal from transient voltages caused by digital switching of the output D0 – D7
  - This contributes to the accuracy of the digital data output



INTERFACING TO ADC AND SENSORS

ADC804 Chip (cont')

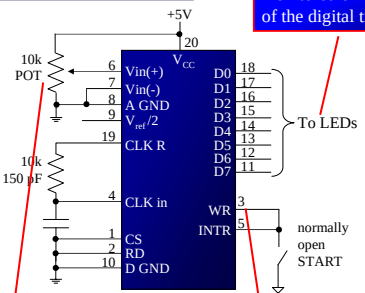
- The following steps must be followed for data conversion by the ADC804 chip
  - Make CS = 0 and send a low-to-high pulse to pin WR to start conversion
  - Keep monitoring the INTR pin
    - If INTR is low, the conversion is finished
    - If the INTR is high, keep polling until it goes low
  - After the INTR has become low, we make CS = 0 and send a high-to-low pulse to the RD pin to get the data out of the ADC804



INTERFACING TO ADC AND SENSORS

Testing ADC804

ADC804 Free Running Test Mode



The binary outputs are monitored on the LED of the digital trainer

a potentiometer used to apply a 0-to-5 V analog voltage to input Vin (+) of the 804 ADC

The CS input is grounded and the WR input is connected to the INTR output

INTERFACING TO ADC AND SENSORS

Testing ADC804 (cont')

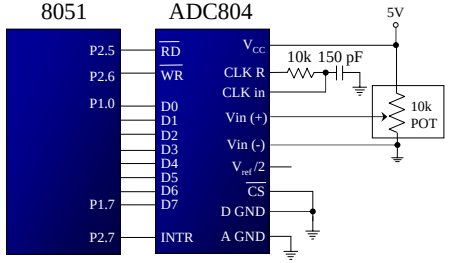
Examine the ADC804 connection to the 8051 in Figure 12-7. Write a program to monitor the INTR pin and bring an analog input into register A. Then call a hex-to ACSII conversion and data display subroutines. Do this continuously.

```
;p2.6=WR (start conversion needs to L-to-H pulse)
;p2.7 When low, end-of-conversion)
;p2.5=RD (a H-to-L will read the data from ADC chip)
;p1.0 – P1.7= D0 - D7 of the ADC804
;
       MOV   P1,#0FFH     ;make P1 = input
BACK:  CLR   P2.6         ;WR = 0
       SETB  P2.6         ;WR = 1 L-to-H to start conversion
HERE:  JB    P2.7,HERE    ;wait for end of conversion
       CLR   P2.5         ;conversion finished, enable RD
       MOV   A,P1         ;read the data
       ACALL CONVERSION   ;hex-to-ASCII conversion
       ACALL DATA_DISPLAY;display the data
       SETB  p2.5         ;make RD=1 for next round
       SJMP  BACK
```

INTERFACING TO ADC AND SENSORS

Testing ADC804 (cont')

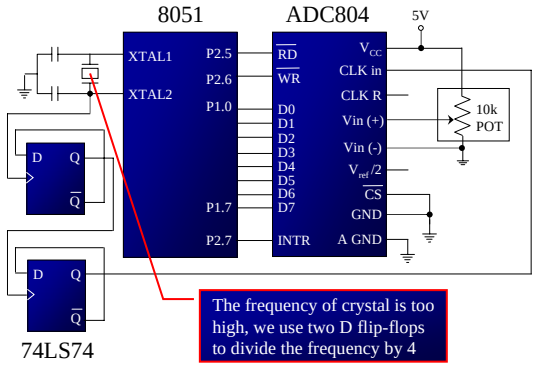
8051 Connection to ADC804 with Self-Clocking



INTERFACING TO ADC AND SENSORS

ADC804 Clock from 8051 XTAL2

8051 Connection to ADC804 with Clock from XTAL2 of 8051



INTERFACING TO ADC AND SENSORS

Interfacing Temperature Sensor

- A *thermistor* responds to temperature change by changing resistance, but its response is not linear
- The complexity associated with writing software for such nonlinear devices has led many manufacturers to market the linear temperature sensor

Temperature (C)	Tf (K ohms)
0	29.490
25	10.000
50	3.893
75	1.700
100	0.817

From William Kleitz, digital Electronics

INTERFACING TO ADC AND SENSORS

LM34 and LM35 Temperature Sensors

- The sensors of the LM34/LM35 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit/Celsius temperature
  - The LM34/LM35 requires no external calibration since it is inherently calibrated
  - It outputs 10 mV for each degree of Fahrenheit/Celsius temperature

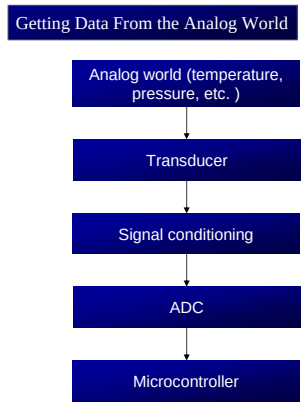
INTERFACING TO ADC AND SENSORS

Signal Conditioning and Interfacing LM35

- *Signal conditioning* is a widely used term in the world of data acquisition
  - It is the conversion of the signals (voltage, current, charge, capacitance, and resistance) produced by transducers to voltage, which is sent to the input of an A-to-D converter
- Signal conditioning can be a current-to-voltage conversion or a signal amplification
  - The thermistor changes resistance with temperature, while the change of resistance must be translated into voltage in order to be of any use to an ADC

INTERFACING TO ADC AND SENSORS

Signal Conditioning and Interfacing LM35 (cont')



INTERFACING TO ADC AND SENSORS

Signal Conditioning and Interfacing LM35 (cont')

Example:

Look at the case of connecting an LM35 to an ADC804. Since the ADC804 has 8-bit resolution with a maximum of 256 steps and the LM35 (or LM34) produces 10 mV for every degree of temperature change, we can condition Vin of the ADC804 to produce a Vout of 2560 mV full-scale output. Therefore, in order to produce the full-scale Vout of 2.56 V for the ADC804, We need to set Vref/2 = 1.28. This makes Vout of the ADC804 correspond directly to the temperature as monitored by the LM35.

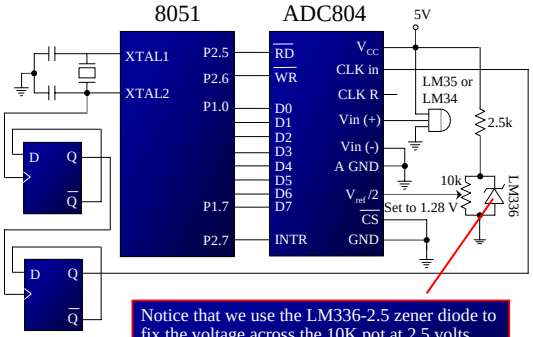
Temperature vs. V_{out} of the ADC804

Temp. (C)	Vin (mV)	Vout (D7 – D0)
0	0	0000 0000
1	10	0000 0001
2	20	0000 0010
3	30	0000 0011
10	100	0000 1010
30	300	0001 1110

INTERFACING TO ADC AND SENSORS

Signal Conditioning and Interfacing LM35 (cont')

8051 Connection to ADC804 and Temperature Sensor



Notice that we use the LM336-2.5 zener diode to fix the voltage across the 10K pot at 2.5 volts. The use of the LM336-2.5 should overcome any fluctuations in the power supply

INTERFACING TO ADC AND SENSORS

ADC808/809 Chip

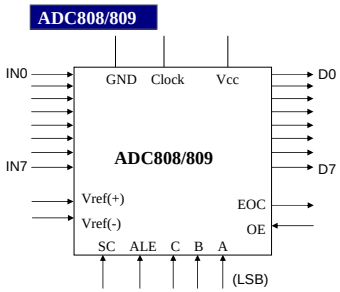
- ADC808 has 8 analog inputs
  - It allows us to monitor up to 8 different transducers using only a single chip
  - The chip has 8-bit data output just like the ADC804
  - The 8 analog input channels are multiplexed and selected according to table below using three address pins, A, B, and C

ADC808 Analog Channel Selection

Selected Analog Channel	C	B	A
IN0	0	0	0
IN1	0	0	1
IN2	0	1	0
IN3	0	1	1
IN4	1	0	0
IN5	1	0	1
IN6	1	1	0
IN7	1	1	1

INTERFACING TO ADC AND SENSORS

ADC808/809 Chip (cont')



INTERFACING TO ADC AND SENSORS

Steps to Program ADC808/809

1. Select an analog channel by providing bits to A, B, and C addresses
2. Activate the ALE pin
   - It needs an L-to-H pulse to latch in the address
3. Activate SC (start conversion) by an H-to-L pulse to initiate conversion
4. Monitor EOC (end of conversion) to see whether conversion is finished
5. Activate OE (output enable) to read data out of the ADC chip
   - An H-to-L pulse to the OE pin will bring digital data out of the chip

LCD AND KEYBOARD INTERFACING

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

- LCD is finding widespread use replacing LEDs
  - The declining prices of LCD
  - The ability to display numbers, characters, and graphics
  - Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD
  - Ease of programming for characters and graphics

LCD INTERFACING

LCD Pin Descriptions

Pin Descriptions for LCD

Pin	Symbol	I/O	Descriptions
1	VSS	--	Ground
2	VCC	--	+5V power supply
3	VEE	--	Power supply to control contrast
4	RS	I	RS=0 to select command register, RS=1 to select data register
5	R/W	I	R/W=0 for write, R/W=1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

used by the LCD to latch information presented to its data bus

- Send displayed information or instruction command codes to the LCD
- Read the contents of the LCD's internal registers

LCD INTERFACING

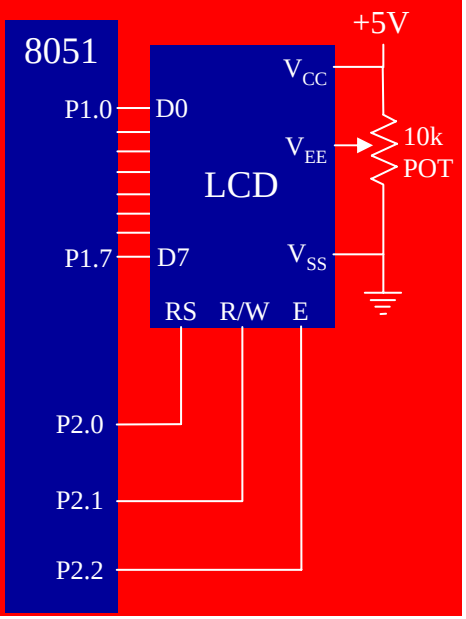
LCD Command Codes

LCD Command Codes

Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning to 1st line
C0	Force cursor to beginning to 2nd line
38	2 lines and 5x7 matrix

LCD INTERFACING

Sending Data/ Commands to LCDs w/ Time Delay

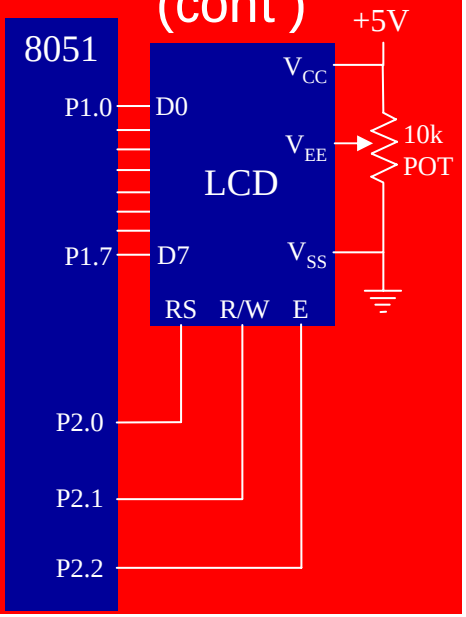


To send any of the commands to the LCD, make pin RS=0. For data, make RS=1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD. This is shown in the code below.

```
;calls a time delay before sending next data/command
;P1.0-P1.7 are connected to LCD data pins D0-D7
;P2.0 is connected to RS pin of LCD
;P2.1 is connected to R/W pin of LCD
;P2.2 is connected to E pin of LCD
        ORG   0H
        MOV   A,#38H  ;INIT. LCD 2 LINES, 5X7 MATRIX
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#0EH  ;display on, cursor on
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#01   ;clear LCD
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#06H  ;shift cursor right
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
        MOV   A,#84H  ;cursor at line 1, pos. 4
        ACALL COMNWRT ;call command subroutine
        ACALL DELAY   ;give LCD some time
.....
```

LCD INTERFACING

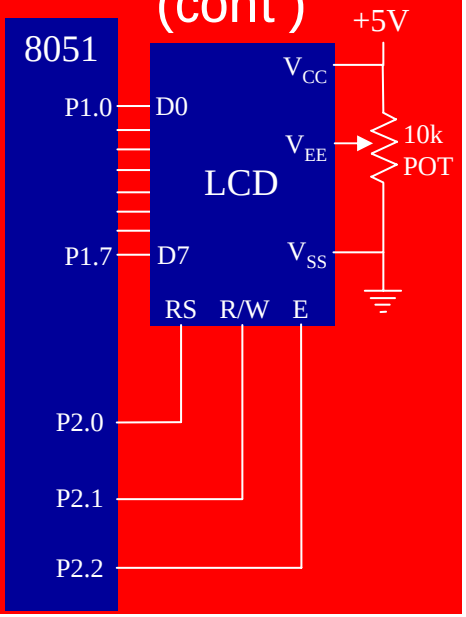
Sending Data/ Commands to LCDs w/ Time Delay (cont')




```
.....
          MOV   A,#'N'    ;display letter N
          ACALL DATAWRT   ;call display subroutine
          ACALL DELAY     ;give LCD some time
          MOV   A,#'O'     ;display letter O
          ACALL DATAWRT   ;call display subroutine
AGAIN:    SJMP  AGAIN     ;stay here
COMNWRT:                  ;send command to LCD
          MOV   P1,A      ;copy reg A to port 1
          CLR   P2.0      ;RS=0 for command
          CLR   P2.1      ;R/W=0 for write
          SETB  P2.2      ;E=1 for high pulse
          ACALL DELAY     ;give LCD some time
          CLR   P2.2      ;E=0 for H-to-L pulse
          RET
DATAWRT:                  ;write data to LCD
          MOV   P1,A      ;copy reg A to port 1
          SETB  P2.0      ;RS=1 for data
          CLR   P2.1      ;R/W=0 for write
          SETB  P2.2      ;E=1 for high pulse
          ACALL DELAY     ;give LCD some time
          CLR   P2.2      ;E=0 for H-to-L pulse
          RET
DELAY:    MOV   R3,#50    ;50 or higher for fast CPUs
HERE2:    MOV   R4,#255   ;R4 = 255
HERE:     DJNZ  R4,HERE   ;stay until R4 becomes 0
          DJNZ  R3,HERE2
          RET
          END
```

LCD INTERFACING

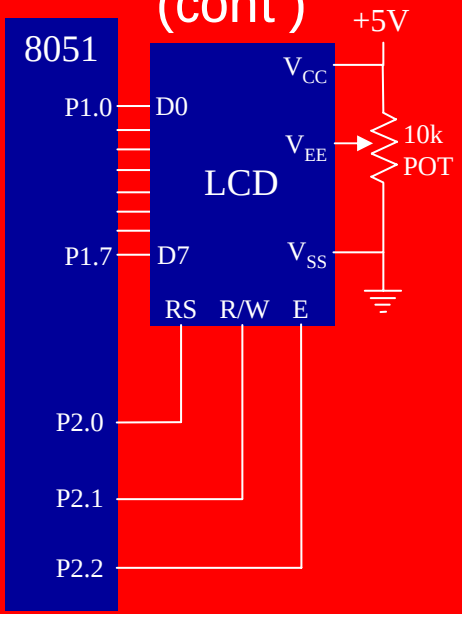
Sending Data/ Commands to LCDs w/ Time Delay (cont')




```
;Check busy flag before sending data, command to LCD
;p1=data pin
;P2.0 connected to RS pin
;P2.1 connected to R/W pin
;P2.2 connected to E pin
      ORG   0H
      MOV   A,#38H        ;init. LCD 2 lines ,5x7 matrix
      ACALL COMMAND       ;issue command
      MOV   A,#0EH        ;LCD on, cursor on
      ACALL COMMAND       ;issue command
      MOV   A,#01H        ;clear LCD command
      ACALL COMMAND       ;issue command
      MOV   A,#06H        ;shift cursor right
      ACALL COMMAND       ;issue command
      MOV   A,#86H        ;cursor: line 1, pos. 6
      ACALL COMMAND       ;command subroutine
      MOV   A,#'N'        ;display letter N
      ACALL DATA_DISPLAY
      MOV   A,#'O'        ;display letter O
      ACALL DATA_DISPLAY
HERE:SJMP  HERE           ;STAY HERE
.....
```

LCD INTERFACING

Sending Codes and Data to LCDs w/ Busy Flag (cont')




```
.....
COMMAND:
      ACALL READY          ;is LCD ready?
      MOV   P1,A           ;issue command code
      CLR   P2.0           ;RS=0 for command
      CLR   P2.1           ;R/W=0 to write to LCD
      SETB  P2.2           ;E=1 for H-to-L pulse
      CLR   P2.2           ;E=0,latch in
      RET
DATA_DISPLAY:
      ACALL READY          ;is LCD ready?
      MOV   P1,A           ;issue data
      SETB  P2.0           ;RS=1 for data
      CLR   P2.1           ;R/W =0 to write to LCD
      SETB  P2.2           ;E=1 for H-to-L pulse
      CLR   P2.2           ;E=0 latch in
      RET
READY:
      SETB  P1.7           ;make P1.7 input port
      CLR   P2.0           ;RS=0 access command reg
      SETB  P2.1           ;R/W=1 read command reg
;read command reg and check busy flag
BACK:SETB  P2.2            ;E=1 for H-to-L pulse
      CLR   P2.2           ;E=0 H-to-L pulse
      JB    P1.7,BACK      ;stay until busy flag=0
      RET
      END
```

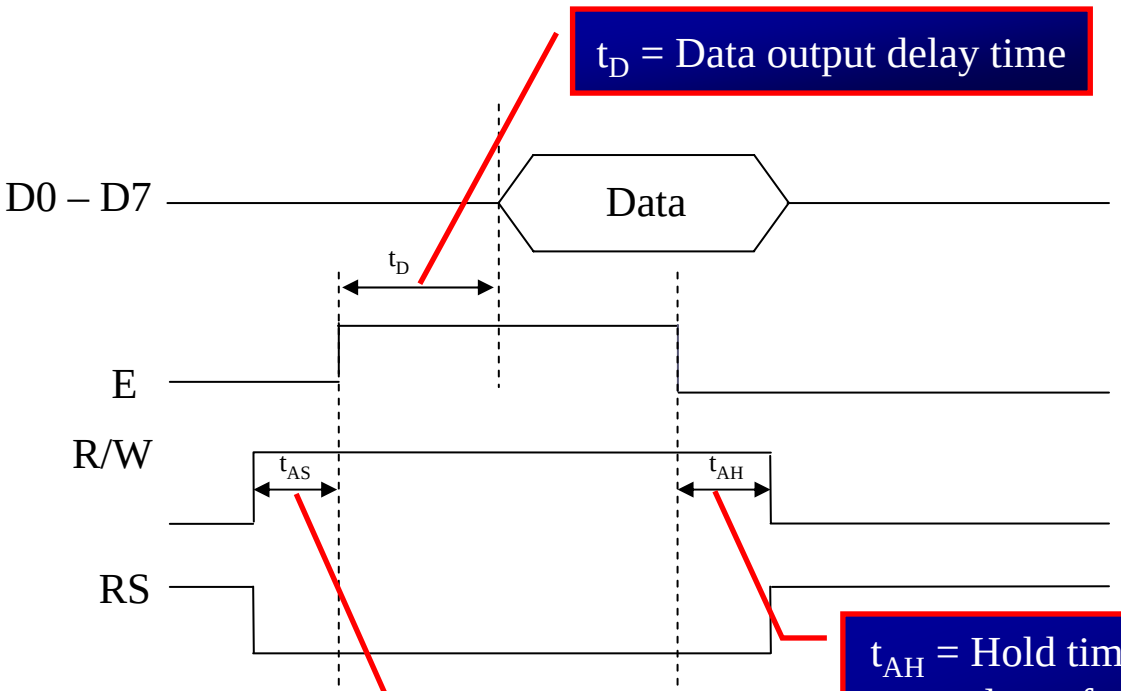
To read the command register, we make R/W=1, RS=0, and a H-to-L pulse for the E pin.

If bit 7 (busy flag) is high, the LCD is busy and no information should be issued to it.

LCD INTERFACING

Sending Codes and Data to LCDs w/ Busy Flag (cont')

LCD Timing for Read



t_D = Data output delay time

t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

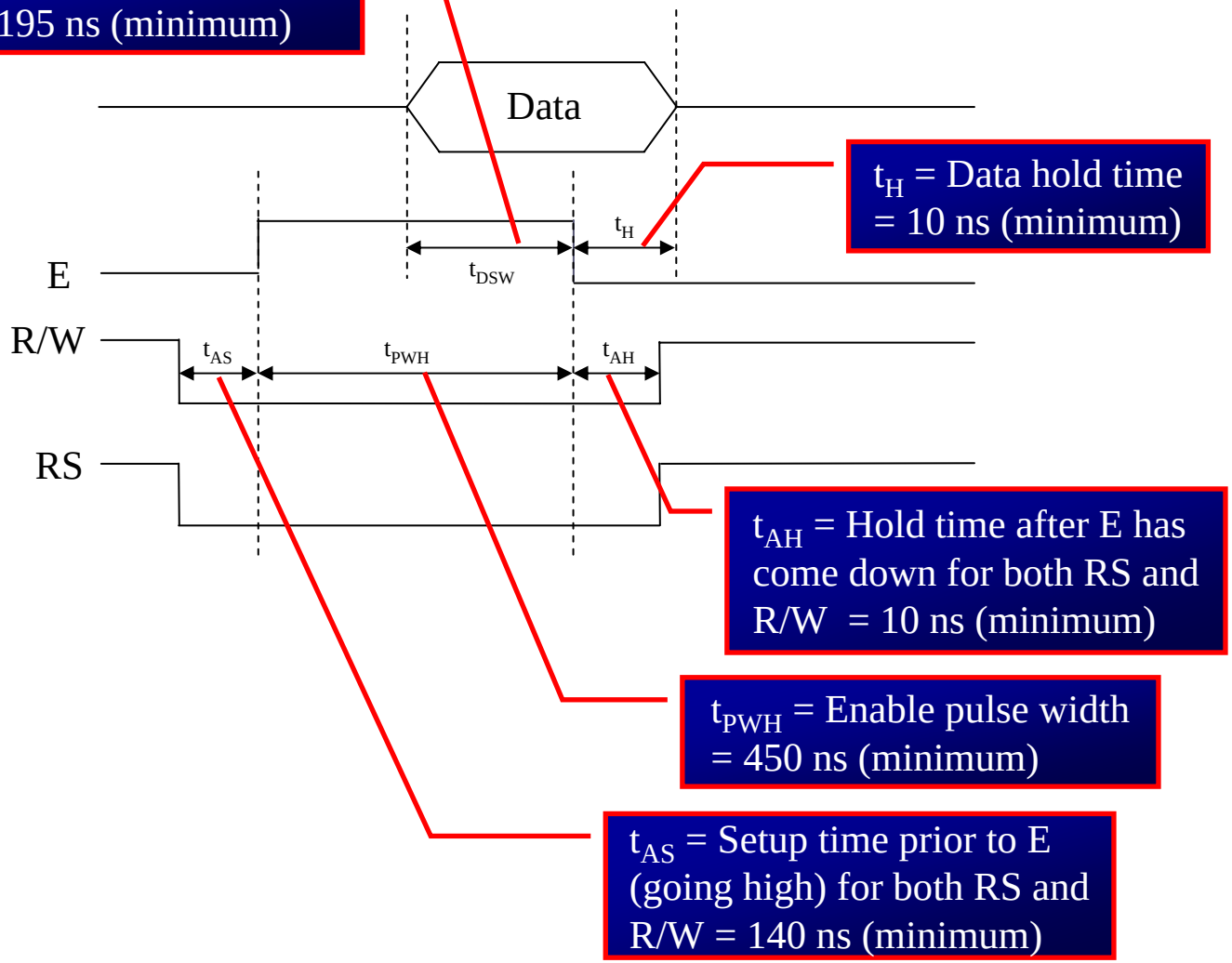
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 140 ns (minimum)

Note : Read requires an L-to-H pulse for the E pin

LCD INTERFACING

Sending Codes and Data to LCDs w/ Busy Flag (cont')

LCD Timing for Write



- One can put data at any location in the LCD and the following shows address locations and how they are accessed

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

  - AAAAAAA=000_0000 to 010_0111 for line1
  - AAAAAAA=100_0000 to 110_0111 for line2

The upper address range can go as high as 0100111 for the 40-character-wide LCD, which corresponds to locations 0 to 39

LCD Addressing for the LCDs of 40×2 size

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Line1 (min)	1	0	0	0	0	0	0	0
Line1 (max)	1	0	1	0	0	1	1	1
Line2 (min)	1	1	0	0	0	0	0	0
Line2 (max)	1	1	1	0	0	1	1	1

LCD INTERFACING

Sending Information to LCD Using MOVC Instruction

```
;Call a time delay before sending next data/command
; P1.0-P1.7=D0-D7, P2.0=RS, P2.1=R/W, P2.2=E

        ORG   0
        MOV   DPTR,#MYCOM
C1:     CLR   A
        MOVC  A,@A+DPTR
        ACALL COMNWRT  ;call command subroutine
        ACALL DELAY    ;give LCD some time
        INC   DPTR
        JZ    SEND_DAT
        SJMP  C1
SEND_DAT:
        MOV   DPTR,#MYDATA
D1:     CLR   A
        MOVC  A,@A+DPTR
        ACALL DATAWRT  ;call command subroutine
        ACALL DELAY    ;give LCD some time
        INC   DPTR
        JZ    AGAIN
        SJMP  D1
AGAIN:  SJMP  AGAIN    ;stay here
.....
```

LCD INTERFACING

Sending Information to LCD Using MOVC Instruction (cont')

```
.....
COMNWRT:                ;send command to LCD
        MOV   P1,A      ;copy reg A to P1
        CLR   P2.0      ;RS=0 for command
        CLR   P2.1      ;R/W=0 for write
        SETB  P2.2      ;E=1 for high pulse
        ACALL DELAY     ;give LCD some time
        CLR   P2.2      ;E=0 for H-to-L pulse
        RET
DATAWRT:                ;write data to LCD
        MOV   P1,A      ;copy reg A to port 1
        SETB  P2.0      ;RS=1 for data
        CLR   P2.1      ;R/W=0 for write
        SETB  P2.2      ;E=1 for high pulse
        ACALL DELAY     ;give LCD some time
        CLR   P2.2      ;E=0 for H-to-L pulse
        RET
DELAY:  MOV   R3,#250   ;50 or higher for fast CPUs
HERE2:  MOV   R4,#255   ;R4 = 255
HERE:   DJNZ  R4,HERE   ;stay until R4 becomes 0
        DJNZ  R3,HERE2
        RET
        ORG   300H
MYCOM:  DB    38H,0EH,01,06,84H,0 ; commands and null
MYDATA: DB    "HELLO",0
        END
```

LCD INTERFACING

Sending Information to LCD Using MOVC Instruction (cont')

Example 12-2

Write an 8051 C program to send letters 'M', 'D', and 'E' to the LCD using the busy flag method.

Solution:

```
#include <reg51.h>
sfr ldata = 0x90; //P1=LCD data pins
sbit rs = P2^0;
sbit rw = P2^1;
sbit en = P2^2;
sbit busy = P1^7;
void main(){
  lcdcmd(0x38);
  lcdcmd(0x0E);
  lcdcmd(0x01);
  lcdcmd(0x06);
  lcdcmd(0x86);   //line 1, position 6
  lcdcmd('M');
  lcdcmd('D');
  lcdcmd('E');
}
.....
```

LCD INTERFACING

Sending Information to LCD Using MOVC Instruction (cont')

```
.....
void lcdcmd(unsigned char value){
  lcdready();       //check the LCD busy flag
  ldata = value;    //put the value on the pins
  rs = 0;
  rw = 0;
  en = 1;           //strobe the enable pin
  MSDelay(1);
  en = 0;
  return;
}

void lcddata(unsigned char value){
  lcdready();       //check the LCD busy flag
  ldata = value;    //put the value on the pins
  rs = 1;
  rw = 0;
  en = 1;           //strobe the enable pin
  MSDelay(1);
  en = 0;
  return;
}
.....
```

LCD INTERFACING

Sending Information to LCD Using MOVC Instruction (cont')

```
.....
void lcdready(){
  busy = 1;          //make the busy pin at input
  rs = 0;
  rw = 1;
  while(busy==1){    //wait here for busy flag
    en = 0;          //strobe the enable pin
    MSDelay(1);
    en = 1;
}

void lcddata(unsigned int itime){
  unsigned int i, j;
  for(i=0;i<itime;i++)
    for(j=0;j<1275;j++);
}
```

KEYBOARD INTERFACING

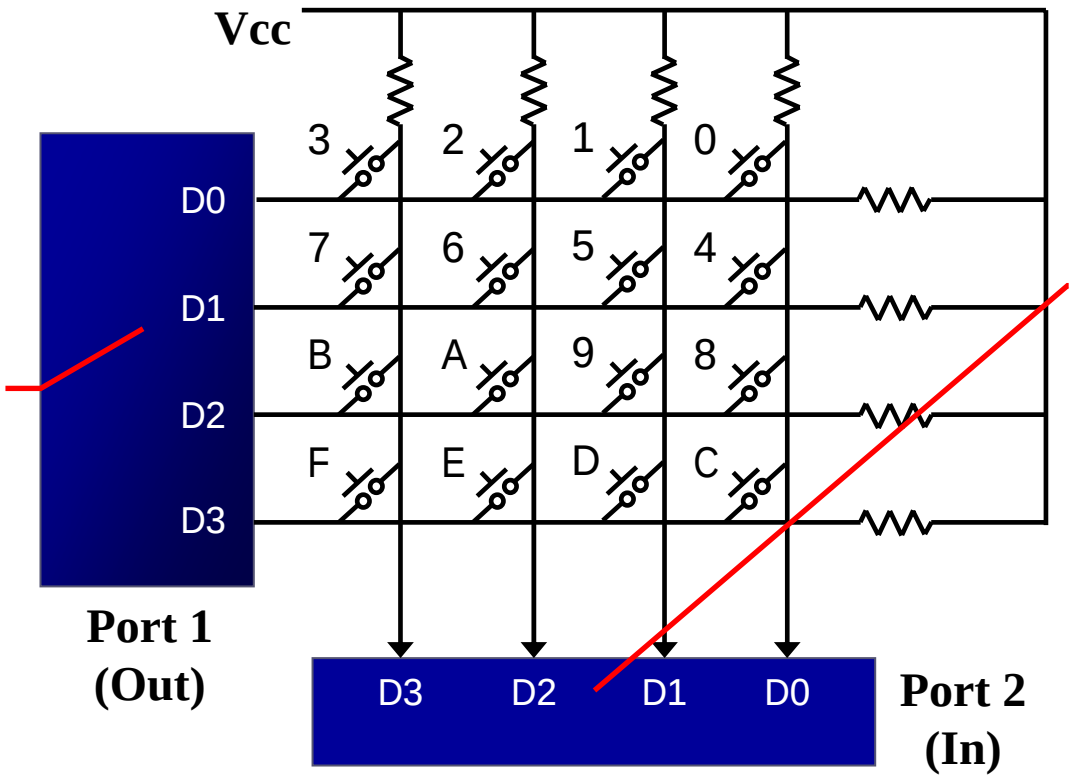
- Keyboards are organized in a matrix of rows and columns
  - The CPU accesses both rows and columns through ports
    - Therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor
  - When a key is pressed, a row and a column make a contact
    - Otherwise, there is no connection between rows and columns
- In IBM PC keyboards, a single microcontroller takes care of hardware and software interfacing

KEYBOARD INTERFACING

Scanning and Identifying the Key

- A 4x4 matrix connected to two ports
  - The rows are connected to an output port and the columns are connected to an input port

Matrix Keyboard Connection to ports



If all the rows are grounded and a key is pressed, one of the columns will have 0 since the key pressed provides the path to ground

If no key has been pressed, reading the input port will yield 1s for all columns since they are all connected to high (V_{cc})

KEYBOARD INTERFACING

Grounding Rows and Reading Columns

- It is the function of the microcontroller to scan the keyboard continuously to detect and identify the key pressed
- To detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, then it reads the columns
  - If the data read from columns is D3 – D0 = 1111, no key has been pressed and the process continues till key press is detected
  - If one of the column bits has a zero, this means that a key press has occurred
    - For example, if D3 – D0 = 1101, this means that a key in the D1 column has been pressed
    - After detecting a key press, microcontroller will go through the process of identifying the key

KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

- Starting with the top row, the microcontroller grounds it by providing a low to row D0 only
  - It reads the columns, if the data read is all 1s, no key in that row is activated and the process is moved to the next row
- It grounds the next row, reads the columns, and checks for any zero
  - This process continues until the row is identified
- After identification of the row in which the key has been pressed
  - Find out which column the pressed key belongs to

KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

Example 12-3

From Figure 12-6, identify the row and column of the pressed key for each of the following.

(a) D3 – D0 = 1110 for the row, D3 – D0 = 1011 for the column

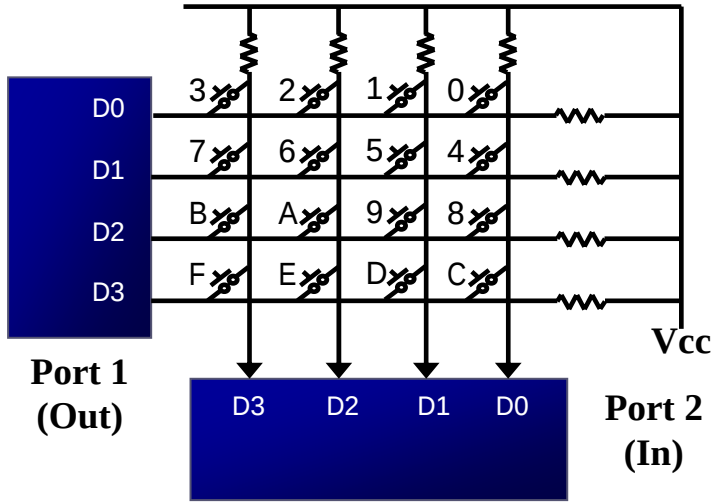
(b) D3 – D0 = 1101 for the row, D3 – D0 = 0111 for the column

Solution :

From Figure 13-5 the row and column can be used to identify the key.

(a) The row belongs to D0 and the column belongs to D2; therefore, key number 2 was pressed.

(b) The row belongs to D1 and the column belongs to D3; therefore, key number 7 was pressed.



- Program 12-4 for detection and identification of key activation goes through the following stages:
  1. To make sure that the preceding key has been released, 0s are output to all rows at once, and the columns are read and checked repeatedly until all the columns are high
     - When all columns are found to be high, the program waits for a short amount of time before it goes to the next stage of waiting for a key to be pressed

KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

2. To see if any key is pressed, the columns are scanned over and over in an infinite loop until one of them has a 0 on it
   - Remember that the output latches connected to rows still have their initial zeros (provided in stage 1), making them grounded
   - After the key press detection, it waits 20 ms for the bounce and then scans the columns again
     (a) it ensures that the first key press detection was not an erroneous one due a spike noise
     (b) the key press. If after the 20-ms delay the key is still pressed, it goes back into the loop to detect a real key press

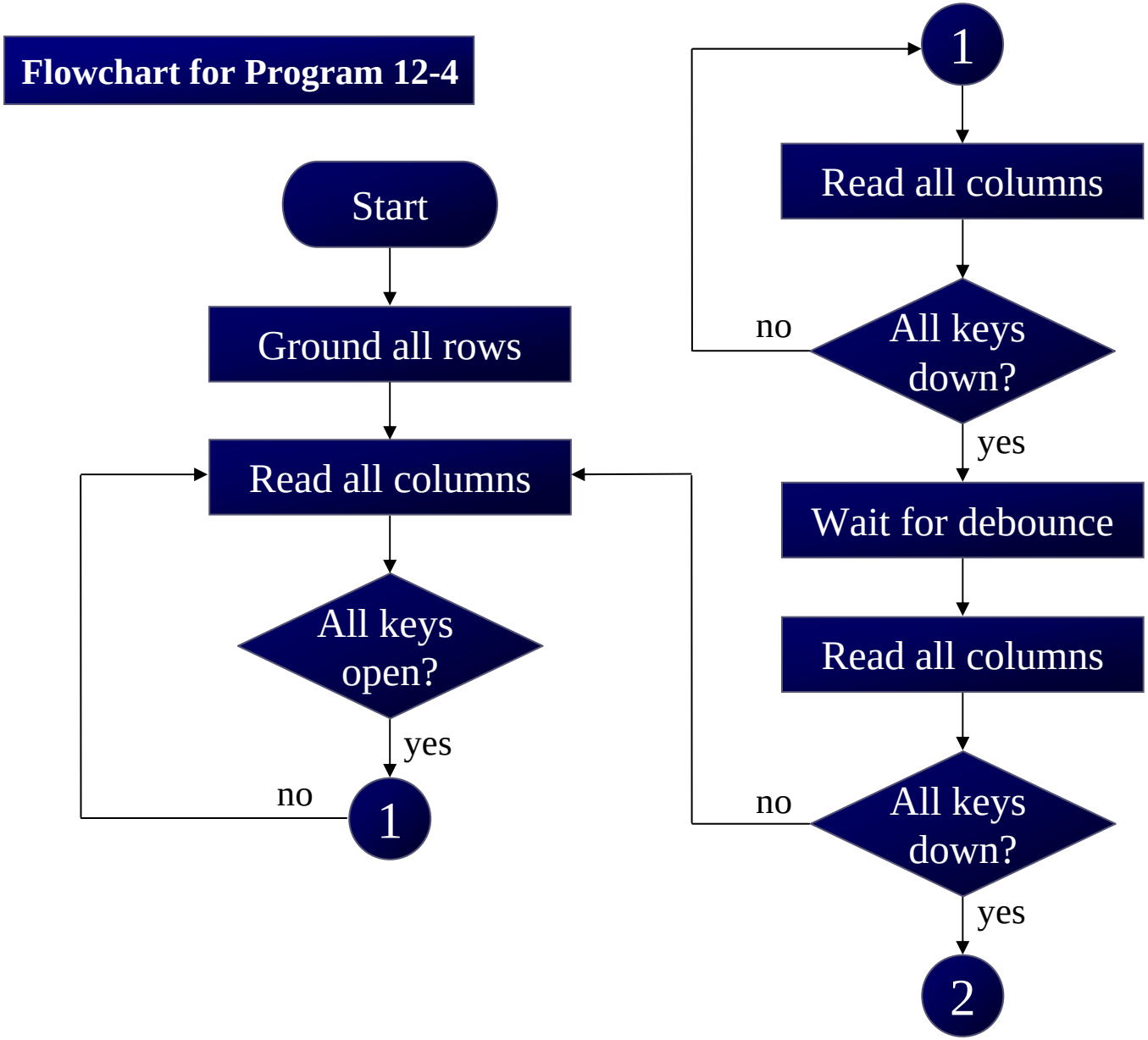
KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

3. To detect which row key press belongs to, it grounds one row at a time, reading the columns each time
   - If it finds that all columns are high, this means that the key press cannot belong to that row
     - Therefore, it grounds the next row and continues until it finds the row the key press belongs to
   - Upon finding the row that the key press belongs to, it sets up the starting address for the look-up table holding the scan codes (or ASCII) for that row
4. To identify the key press, it rotates the column bits, one bit at a time, into the carry flag and checks to see if it is low
   - Upon finding the zero, it pulls out the ASCII code for that key from the look-up table
   - otherwise, it increments the pointer to point to the next element of the look-up table

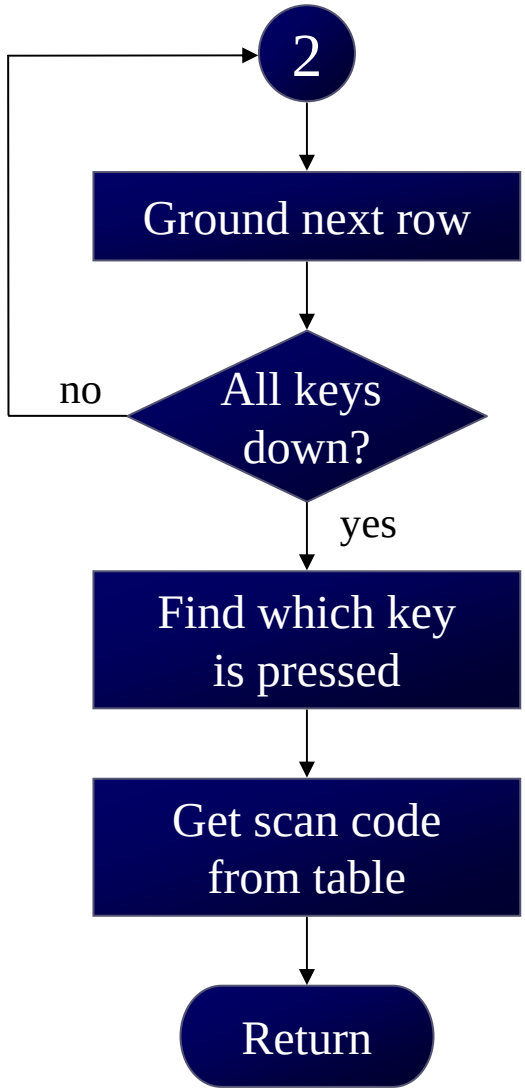
KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')



KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')



KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

Program 12-4: Keyboard Program

```
;keyboard subroutine. This program sends the ASCII
;code for pressed key to P0.1
;P1.0-P1.3 connected to rows, P2.0-P2.3 to column

        MOV  P2,#0FFH  ;make P2 an input port
K1:     MOV  P1,#0     ;ground all rows at once
        MOV  A,P2      ;read all col
                       ;(ensure keys open)
        ANL  A,00001111B     ;masked unused bits
        CJNE A,#00001111B,K1 ;till all keys release
K2:     ACALL DELAY    ;call 20 msec delay
        MOV  A,P2      ;see if any key is pressed
        ANL  A,00001111B     ;mask unused bits
        CJNE A,#00001111B,OVER;key pressed, find row
        SJMP K2        ;check till key pressed
OVER:   ACALL DELAY    ;wait 20 msec debounce time
        MOV  A,P2      ;check key closure
        ANL  A,00001111B     ;mask unused bits
        CJNE A,#00001111B,OVER1;key pressed, find row
        SJMP K2        ;if none, keep polling

....
```

KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

```
....
OVER1:  MOV P1, #11111110B  ;ground row 0
        MOV A,P2                    ;read all columns
        ANL A,#00001111B            ;mask unused bits
       CJNE A,#00001111B,ROW_0  ;key row 0, find col.
        MOV P1,#11111101B           ;ground row 1
        MOV A,P2                    ;read all columns
        ANL A,#00001111B            ;mask unused bits
       CJNE A,#00001111B,ROW_1  ;key row 1, find col.
        MOV P1,#11111011B           ;ground row 2
        MOV A,P2                    ;read all columns
        ANL A,#00001111B            ;mask unused bits
       CJNE A,#00001111B,ROW_2  ;key row 2, find col.
        MOV P1,#11110111B           ;ground row 3
        MOV A,P2                    ;read all columns
        ANL A,#00001111B            ;mask unused bits
       CJNE A,#00001111B,ROW_3  ;key row 3, find col.
        LJMP K2                 ;if none, false input,
                                ;repeat
....
```

KEYBOARD INTERFACING

Grounding Rows and Reading Columns (cont')

```
....
ROW_0:  MOV  DPTR,#KCODE0   ;set DPTR=start of row 0
        SJMP FIND           ;find col. Key belongs to
ROW_1:  MOV  DPTR,#KCODE1   ;set DPTR=start of row
        SJMP FIND           ;find col. Key belongs to
ROW_2:  MOV  DPTR,#KCODE2   ;set DPTR=start of row 2
        SJMP FIND           ;find col. Key belongs to
ROW_3:  MOV  DPTR,#KCODE3   ;set DPTR=start of row 3
FIND:   RRC  A              ;see if any CY bit low
        JNC  MATCH          ;if zero, get ASCII code
        INC  DPTR           ;point to next col. addr
        SJMP FIND           ;keep searching
MATCH:  CLR  A              ;set A=0 (match is found)
        MOVC A,@A+DPTR      ;get ASCII from table
        MOV  P0,A           ;display pressed key
        LJMP K1
;ASCII LOOK-UP TABLE FOR EACH ROW
        ORG  300H
KCODE0: DB   '0','1','2','3' ;ROW 0
KCODE1: DB   '4','5','6','7' ;ROW 1
KCODE2: DB   '8','9','A','B' ;ROW 2
KCODE3: DB   'C','D','E','F' ;ROW 3
        END
```

8031/51 INTERFACING WITH THE 8255

The 8051 Microcontroller and Embedded Systems: Using Assembly and C
Mazidi, Mazidi and McKinlay

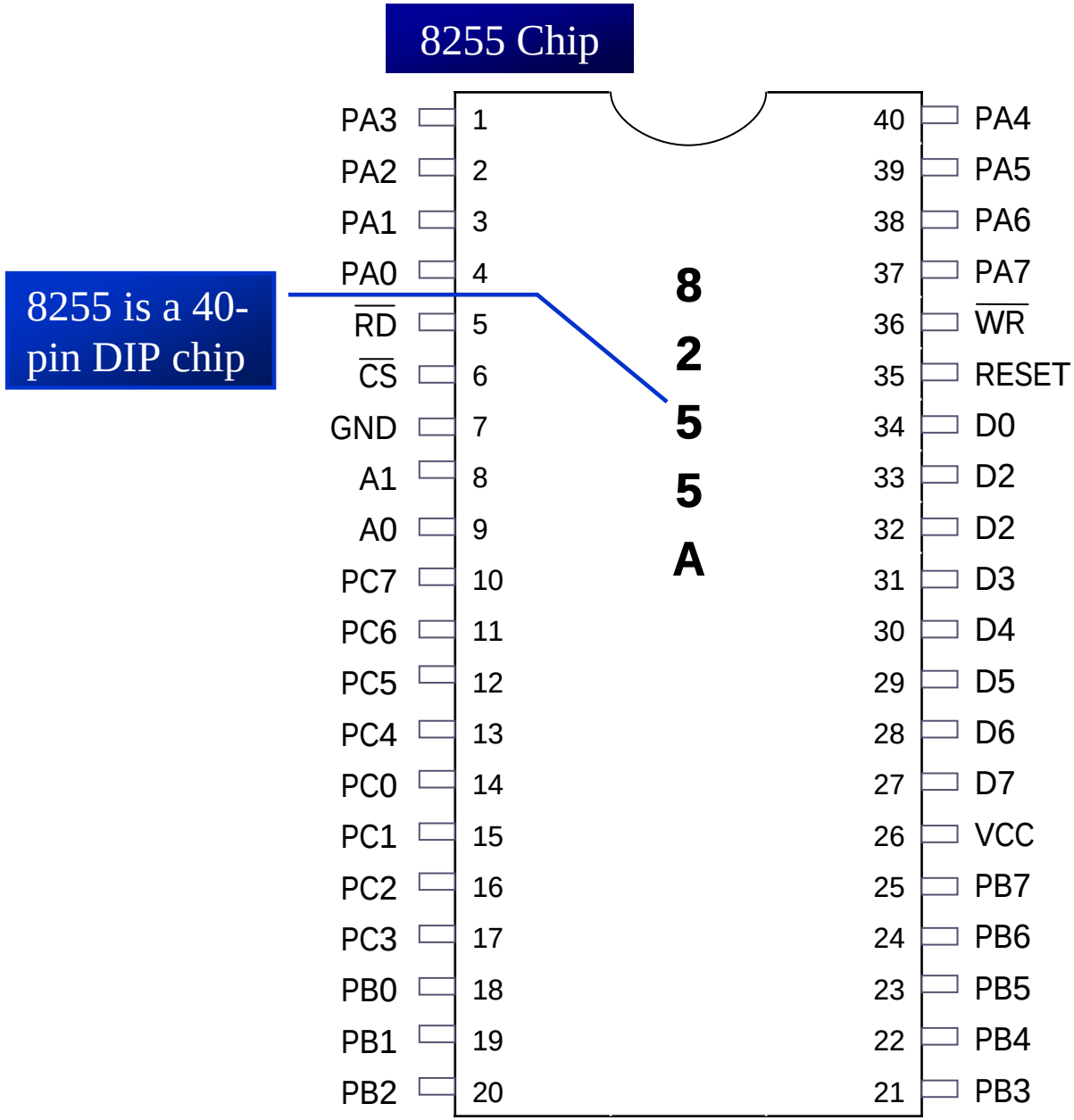
Chung-Ping Young
楊中平

Home Automation, Networking, and Entertainment Lab

Dept. of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

PROGRAMMING THE 8255

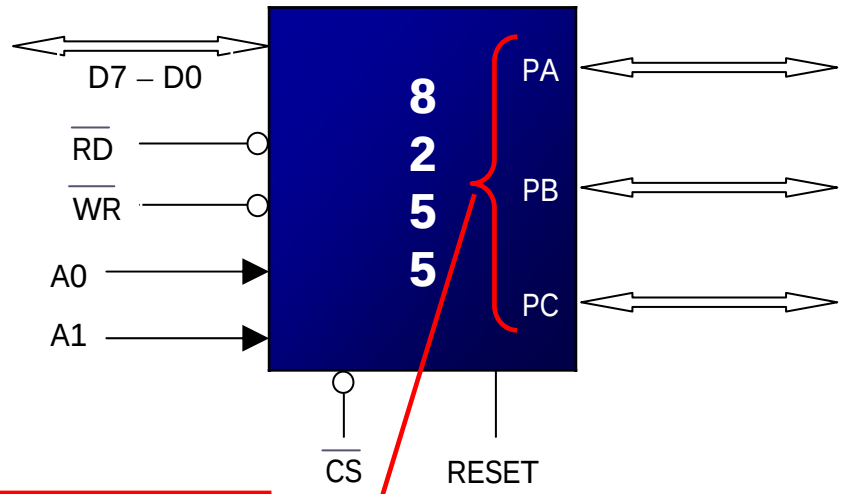
8255 Features



PROGRAMMING THE 8255

8255 Features (cont')

8255 Block Diagram

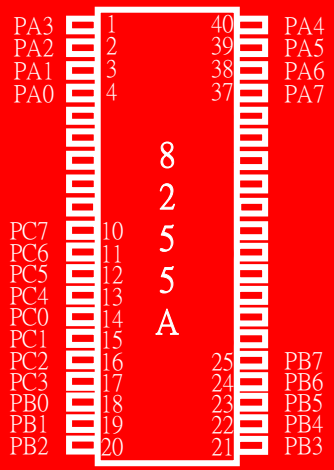


It has three separately accessible 8-bit ports, A, B, and C

- They can be programmed to input or output and can be changed dynamically
- They have handshaking capability

PROGRAMMING THE 8255

8255 Features (cont')

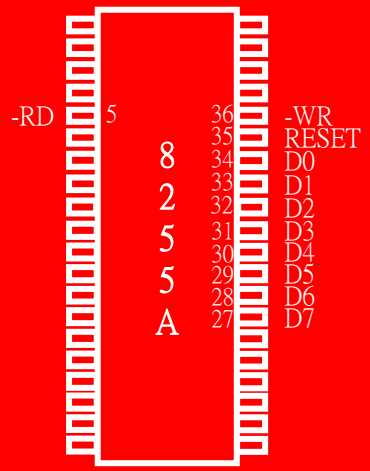


- PA0 - PA7 (8-bit port A)
  - Can be programmed as all input or output, or all bits as bidirectional input/output
- PB0 - PB7 (8-bit port B)
  - Can be programmed as all input or output, but cannot be used as a bidirectional port
- PC0 – PC7 (8-bit port C)
  - Can be all input or output
  - Can also be split into two parts:
    - CU (upper bits PC4 - PC7)
    - CL (lower bits PC0 – PC3)

    each can be used for input or output
  - Any of bits PC0 to PC7 can be programmed individually

PROGRAMMING THE 8255

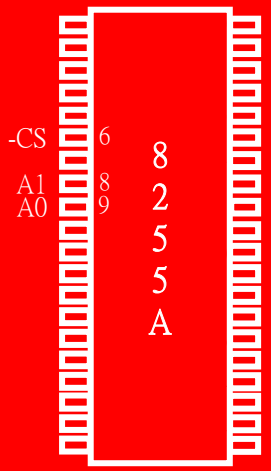
8255 Features (cont')



- $\overline{RD}$ and $\overline{WR}$
  - These two active-low control signals are inputs to the 8255
  - The $\overline{RD}$ and $\overline{WR}$ signals from the 8031/51 are connected to these inputs
- D0 – D7
  - are connected to the data pins of the microcontroller
  - allowing it to send data back and forth between the controller and the 8255 chip
- RESET
  - An active-high signal input
  - Used to clear the control register
    - When RESET is activated, all ports are initialized as input ports

PROGRAMMING THE 8255

8255 Features (cont')



- A0, A1, and $\overline{CS}$ (chip select)
  - $\overline{CS}$ is active-low
  - While $\overline{CS}$ selects the entire chip, it is A0 and A1 that select specific ports
  - These 3 pins are used to access port A, B, C, or the control register

8255 Port Selection

CS	A1	A0	Selection
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control register
1	X	X	8255 is not selected

PROGRAMMING THE 8255

Mode Selection of 8255

- While ports A, B and C are used to input or output data, the control register must be programmed to select operation mode of three ports
- The ports of the 8255 can be programmed in any of the following modes:
  1. Mode 0, simple I/O
     - Any of the ports A, B, CL, and CU can be programmed as input out output
     - All bits are out or all are in
     - There is no signal-bit control as in P0-P3 of 8051

PROGRAMMING THE 8255

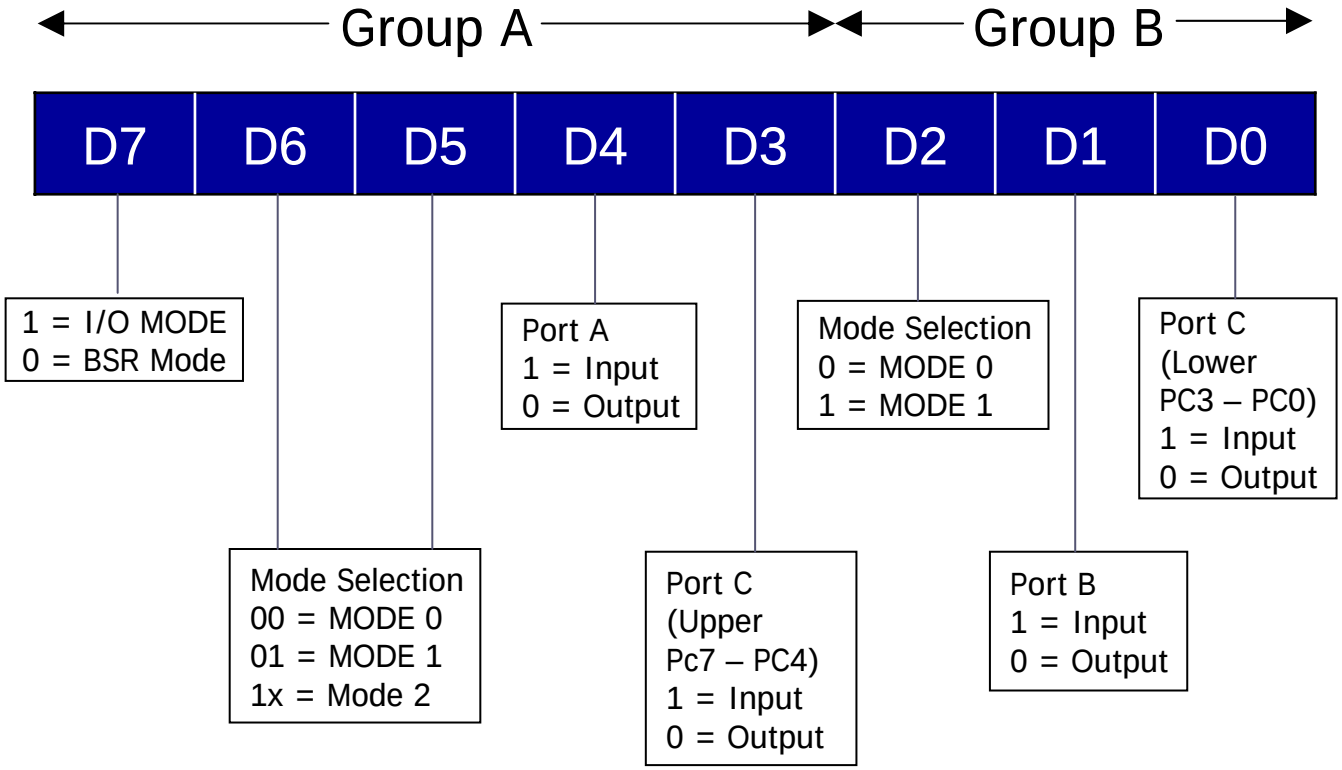
Mode Selection of 8255 (cont')

2. Mode 1
   - Port A and B can be used as input or output ports with handshaking capabilities
   - Handshaking signals are provided by the bits of port C
3. Mode 2
   - Port A can be used as a bidirectional I/O port with handshaking capabilities provided by port C
   - Port B can be used either in mode 0 or mode 1
4. BSR (bit set/reset) mode
   - Only the individual bits of port C can be programmed

PROGRAMMING THE 8255

Mode Selection of 8255 (cont')

8255 Control Word Format (I/O Mode)



PROGRAMMING THE 8255

Simple I/O Programming

- The more commonly used term is I/O
- Mode 0
  - Intel calls it the basic input/output mode
  - In this mode, any ports of A, B, or C can be programmed as input or output
    - A given port cannot be both input and output at the same time

Example 15-1

Find the control word of the 8255 for the following configurations:

(a) All the ports of A, B and C are output ports (mode 0)

(b) PA = in, PB = out, PCL = out, and PCH = out

Solution:

From Figure 15-3 we have:

(a) `1000 0000 = 80H` (b)`1001 0000 = 90H`

PROGRAMMING THE 8255

Connecting 8031/51 to 8255

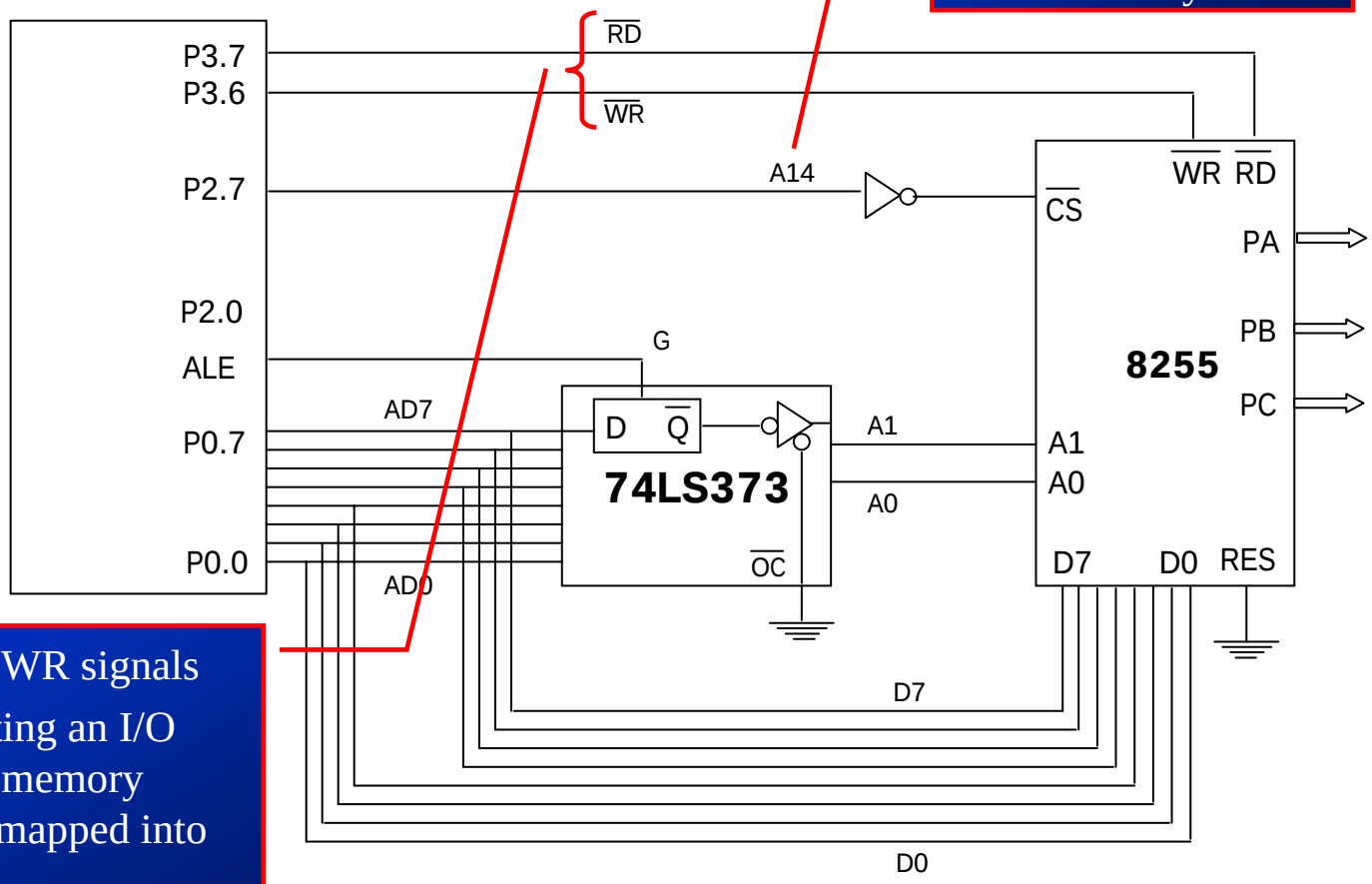
- The 8255 chip is programmed in any of the 4 modes
  - mentioned earlier by sending a byte (Intel calls it a control word) to the control register of 8255
- We must first find the port address assigned to each of ports A, B ,C and the control register
  - called mapping the I/O port

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

8051 Connection to the 8255

8255 is connected to an 8031/51 as if it is a RAM memory



Notice the use of RD and WR signals

- This method of connecting an I/O chip to a CPU is called memory mapped I/O, since it is mapped into memory space
- use memory space to access I/O
- use instructions such as MOVX to access 8255

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

Example 15-2

For Figure 15-4.

(a) Find the I/O port addresses assigned to ports A, B, C, and the control register.

(b) Program the 8255 for ports A, B, and C to be output ports.

(c) Write a program to send 55H and AAH to all ports continuously.

Solution

(a) The base address for the 8255 is as follows:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
X	1	X	x	X	x	x	x	x	x	x	x	x	x	0	0	= 4000H PA
X	1	X	X	x	x	x	x	x	x	x	x	x	X	0	1	= 4001H PB
X	1	X	X	x	x	x	x	x	x	x	x	X	X	1	0	= 4002H PC
x	1	x	X	x	x	x	x	x	x	x	x	x	x	1	1	= 4003H CR

(b) The control byte (word) for all ports as output is 80H as seen in Example 15-1.

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

Example 15-2 (cont')

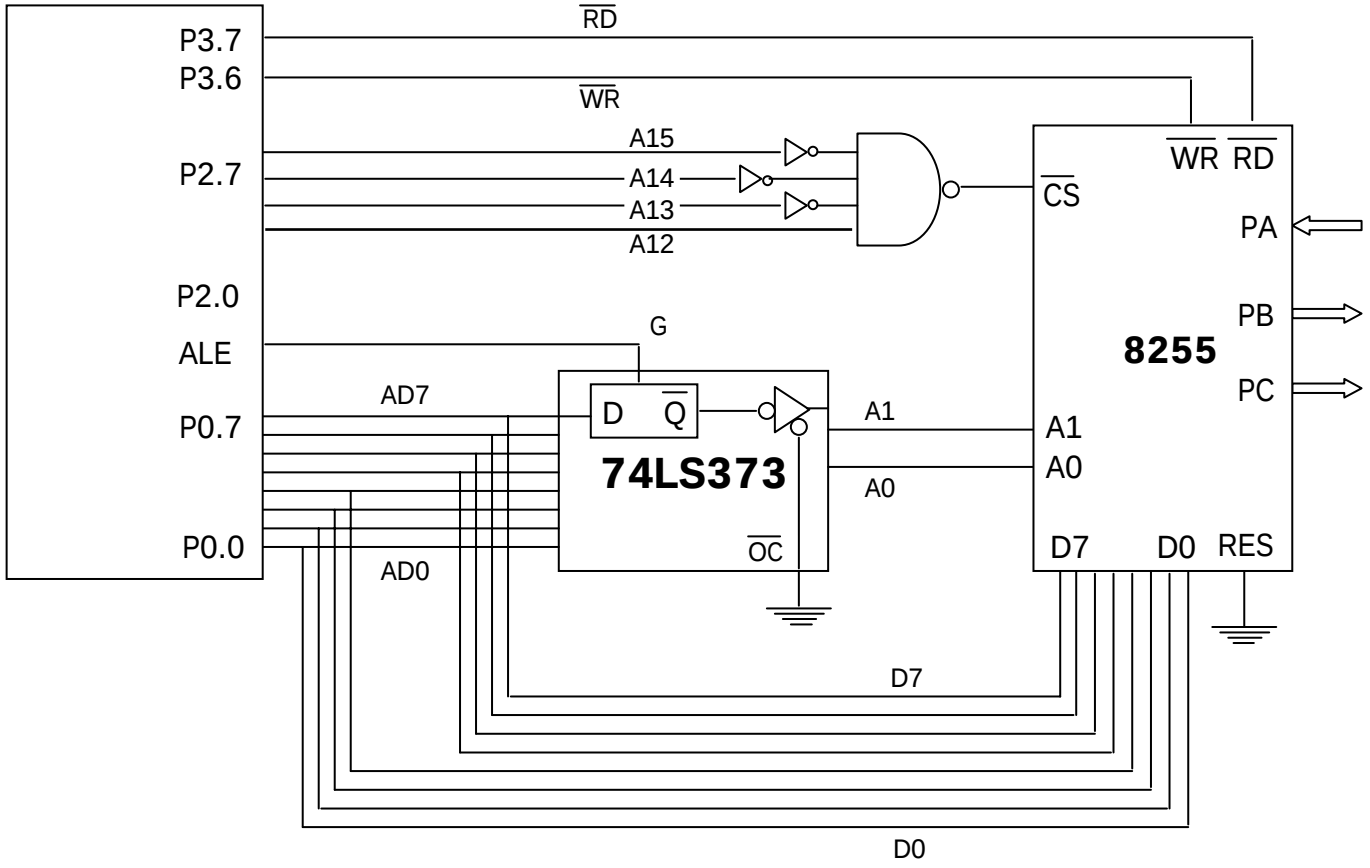
(c)

```
        MOV    A,#80H          ;control word
                               ;(ports output)
        MOV    DPTR,#4003H     ;load control reg
                               ;port address
        MOVX   @DPTR,A         ;issue control word
        MOV    A,#55H          ;A = 55H
AGAIN:  MOV    DPTR,#4000H     ;PA address
        MOVX   @DPTR,A         ;toggle PA bits
        INC    DPTR            ;PB address
        MOVX   @DPTR,A         ;toggle PB bits
        INC    DPTR            ;PC address
        MOVX   @DPTR,A         ;toggle PC bits
        CPL    A               ;toggle bit in reg A
        ACALL  DELAY           ;wait
        SJMP   AGAIN           ;continue
```

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

8051 Connection to the 8255



PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

Example 15-3

For Figure 15-5.
(a) Find the I/O port addresses assigned to ports A, B, C, and the control register.
(b) Find the control byte for PA = in, PB = out, PC = out.
(c) Write a program to get data from PA and send it to both B and C.

Solution:

(a) Assuming all the unused bits are 0s, the base port address for 8255 is 1000H. Therefore we have:

```
1000H  PA
1001H  PB
1002H  PC
1003H  Control register
```

(b) The control word is 10010000, or 90H.

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

Example 15-3 (cont')

(c)

```
    MOV  A,#90H        ;(PA=IN, PB=OUT, PC=OUT)
    MOV  DPTR,#1003H ;load control register
                       ;port address
    MOVX @DPTR,A       ;issue control word
    MOV  DPTR,#1000H ;PA address
    MOVX A,@DPTR       ;get data from PA
    INC  DPTR          ;PB address
    MOVX @DPTR, A      ;send the data to PB
    INC  DPTR         ;PC address
    MOVX @DPTR, A     ;send it also to PC
```

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

- For the program in Example 15-3
  - it is recommended that you use the EQU directive for port address as shown next

```
APORT           EQU    1000H
BPORT           EQU    1001H
CPORT           EQU    1002H
CNTPORT         EQU    1003H

MOV   A,#90H          ;(PA=IN, PB=OUT, PC=OUT)
MOV   DPTR,#CNTPORT   ;load cntr reg port addr
MOVX  @DPTR,A         ;issue control word
MOV   DPTR,#APORT     ;PA address
MOVX  A,@DPTR         ;get data from PA
INC   DPTR            ;PB address
MOVX  @DPTR,A         ;send the data to PB
INC   DPTR            ;PC address
MOVX  @DPTR,A         ;send it also to PC
```

PROGRAMMING THE 8255

Connecting 8031/51 to 8255 (cont')

- or, see the following, also using EQU:

```
CONTRBYT EQU 90H          ;(PA=IN, PB=OUT, PC=OUT)
BAS8255P EQU 1000H        ;base address for 8255
         MOV  A,#CONTRBYT
         MOV  DPTR,#BAS8255P+3 ;load c port addr
         MOVX @DPTR,A     ;issue control word
         MOV  DPTR,#BAS8255P+3 ;PA address
         . . .
```

- Example 15-2 and 15-3
  - use the DPTR register since the base address assigned to 8255 was 16-bit
  - if it was 8-bit, we can use "MOVX A,@R0" and "MOVX @R0,A"
- Example 15-4
  - use a logic gate to do address decoding
- Example 15-5
  - use a 74LS138 for multiple 8255s

PROGRAMMING THE 8255

Address Aliases

- Examples 15-4 and 15-5
  - decode the A0 - A7 address bit
- Examples 15-3 and 15-2
  - decode a portion of upper address A8 - A15
  - this partial address decoding leads to what is called ***address aliases***
  - could have changed all x's to various combinations of 1s and 0s
    - to come up with different address
    - they would all refer to the same physical port
- Make sure that all address aliases are documented, so that the users know what address are available if they want to expanded the system

PROGRAMMING THE 8255

Address Aliases (cont')

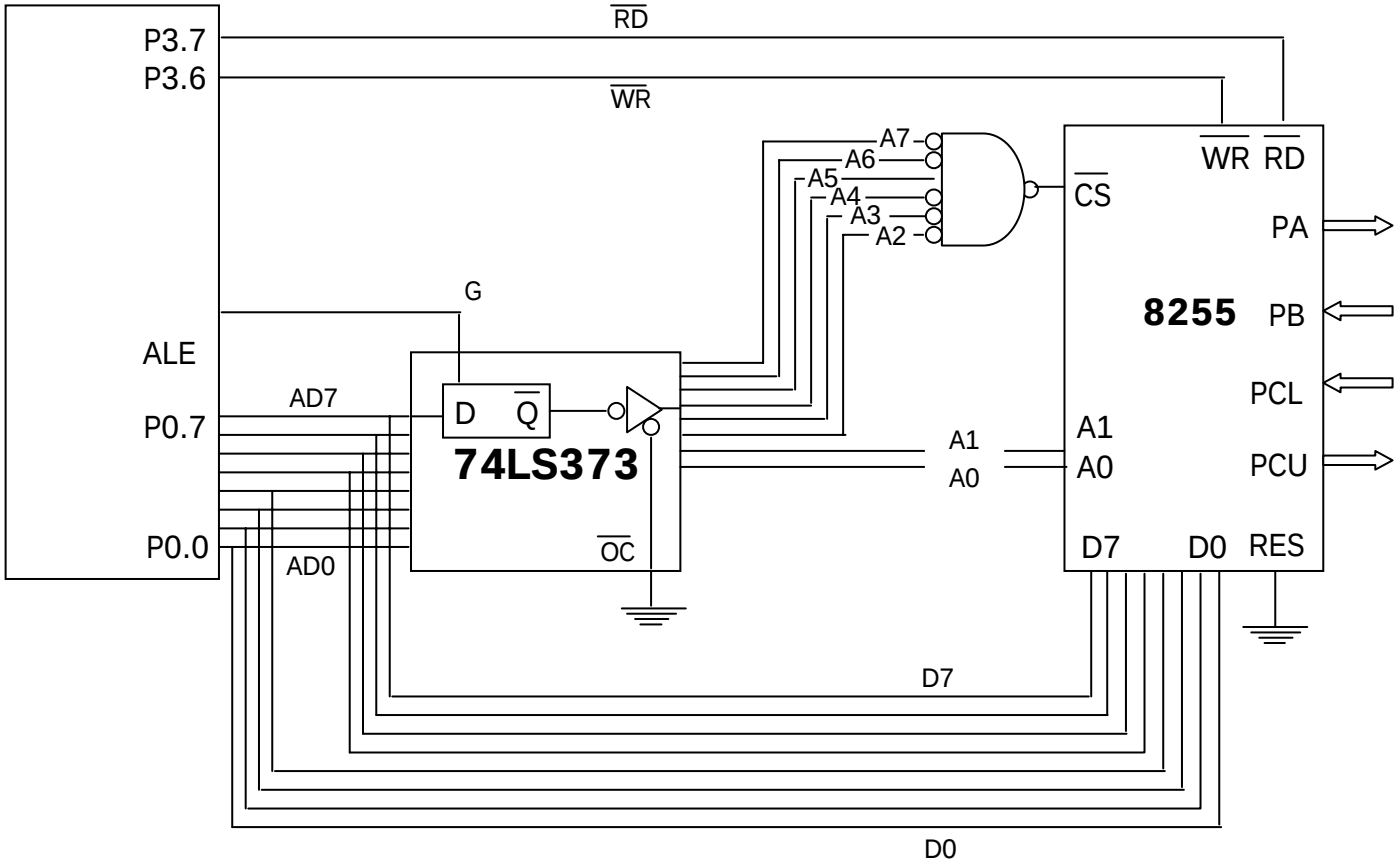


Figure 15-6. 8051 Connection to the 8255 for Example 15-4

PROGRAMMING THE 8255

Address Aliases (cont')

Example 15-4

For Figure 15-6.

(a) Find the I/O port addresses assigned to ports A, B, C, and the control register.

(b) Find the control byte for PA = out, PB = out, PC0 – PC3 = in, and PC4 – PC7 =out

(c) Write a program to get data from PB and send it to PA. In addition, data from PCL is sent out to PCU.

Solution:

(a) The port addresses are as follows:

	$\overline{CS}$	A1	A0	Address	Port
0010	00	0	0	20H	Port A
0010	00	0	1	21H	Port B
0010	00	1	0	22H	Port C
0010	00	1	1	23H	Control Reg

(a) The control word is 10000011, or 83H.

PROGRAMMING THE 8255

Address Aliases (cont')

Example 15-4 (cont')

(c)

```
CONTRBT EQU  83H ;PA=OUT, PB=IN, PCL=IN, PCU=OUT
APORT   EQU  20H
BPORT   EQU  21H
CPORT   EQU  22H
CNTPORT EQU  23H
         ...
        MOV  A,#CONTRBYT ;PA=OUT,PB=IN,PCL=IN,PCU=OUT
        MOV  R0,#CNTPORT ;LOAD CONTROL REG ADDRESS
        MOVX @R0,A       ;ISSUE CONTROL WORD
        MOV  R0,#BPORT   ;LOAD PB ADDRESS
        MOVX A,@R0       ;READ PB
        DEC  R0          ;POINT TO PA(20H)
        MOVX @R0,A       ;SEND IT TO PA
        MOV  R0,#CPORT   ;LOAD PC ADDRESS
        MOVX A,@R0       ;READ PCL
        ANL  A,#0FH      ;MASK UPPER NIBBLE
        SWAP A           ;SWAP LOW AND HIGH NIBBLE
        MOVX @R0,A       ;SEND TO PCU
```

PROGRAMMING THE 8255

Address Aliases (cont')

Example 15-5

Find the base address for the 8255 in Figure 15-7.

Solution:

G1	G2B	G2A	C	B	A			Address
A7	A6	A5	A4	A3	A2	A1	A0	
1	0	0	0	1	0	0	0	88H

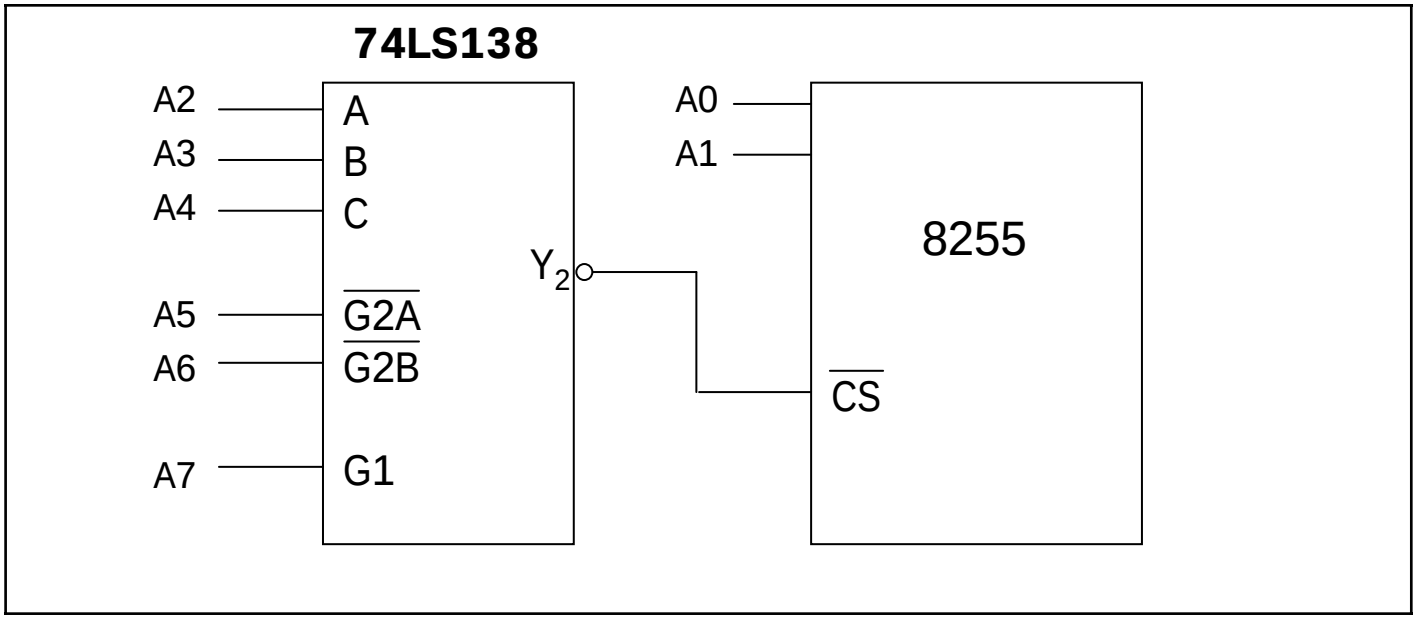


Figure 15-7. 8255 Decoding Using 74LS138

PROGRAMMING THE 8255

8031 System With 8255

- In 8031-based system
  - external program ROM is an absolute must
  - the use of 8255 is most welcome
  - this is due to the fact that 3031 to external program ROM, we lose the two ports P0 and P2, leaving only P1
- Therefore, connecting an 8255 is the best way to gain some extra ports.
  - Shown in Figure 15-8

PROGRAMMING THE 8255

8031 System With 8255 (cont')

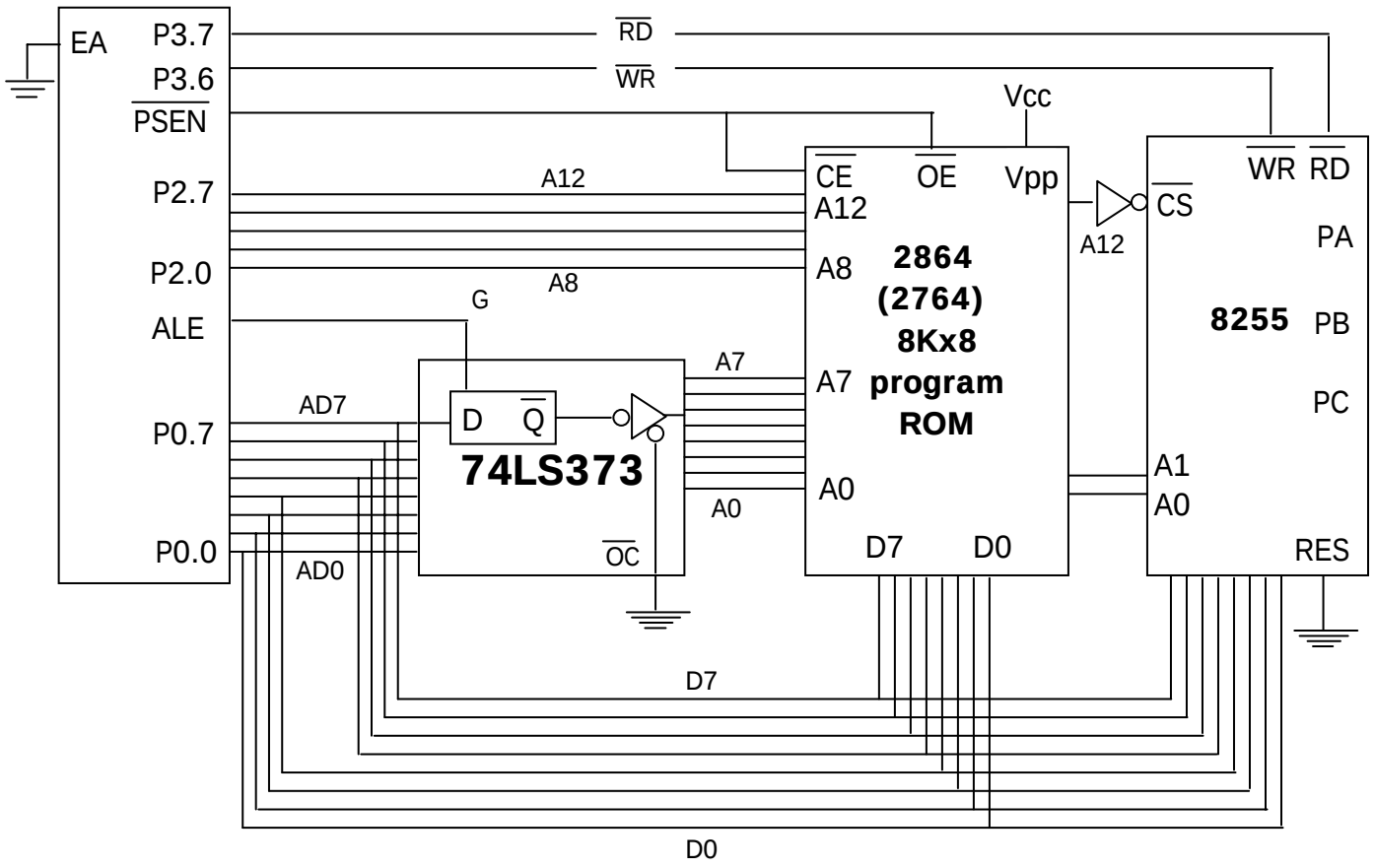


Figure 15-8. 8031 Connection to External Program ROM and the 8255

8255 INTERFACING

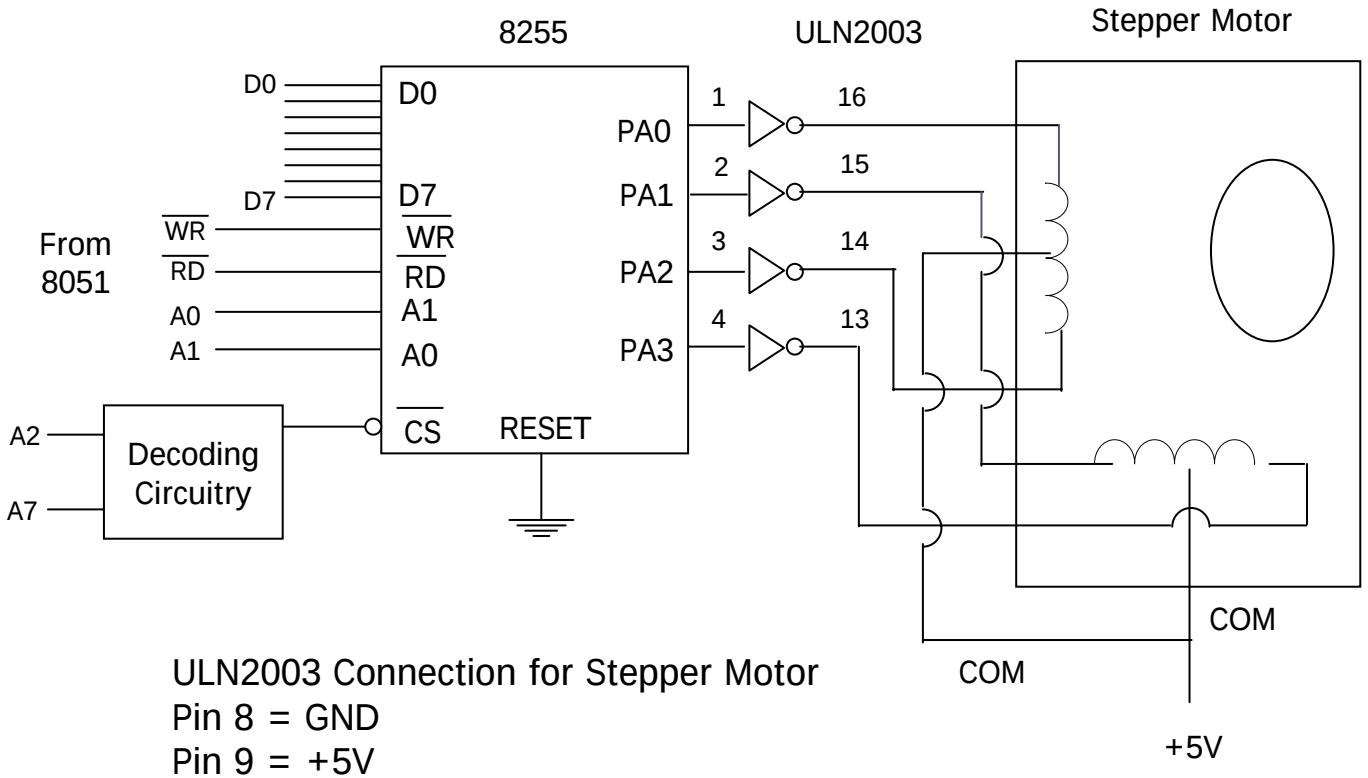
Stepper Motor Connection To The 8255

- Ch 13 detailed the interface of a stepper motor to the 8051
- Here show stepper motor connection to the 8255 and programming in Fig 15-9

```
        MOV   A,#80H      ;control word for PA=out
        MOV   R1,#CRPORT ;control reg port
      address
        MOVX  @R1,A       ;configure PA=out
        MOV   R1,#APORT   ;load PA address
        MOV   A,#66H ;A=66H,stepper motor
      sequence
AGAIN MOVX  @R1,A     ;issue motor sequence to
   PA
        RR    A      ;rotate sequence for
   clockwise
        ACALL DELAY        ;wait
        SJMP AGAIN
```

8255 INTERFACING

Stepper Motor Connection To The 8255 (cont')



ULN2003 Connection for Stepper Motor
Pin 8 = GND
Pin 9 = +5V

Use a separate power supply for the motor

Figure 15-9. 8255 Connection to Stepper Motor

8255 INTERFACING

LCD Connection To The 8255

- Program 15-1
  - Shows how to issue commands and data to an LCD. See Figure 15-10
  - must put a long delay before issue any information to the LCD
- Program 15-2
  - A repeat of Program 15-1 with the checking of the busy flag
  - Notice that no DELAY is used in the main program

8255 INTERFACING

LCD Connection To The 8255 (cont')

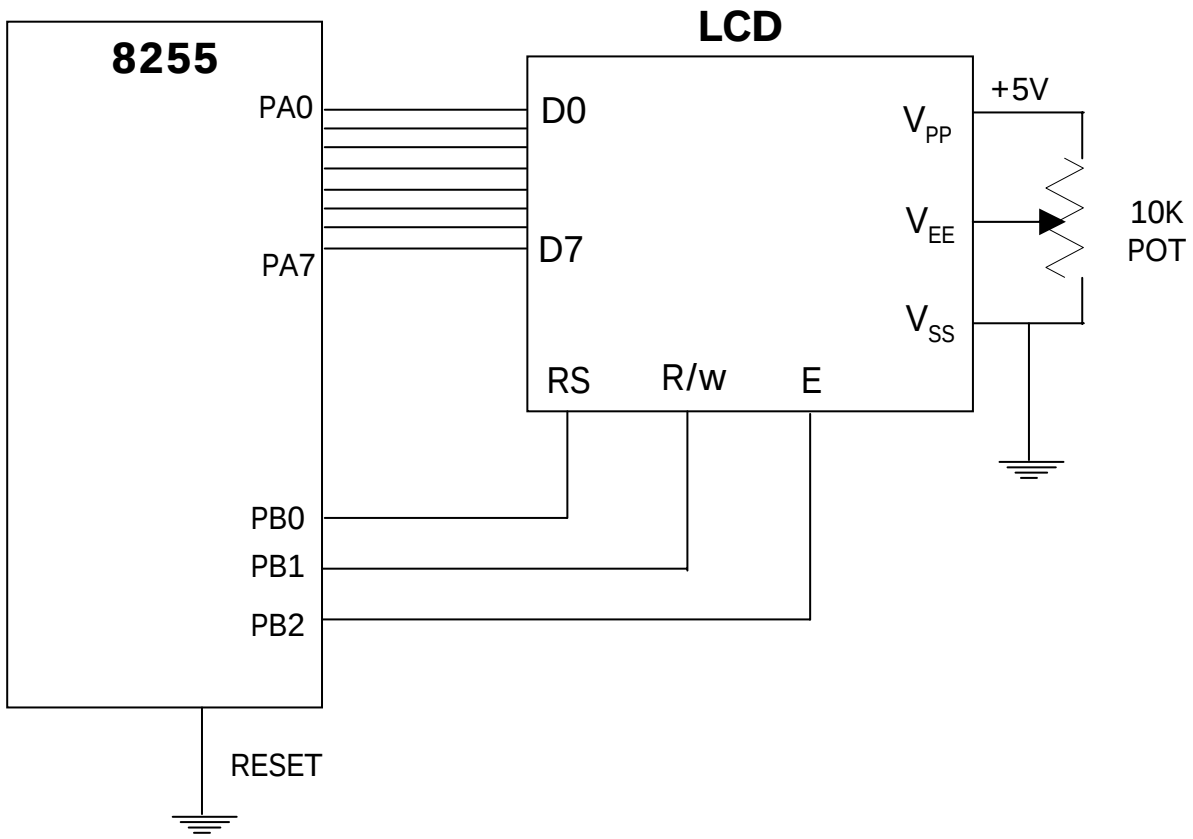


Figure 15-10. LCD Connection

8255 INTERFACING

LCD Connection To The 8255 (cont')

```
;Writing commands and data to LCD without checking busy flag
;Assume PA of 8255 connected to D0-D7 of LCD and
;PB0=RS, PB1=R/W, PB2=E for LCD's control pins connection
   MOV   A,#80H          ;all 8255 ports as output
   MOV   R0,#CNTPORT     ;load control reg address
   MOVX  @R0,A           ;issue control word
   MOV   A,#38H          ;LCD:2lines, 5X7 matrix
   ACALL CMDWRT          ;write command to LCD
   ACALL DELAY           ;wait before next issue(2 ms)
   MOV   A,#0EH          ;LCD command for cursor on
   ACALL CMDWRT          ;write command to LCD
   ACALL DELAY           ;wait before next issue
   MOV   A,#01H          ;clear LCD
   ACALL CMDWRT          ;write command to LCD
   ACALL DELAY           ;wait before next issue
   MOV   A,#06H          ;shift cursor right command
   ACALL CMDWRT          ;write command to LCD
   ACALL DELAY           ;wait before next issue
   . . . .               ;etc. for all LCD commands
   MOV   A,#'N'          ;display data (letter N)
   ACALL DATAWRT         ;send data to LCD display
   ACALL DELAY           ;wait before next issue
   MOV   A,#'O'          ;display data (letter O)
   ACALL DATAWRT         ;send data to LCD display
   ACALL DELAY           ;wait before next issue
   . . . .               ;etc. for other data
```

Program 15-1.

8255 INTERFACING

LCD Connection To The 8255 (cont')

```
;Command write subroutine, writes instruction commands to LCD
CMDWRT:  MOV   R0,#APORT     ;load port A address
         MOVX  @R0,A         ;issue info to LCD data pins
         MOV   R0,#BPORT     ;load port B address
         MOV   A,#00000100B  ;RS=0,R/W=0,E=1 for H-TO-L
         MOVX  @R0,A         ;activate LCD pins RS,R/W,E
         NOP                ;make E pin pulse wide enough
         NOP
         MOV   A,#00000000B  ;RS=0,R/W=0,E=0 for H-To-L
         MOVX  @R0,A         ;latch in data pin info
         RET
;Data write subroutine, write data to be display
DATAWRY:MOV   R0,#APORT     ;load port A address
         MOVX  @R0,A         ;issue info to LCD data pins
         MOV   R0,#BPORT     ;load port B address
         MOV   A,#00000101B  ;RS=1,R/W=0,E=1 for H-TO-L
         MOVX  @R0,A         ;activate LCD pins RS,R/W,E
         NOP                ;make E pin pulse wide enough
         NOP
         MOV   A,#00000001B  ;RS=1,R/W=0,E=0 for H-To-L
         MOVX  @R0,A        ;latch in LCD's data pin info
         RET
```

Program 15-1. (cont')

8255 INTERFACING

LCD Connection To The 8255 (cont')

```
;Writing commands to the LCD without checking busy flag
;PA of 8255 connected to D0-D7 of LCD and
;PB0=RS, PB1=R/W, PB2=E for 8255 to LCD's control pins connection
   MOV   A,#80H          ;all 8255 ports as output
   MOV   R0,#CNTPORT     ;load control reg address
   MOVX  @R0,A           ;issue control word
   MOV   A,#38H          ;LCD:2 LINES, 5X7 matrix
   ACALL NCMDWRT         ;write command to LCD
   MOV   A,#0EH          ;LCD command for cursor on
   ACALL NCMDWRT         ;write command to LCD
   MOV   A,#01H          ;clear LCD
   ACALL NCMDWRT         ;write command to LCD
   MOV   A,#06H          ;shift cursor right command
   ACALL NCMDWRT         ;write command to LCD
   . . . .               ;etc. for all LCD commands
   MOV   A,#'N'          ;display data (letter N)
   ACALL NDATAWRT        ;send data to LCD display
   MOV   A,#'O'          ;display data (letter O)
   CALL NDATAWRT         ;send data to LCD display
   . . . .               ;etc. for other data
```

Program 15-2.

8255 INTERFACING

LCD Connection To The 8255 (cont')

```
;New command write subroutine with checking busy flag
NCMDWRT:MOV  R2,A          ;save a value
        MOV  A,#90H        ;PA=IN to read LCD status
        MOV  R0,#CNTPORT   ;load control reg address
        MOVX @R0,A         ;configure PA=IN, PB=OUT
        MOV  A,#00000110B  ;RS=0,R/W=1,E=1 read command
        MOV  R0,#BPORT     ;load port B address
        MOVX @R0,A       ;RS=0,R/W=1 for RD and RS pins
        MOV  R0,#APORT     ;load port A address
READY:  MOVX A,@R0         ;read command reg
        PLC  A           ;move D7(busy flag) into carry
        JC   READY         ;wait until LCD is ready
        MOV  A,#80H        ;make PA and PB output again
        MOV  R0,#CNTPORT   ;load control port address
        MOVX @R0,A         ;issue control word to 8255
        MOV  A,R2          ;get back value to LCD
        MOV  R0,#APORT     ;load port A address
        MOVX @R0,A       ;issue info to LCD's data pins
        MOV  R0,#BPORT     ;load port B address
        MOV  A,#00000100B  ;RS=0,R/W=0,E=1 for H-To-L
        MOVX @R0,A       ;activate RS,R/W,E pins of LCD
        NOP                ;make E pin pulse wide enough
        NOP
        MOV  A,#00000000B  ;RS=0,R/W=0,E=0 for H-To-L
        MOVX @R0,A         ;latch in LCD's data pin info
        RET
```

Program 15-2. (cont')

8255 INTERFACING

LCD Connection To The 8255 (cont')

```
;New data write subroutine with checking busy flag
NDATAWRT:MOV  R2,#A          ;save a value
         MOV  A,#90H    ;PA=IN to read LCD status,PB=out
         MOV  R0,#CNTPORT  ;load control port address
         MOVX @R0,A         ;configure PA=IN, PB=OUT
         MOV  A,#00000110B ;RS=0,R/W=1,E=1 read command
         MOV  R0,#BPORT     ;load port B address
         MOVX @R0,A       ;RS=0,R/W=1 for RD and RS pins
         MOV  R0,#APORT     ;load port A address
READY:   MOVX A,@R0         ;read command reg
         PLC  A            ;move D7(busy flag) into carry
         JC   READY          ;wait until LCD is ready
         MOV  A,#80H         ;make PA and PB output again
         MOV  R0,#CNTPORT  ;load control port address
         MOVX @R0,A          ;issue control word to 8255
         MOV  A,R2           ;get back value to LCD
         MOV  R0,#APORT     ;load port A address
         MOVX @R0,A       ;issue info to LCD's data pins
         MOV  R0,#BPORT     ;load port B address
         MOV  A,#00000101B ;RS=1,R/W=0,E=1 for H-To-L
         MOVX @R0,A       ;activate RS,R/W,E pins of LCD
         NOP                ;make E pin pulse wide enough
         NOP
         MOV  A,#00000001B ;RS=1,R/W=0,E=0 for H-To-L
         MOVX @R0,A        ;latch in LCD's data pin info
         RET
```

Program 15-2. (cont')

8255 INTERFACING

ADC Connection To The 8255

- the following is a program for the ADC connected to 8255 as show in fig 15-11

```
    MOV  A,#80H    ;ctrl word for PA=OUT
PC=IN
    MOV  R1,#CRPORT    ;ctrl reg port address
    MOVX @R1,A         ;configure PA=OUT
PC=IN
BACK: MOV  R1,#CRORT     ;load port C address
    MOVX A,@R1 ;read PC to see if ADC is
ready
    ANL  A,#00000001B  ;mask all except PC0
    ;end of conversation, now get ADC data
    MOV  R1,#APORT     ;load PA address
    MOVX A,@R1         ;A=analog data input
```

8255 INTERFACING

ADC Connection To The 8255 (cont')

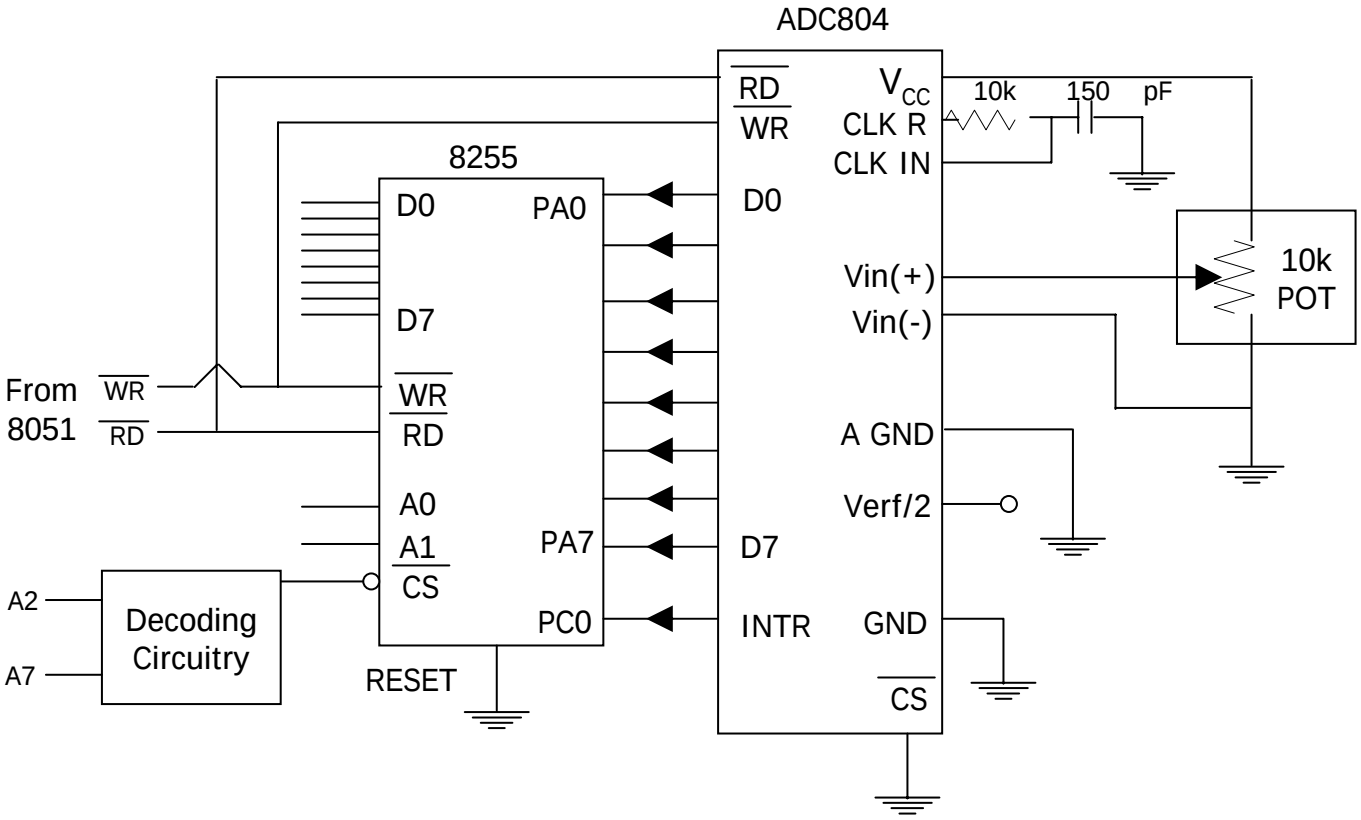


Figure 15-11. 8255 Connection to ADC804

OTHER MODES OF THE 8255

BSR (Bit Set/Reset) Mode

- A unique feature of port C
  - The bits can be controlled individually
- BSR mode allows one to set to high or low any of the PC0 to PC7, see figure 15-12.

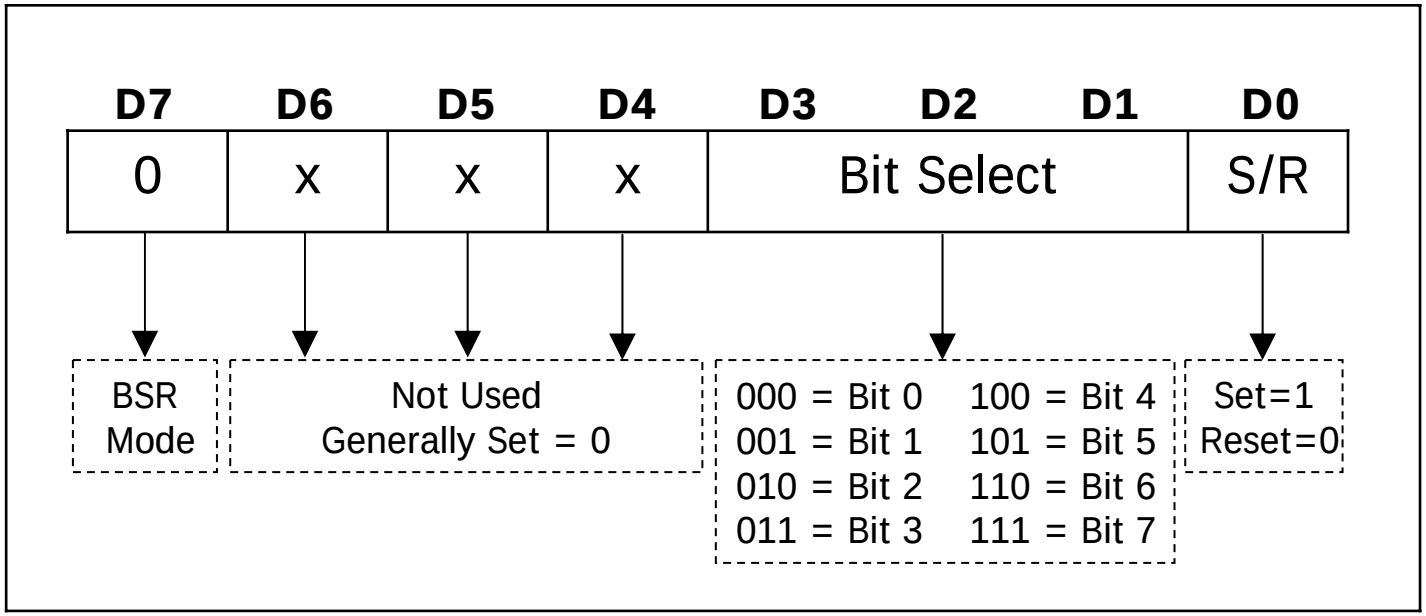


Figure 15-12. BSR Control Word

OTHER MODES OF THE 8255

BSR (Bit Set/Reset) Mode (cont')

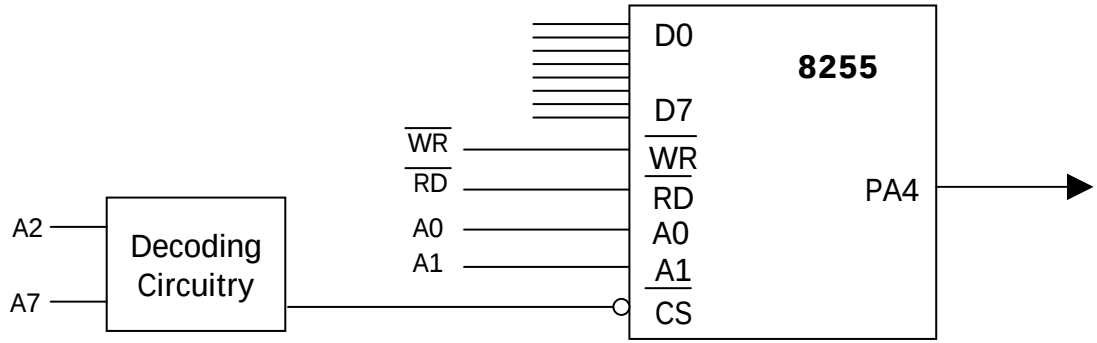
Example 15-6

Program PC4 of the 8255 to generate a pulse of 50 ms with 50% duty cycle.

Solution:

To program the 8255 in BSR mode, bit D7 of the control word must be low. For PC4 to be high, we need a control word of "0xxx1001". Likewise, for low we would need "0xxx1000" as the control word. The x's are for "don't care" and generally are set to zero.

```
MOV     a,#00001001B   ;control byte for PC4=1
MOV     R1,#CNTPORT    ;load control reg port
MOVX    @R1,A          ;make PC4=1
ACALL   DELAY          ;time delay for high pulse
MOV     A,00001000B    ;control byte for PC4=0
MOVX    @R1,A          ;make PC4=0
ACALL   DELAY
```



Configuration for Examples 15-6, 15-7

OTHER MODES OF THE 8255

BSR (Bit Set/Reset) Mode (cont')

Example 15-7

Program the 8255 in Figure 15-13 for the following.
(a) Set PC2 to high.
(b) Use PC6 to generate a square

Solution:

(a)

```
        MOV   R0,#CNTPORT
        MOV   A,#0XXX0101   ;control byte
        MOVX  @R0,A
```

(b)

```
AGAIN   MOV   A,#00001101B ;PC6=1
        NOV   R0,#CNTPROT  ;load control port add
        MOVX  @R0,A        ;make PC6=1
        ACALL DELAY
        ACALL DELAY
        MOV   A,#00001100B ;PC6=0
        ACALL DELAY      ;time delay for low pulse
        SJMP  AGAIN
```

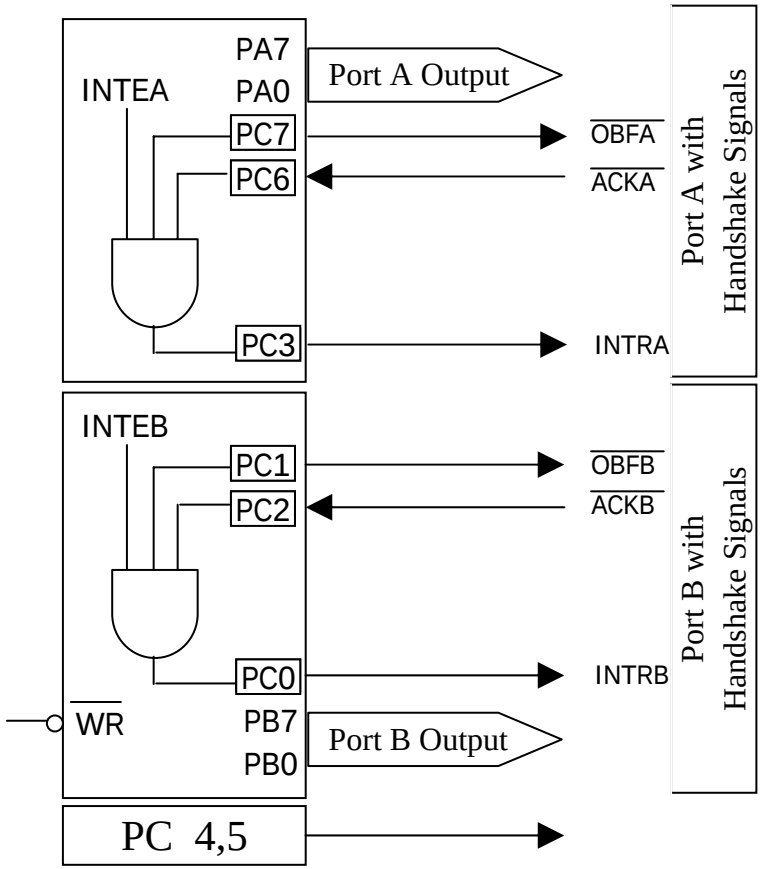
OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability

- One of the most powerful features of 8255 is to handle handshaking signals
- Handshaking refers to the process of two intelligent devices communicating back and forth
  - Example--printer
- Mode 1: outputting data with handshaking signals
  - As show in Figure 15-14
  - A and B can be used to send data to device with handshaking signals
  - Handshaking signals are provided by port C
  - Figure 15-15 provides a timing diagram

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')



Control Word – Mode 1 Output

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	0	1/0	1	0	x
I/O Mode	Port A Mode 1	Port A Mode 1	Port A Output	PC 4,5 1=Input,0=Output	Port B Output	Port B Mode 1	Port B Output

Status Word – Mode 1 Output

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTEA	I/O	I/O	INTRA	INTEB	OBFB	INTRB

INTEA is controlled by PC6 in BSR mode.

INTEB is controlled by PC2 in BSR mode.

8255 Mode 1 Output Diagram

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')

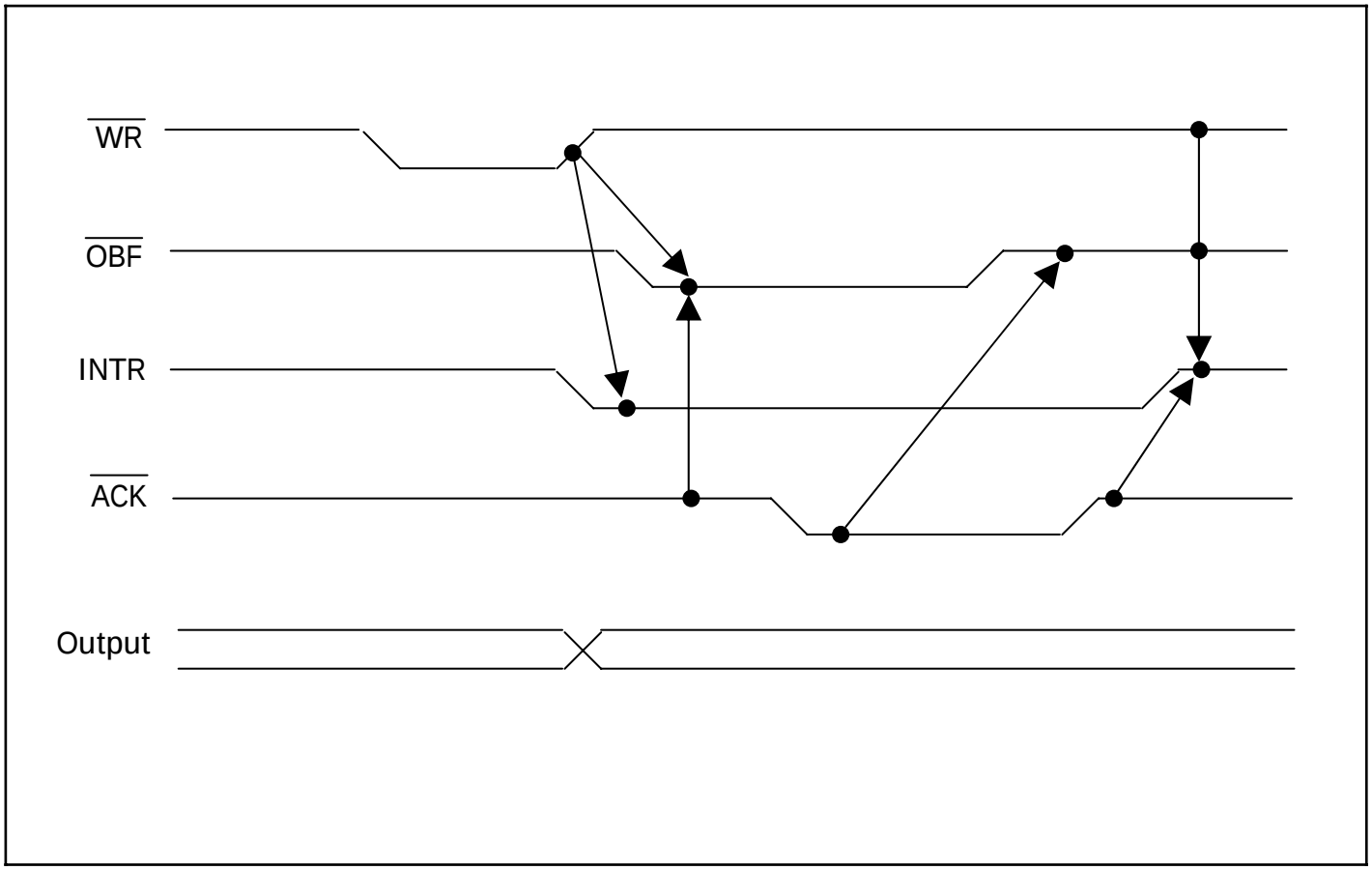


Figure 15-15. Timing Diagram for Mode 1 Output

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')

- The following paragraphs provide the explanation of and reasoning behind handshaking signals only for port A, but in concept they re exactly the same as for port B

  - $\overline{\text{OBFa}}$ (output buffer full for port A)
    - an active-low signal going out of PC7
    - indicate CPU has written a byte of data in port A
    - OBFa must be connected to STROBE of the receiving equipment (such as printer) to inform it that it can now read a byte of data from the Port A latch

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')

- $\overline{ACK}$a (acknowledge for port A)
  - active-low input signal received at PC6 of 8255
  - Through $\overline{ACK}$, 8255 knows that the data at port A has been picked up by the receiving device
  - When the receiving device picks up the data at port A, it must inform the 8255 through $\overline{ACK}$
  - 8255 in turn makes $\overline{OBF}$a high, to indicate that the data at the port is now old data
  - $\overline{OBF}$a will not go low until the CPU writes a new byte pf data to port A
- INTRa (interrupt request for port A)
  - Active-high signal coming out of PC3
  - The $\overline{ACK}$ signal is a signal of limited duration

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')

- When it goes active it makes $\overline{OBF}$a inactive, stays low for a small amount of time and then goes back to high
- it is a rising edge of $\overline{ACK}$ that activates INTRa by making it high
- This high signal on INTRa can be used to get the attention of the CPU
- The CPU is informed through INTRa that the printer has received the last byte and is ready to receive another one
- INTRa interrupts the CPU in whatever it is doing and forces it to write the next byte to port A to be printed
- It is important to note that INTRa is set to 1 only if INTEa, $\overline{OBF}$, and $\overline{ACK}$a are all high
- It is reset to zero when the CPU writes a byte to port A

OTHER MODES OF THE 8255

8255 in Mode 1: I/O With Handshaking Capability (cont')

- INTEa (interrupt enable for port A)
  - The 8255 can disable INTRa to prevent it if from interrupting the CPU
  - It is internal flip-plop designed to mask INTRa
  - It can be set or reset through port C in BSR mode since the INTEa flip-flop is controlled through PC6
  - INTEb is controlled by PC2 in BSR mode
- Status word
  - 8255 enables monitoring of the status of signals INTR, OBF, and INTE for both ports A and B
  - This is done by reading port C into accumulator and testing the bits
  - This feature allows the implementation of polling instead of a hardware interrupt

OTHER MODES OF THE 8255

Printer Signal

- To understand handshaking with the 8255, we give an overview of printer operation, handshaking signals
- The following enumerates the steps of communicating with a printer
  - 1. A byte of data is presented to the data bus of the printer
  - 2. The printer is informed of the presence of a byte of data to be printed by activating its $\overline{\text{Strobe}}$ input signal
  - 3. whenever the printer receives the data it informs the sender by activating an output signal called $\overline{\text{ACK}}$ (acknowledge)
  - 4. signal $\overline{\text{ACK}}$ initiates the process of providing another byte of data to printer
- Table 15-2 provides a list of signals for Centronics printers

OTHER MODES OF THE 8255

Printer Signal (cont')

Table 15-2. DB-25 Printer Pins

Pin	Description
1	$\overline{\text{Srtobe}}$
2	Data bit 0
3	Data bit 1
4	Data bit 2
5	Data bit 3
6	Data bit 4
7	Data bit 5
8	Data bit 6
9	Data bit 7
10	$\overline{\text{ACK}}$ (acknowledge)
11	Busy
12	Out of paper
13	Select
14	$\overline{\text{Auto feed}}$
15	$\overline{\text{Error}}$
16	Initialize printer
17	$\overline{\text{Select input}}$
18 - 25	Ground

OTHER MODES OF THE 8255

Printer Signal (cont')

- As we can see from the steps above, merely presenting a byte of data to the printer is not enough
  - The printer must be informed of the presence of the data
  - At the time the data is sent, the printer might be busy or out of paper
    - So the printer must inform the sender whenever it finally pick up the data from its data bus
- Fig 15-16 and 15-17 show DB-25 and Centronics sides of the printer cable
- Connection of the 8031/51 with the printer and programming are left to the reader to explore

OTHER MODES OF THE 8255

Printer Signal (cont')

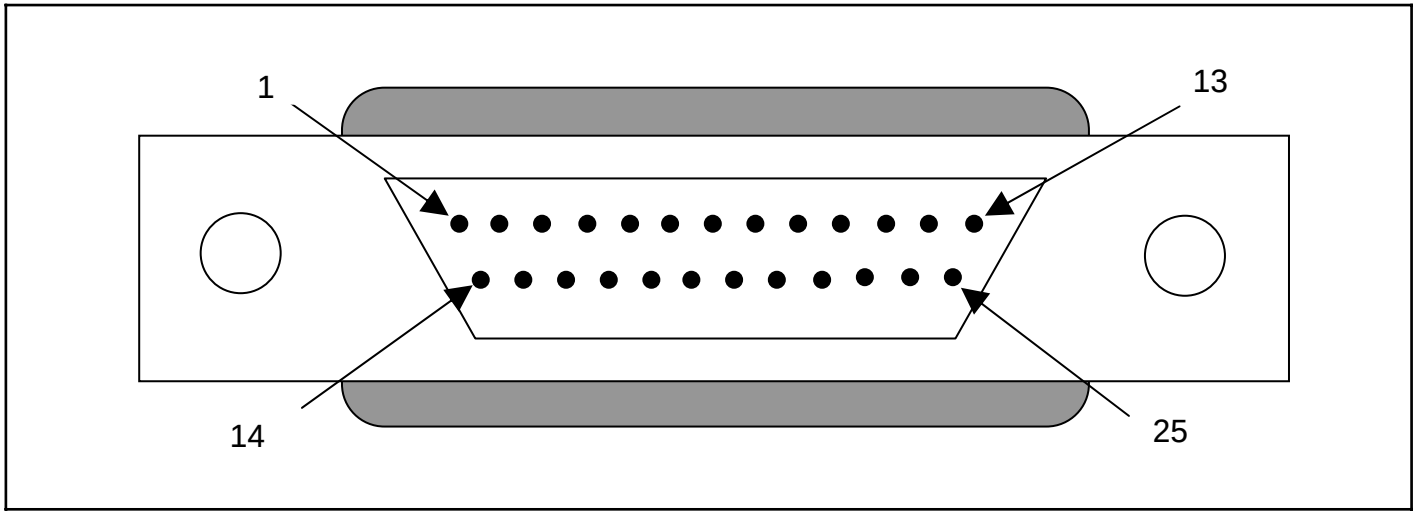


Figure 15-16. DB-25 Connector

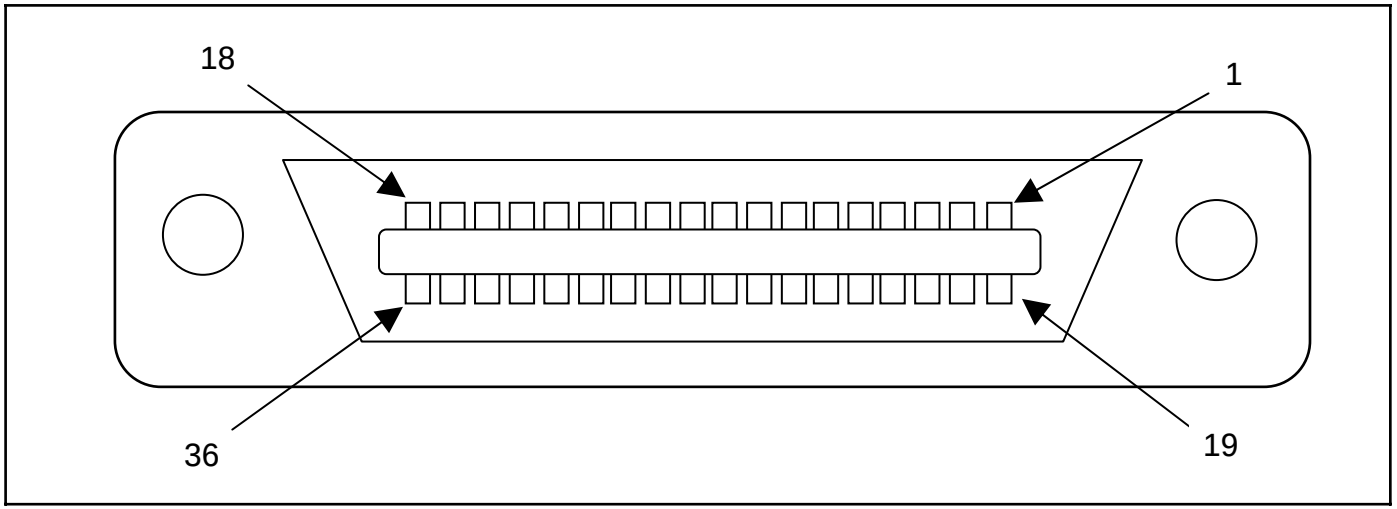


Figure 15-17. 36-Pin Centronics Connector

OTHER MODES OF THE 8255

Printer Signal (cont')

Table 15-3. Centronics Printer Specification

Serial	Return	Signal	Direction	Description
1	19	STROBE	IN	STROBE pulse to read data in. Pulse width must be more than 0.5 μs at receiving terminal. The signal level is normally "high"; read-in of data is performed at the "low" level of this signal.
2	20	DATA 1	IN	These signals represent information of the 1st to 8th bits of parallel data, respectively. Each signal is at "high" level when data is logical "1", and "low" when logical "0"
3	21	DATA 2	IN	" "
4	22	DATA 3	IN	" "
5	23	DATA 4	IN	" "
6	24	DATA 5	IN	" "
7	25	DATA 6	IN	" "
8	26	DATA 7	IN	" "
9	27	DATA 8	IN	" "
10	28	ACKNLG	OUT	Approximately 0.5 μs pulse; "low" indicates data has been received and printer is ready for data.
11	29	BUSY	OUT	A "high" signal indicates that the printer cannot receive data. The signal becomes "high" in the following case: (1)during data entry, (2) during printing operation,(3)in "off-line" status, (4)during printer error status.
12	30	PE	OUT	A "high" signal indicates that printer is out of paper
13	--	SLCT	OUT	Indicates that the printer is in the state selected.

OTHER MODES OF THE 8255

Printer Signal (cont')

Table 15-3. Centronics Printer Specification (cont')

Serial	Return	Signal	Direction	Description
14	--	AUTOFEEDXT	IN	When the signal is at "low" level, the paper is fed automatically one line after printing. (The signal level can be fixed to "low" with DIP SW pin 2-3 provided on the control circuit board.)
15	--	NC	--	Not used
16	--	0V	--	Logic GND level
17	--	CHASISGND	--	Printer chassis GND. In the printer, chassis GND and the logical GND are isolated from each other.
18	--	NC	--	Not used
19–30	--	GND	--	"Twisted-pair return" signal; GND level
31	--	INIT	IN	When this signal becomes "low" the printer controller is reset to its initial state and the print buffer is cleared. Normally at "high" level; its pulse width must be more than 50 μs at receiving terminal
32	--	ERROR	OUT	The level of this signal becomes "low" when printer is in "paper end", "off-line", and error state
33	--	GND	--	Same as with pin numbers 19 t0 30
34	--	NC	--	Not used
35	--		--	Pulled up to +5V dc through 4.7 K ohms resistance.
36	--	SLCTIN	IN	Data entry to the printer is possible only when the level of this signal is "low" .(Internal fixing can be carried out with DIP SW 1-8. The condition at the time of shipment is set "low" for this signal.)