Microcontrollers

BUNTY
Lecturer in ECE
Govt. Polytechnic Jhajjar (HR)

Chapter-1
Microcontroller series (MCS) – 51
Overview

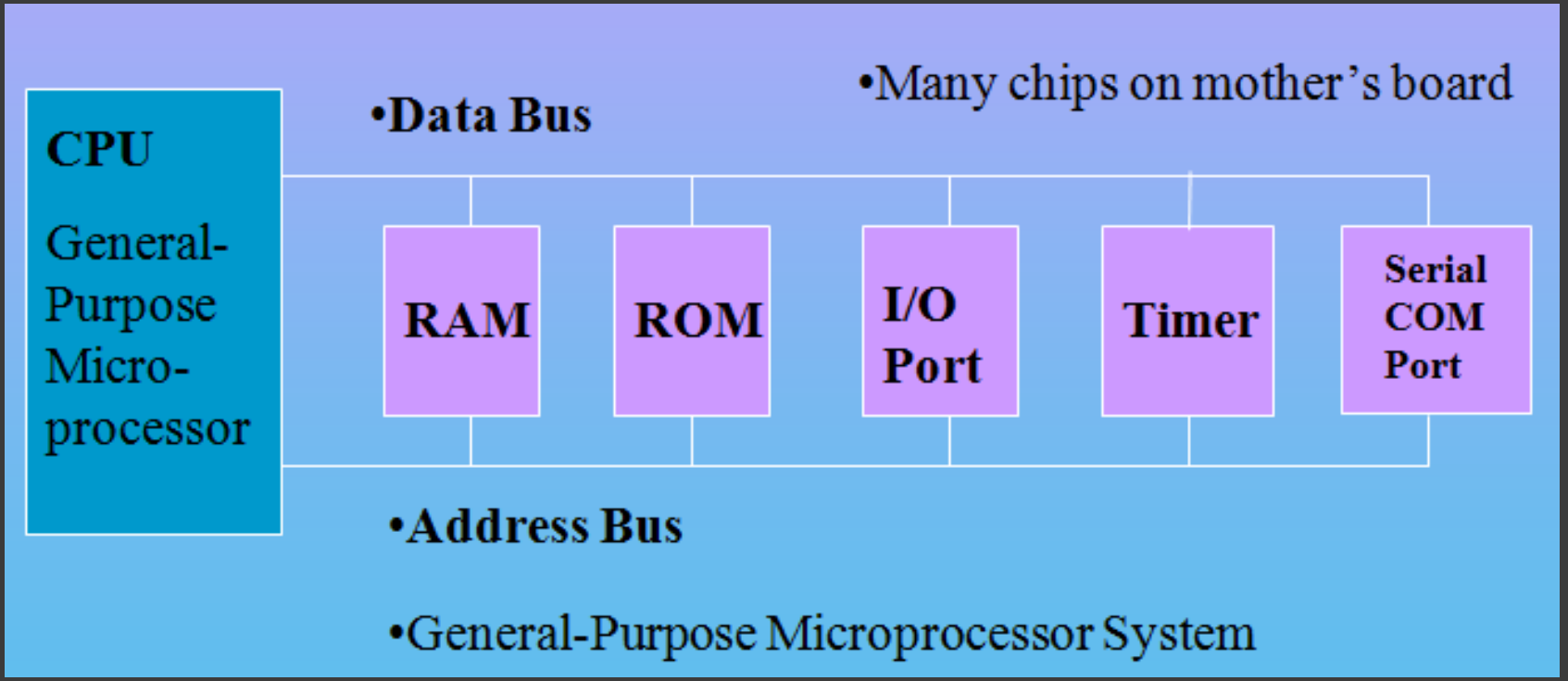
1. Architecture of 8051
2. Pin Details
3. I/O Port Structure

Microprocessor Based System

CPU

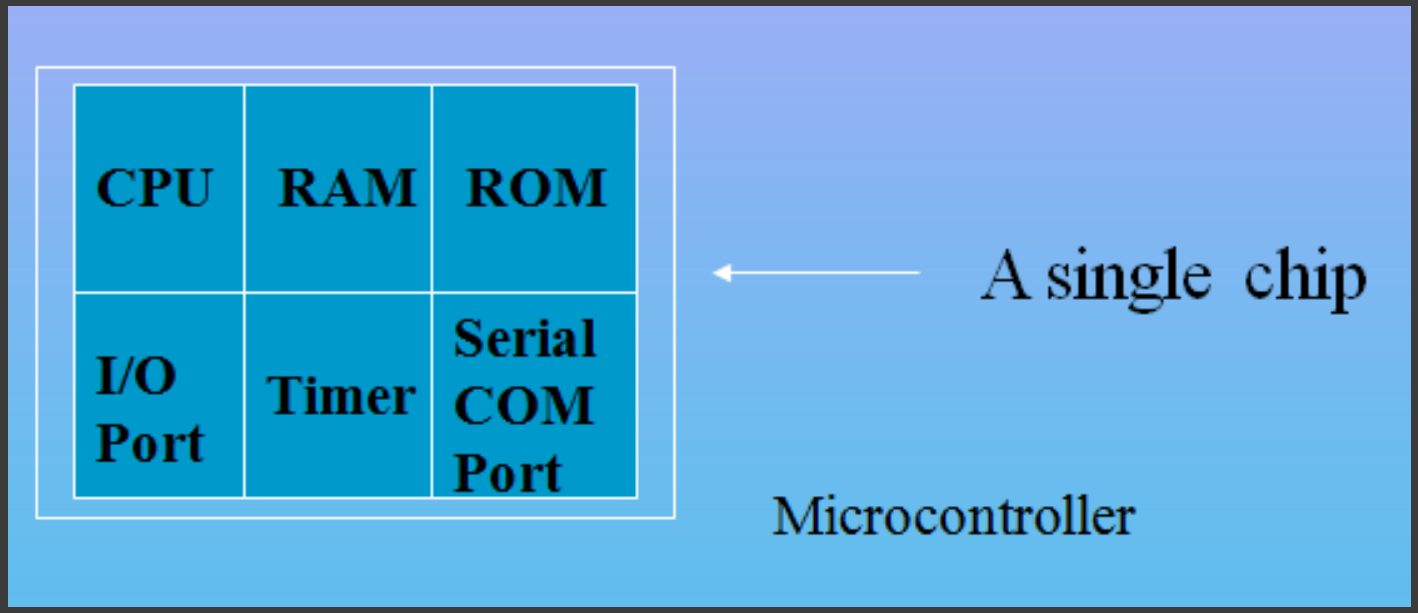
External RAM, ROM, I/O

(No internal RAM, ROM, I/O ports in the CPU)



Microcontroller

- A smaller computer on a CHIP
- On-chip RAM, ROM, I/O Ports, Timer, Serial Controller…
- Example: Motorola's 6811, Intel's 8051, Atmel 32



Microprocessor vs. Microcontroller

Microprocessor

- CPU is stand-alone, RAM, ROM, I/O, timer are separate
- Designer can decide on the amount of ROM, RAM and I/O ports.
- Expansive
- Versatility
- General-purpose

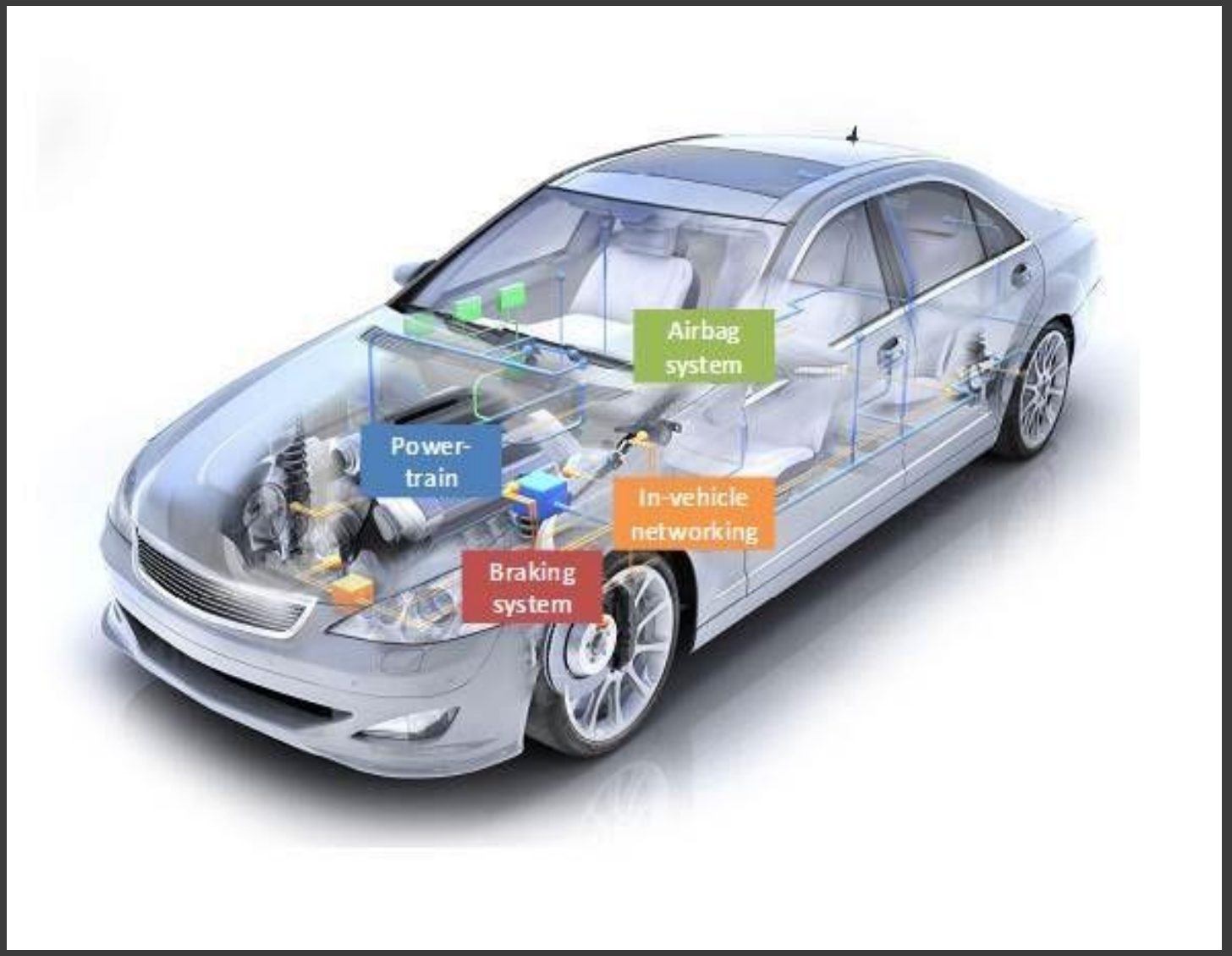
Microcontroller

- CPU, RAM, ROM, I/O and timer are all on a single chip
- Fixed amount of on-chip ROM, RAM, I/O ports
- Not Expansive
- Single-purpose
- Special Purpose.

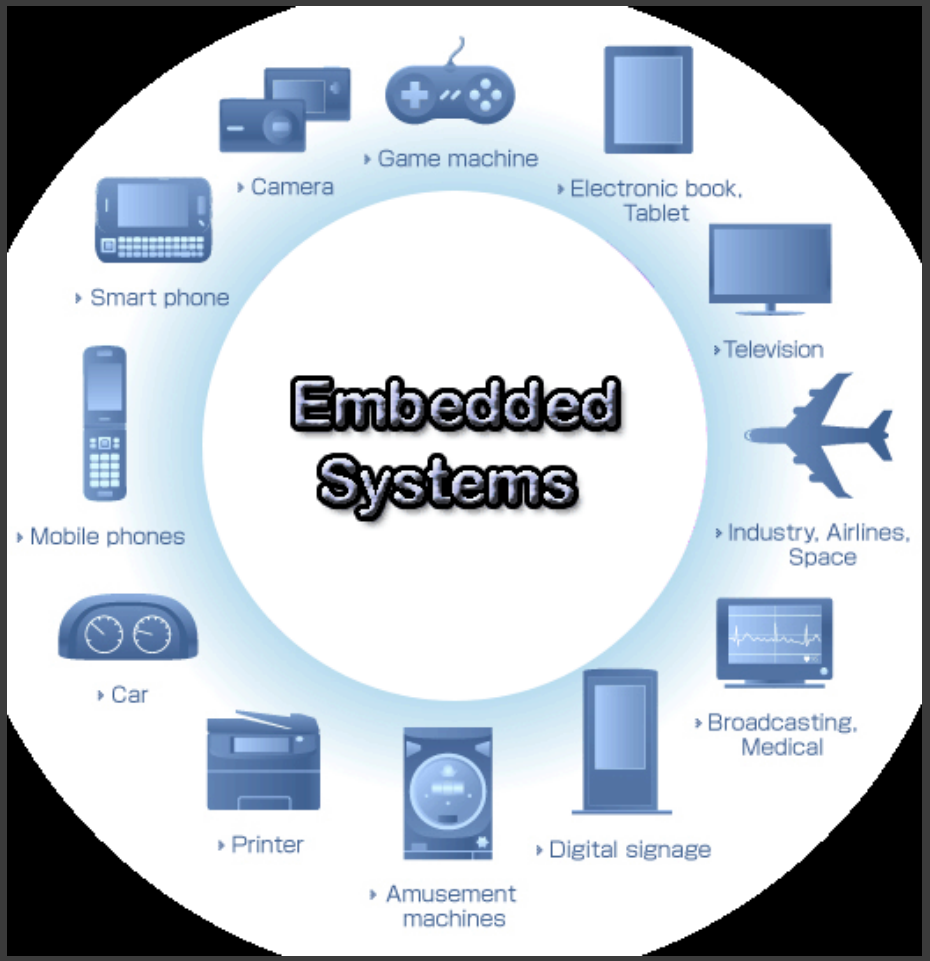
μC based Embedded Systems

- Special purpose computer system usually completely inside the device it controls
- Has specific requirements and performs pre-defined tasks
- Cost reduction compared to general purpose processor
- Different design criteria
  - Performance
  - Reliability
  - Availability
  - Safety

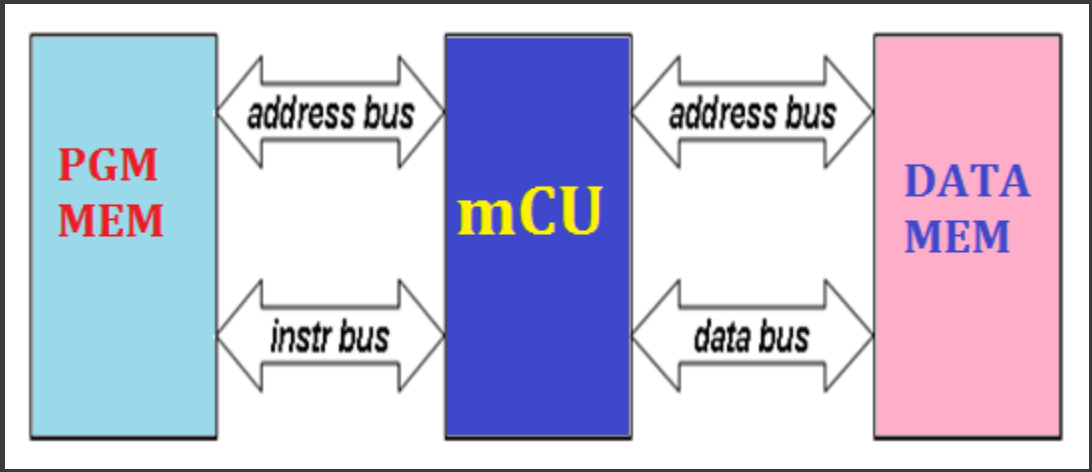
Embedded Systems Examples



Examples


Embedded Systems
Camera
Game machine
Electronic book, Tablet
Smart phone
Television
Mobile phones
Industry, Airlines, Space
Car
Broadcasting, Medical
Printer
Digital signage
Amusement machines

Harvard Architecture



In Harvard Architecture the data and instructions are stored in separate memory units each with their own bus.

Advantages:

- Speeding up the data transfer rate,
- Permits the designer to implement different bus widths and word sizes for program and data memory space.

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 became widely popular after allowing other manufactures to make and market any flavor of the 8051.

Features of 8051

8 bit Processor

4KB Internal ROM

128 Bytes Internal RAM

Four 8 BIT I/O PORTS (32 I/O LINES)

Two 16 Bit Timers/Counters

On Chip Full Duplex UART for Serial Communication

5 Vector Interrupts (2 External, 3 Internal - Timer0,Timer1,Serial)

On Chip Clock Oscillator

16 bit Address bus

- 64k External Code Memory
- 64k External Data Memory

16-bit program counter to access external Code Memory and

16 bit Data Pointer to access external Data Memory

128 user defined flags

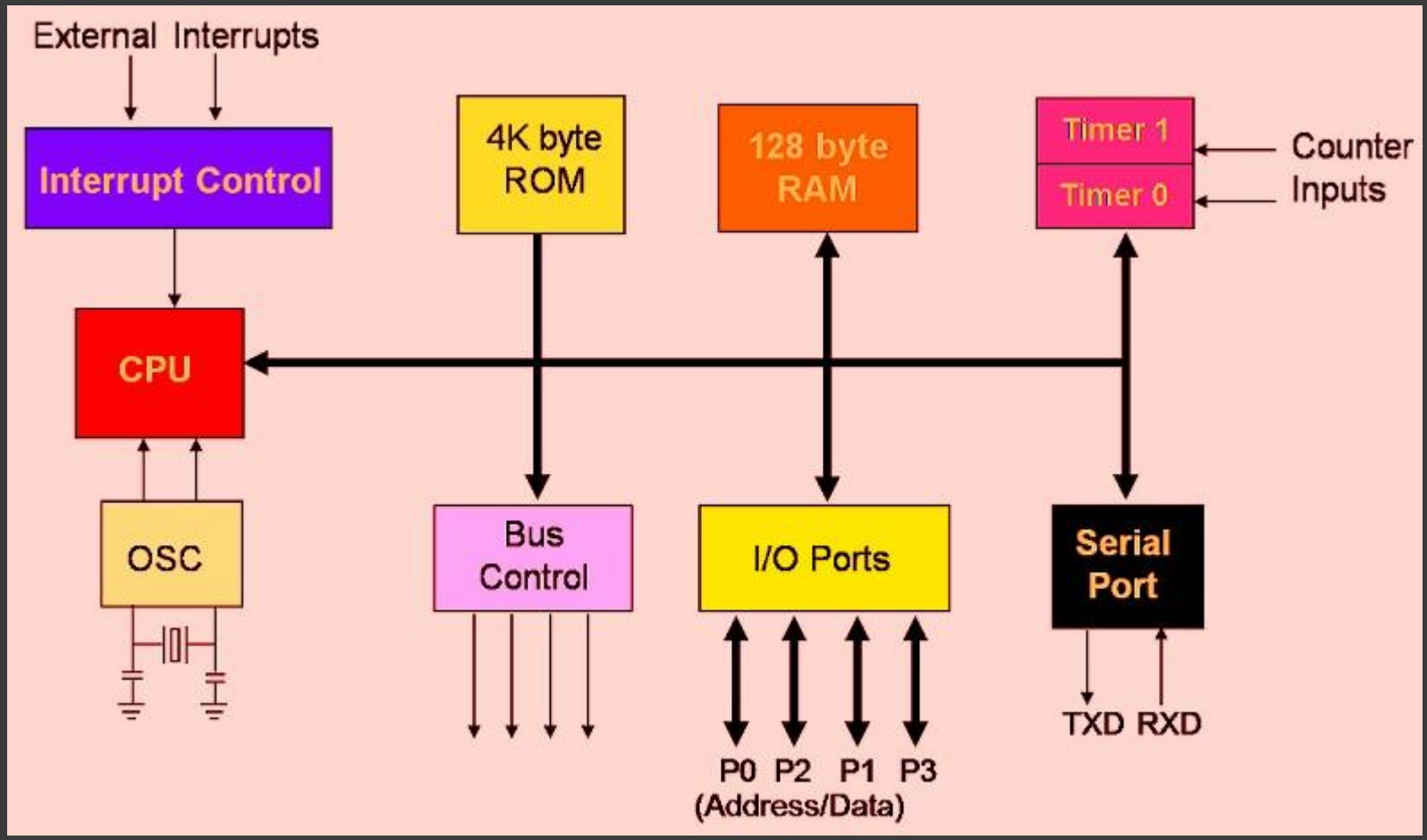
32 General Purpose Registers each of 8 bits

8051 Family

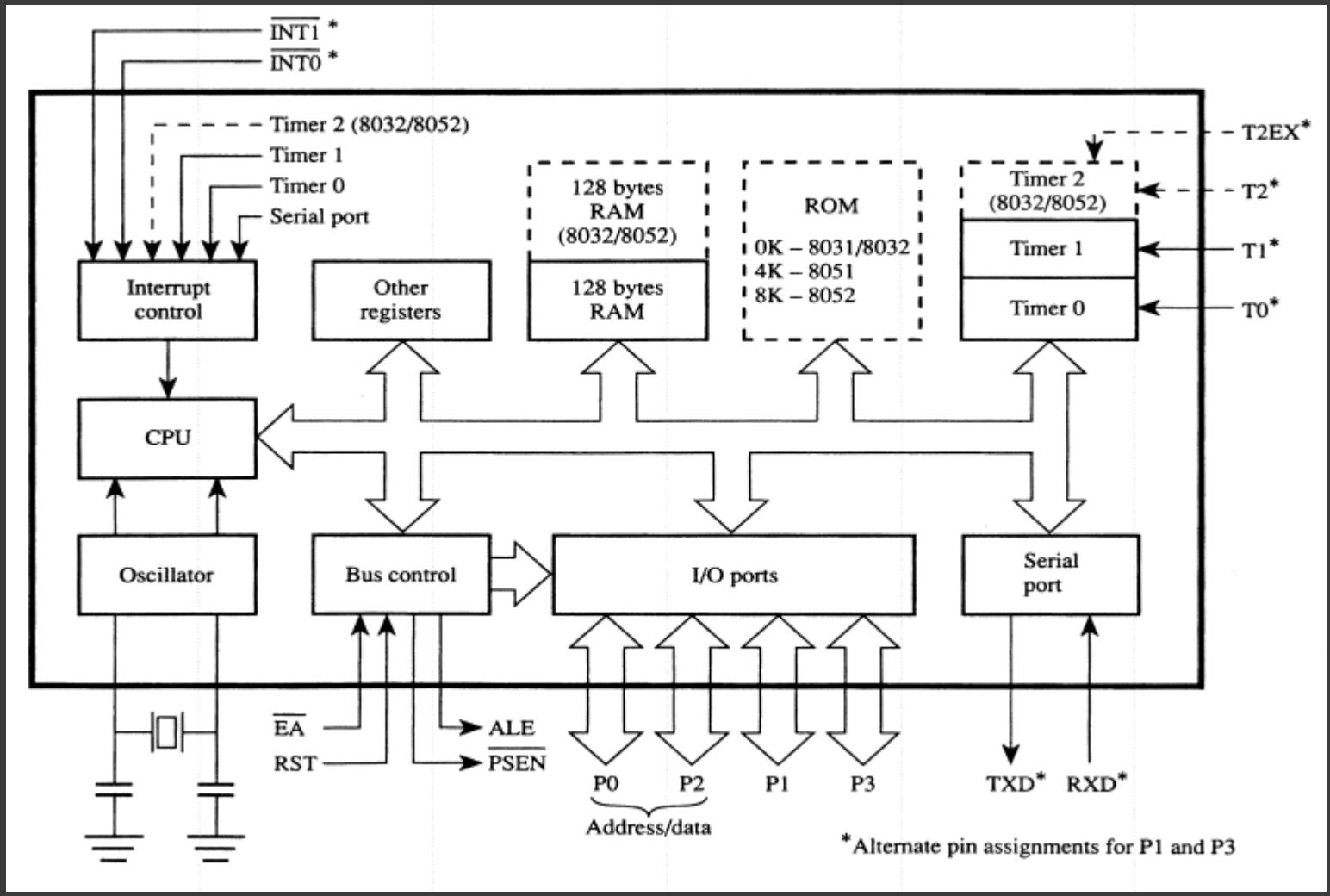
- The 8051 is a subset of the 8052
- The 8031 is a ROM-less 8051
  - Add external ROM to it
  - You lose two ports, and leave only 2 ports for I/O operations

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

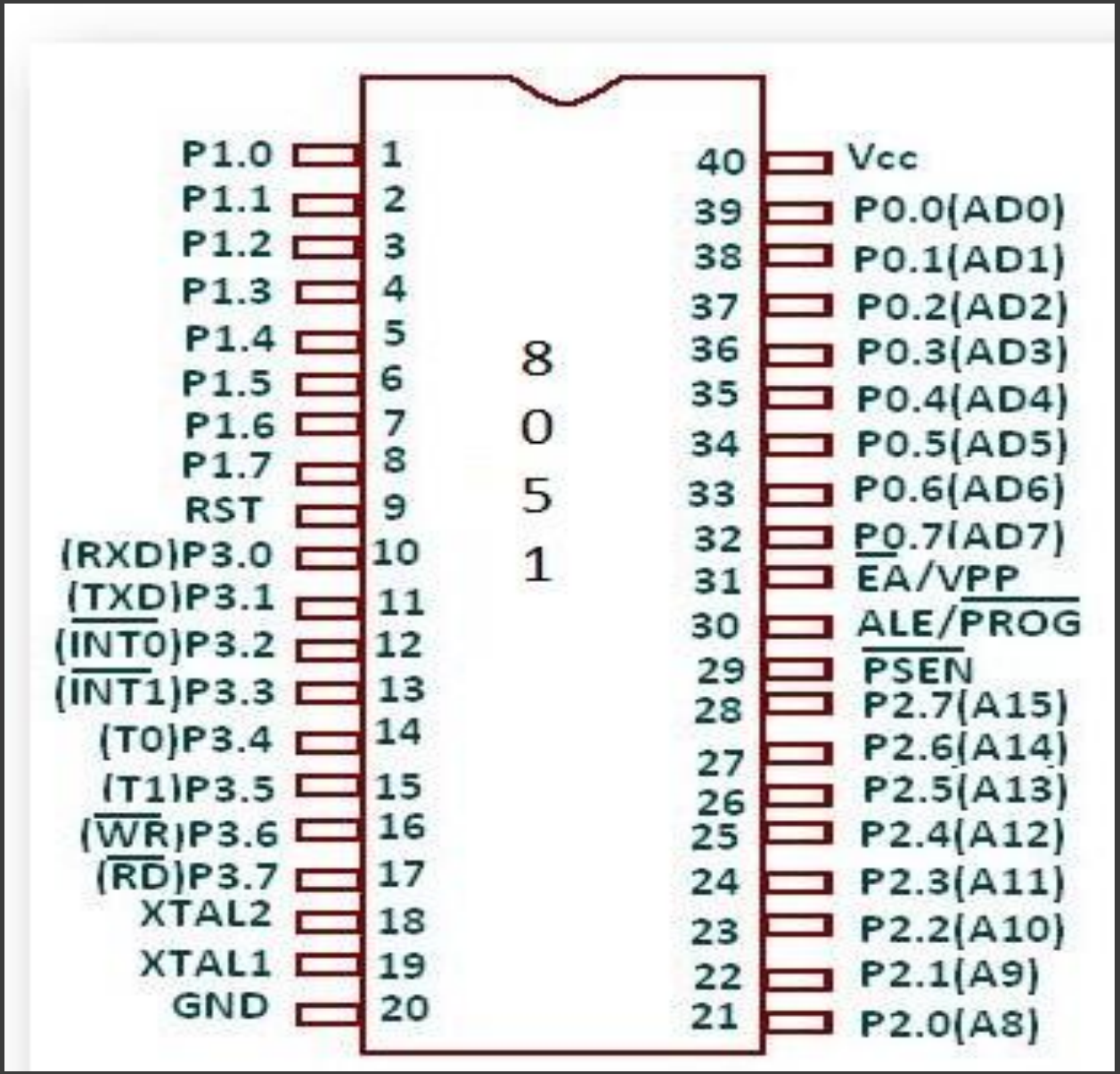
Block Diagram of 8051



Detailed Block Diagram of 8051



Pin Diagram

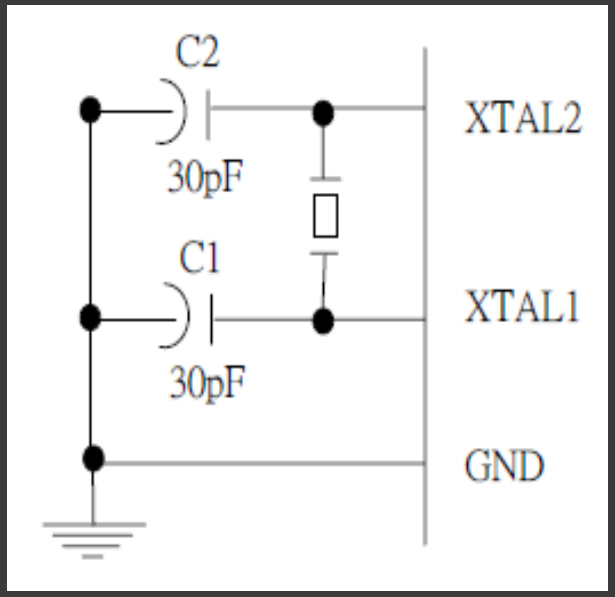

P1.0 1
P1.1 2
P1.2 3
P1.3 4
P1.4 5
P1.5 6
P1.6 7
P1.7 8
RST 9
(RXD)P3.0 10
(TXD)P3.1 11
(INT0)P3.2 12
(INT1)P3.3 13
(T0)P3.4 14
(T1)P3.5 15
(WR)P3.6 16
(RD)P3.7 17
XTAL2 18
XTAL1 19
GND 20
8051
40 Vcc
39 P0.0(AD0)
38 P0.1(AD1)
37 P0.2(AD2)
36 P0.3(AD3)
35 P0.4(AD4)
34 P0.5(AD5)
33 P0.6(AD6)
32 P0.7(AD7)
31 EA/VPP
30 ALE/PROG
29 PSEN
28 P2.7(A15)
27 P2.6(A14)
26 P2.5(A13)
25 P2.4(A12)
24 P2.3(A11)
23 P2.2(A10)
22 P2.1(A9)
21 P2.0(A8)

Pin Description of the 8051

- 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

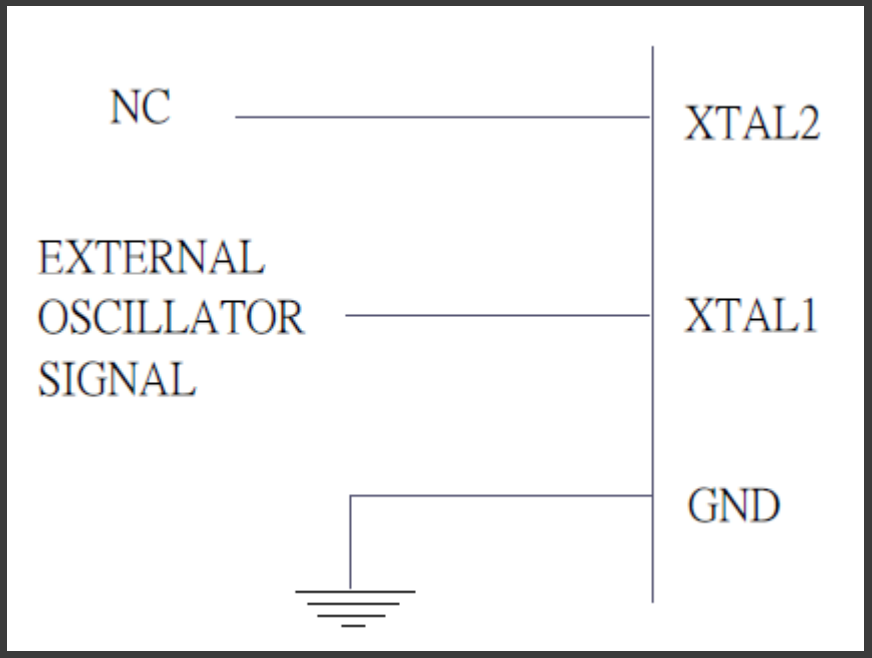
XTAL1 and XTAL2

- The 8051 has an on-chip oscillator but requires an external crystal 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
  - The original 8051 operates at 12 MHZ



XTAL1 and XTAL2 .....

- 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



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

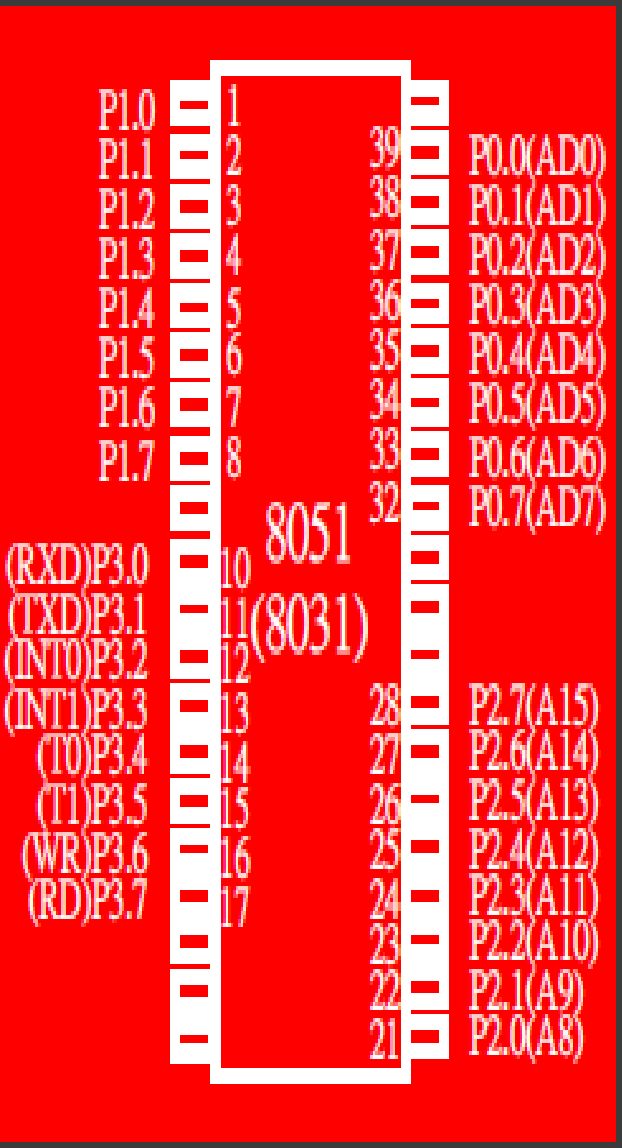
EA’

- 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 and also have an external code and data memory.
- Normally EA pin is connected to Vcc (Internal Access)
- EA pin must be connected to GND to indicate that the code or data is stored externally.

PSEN’ and ALE

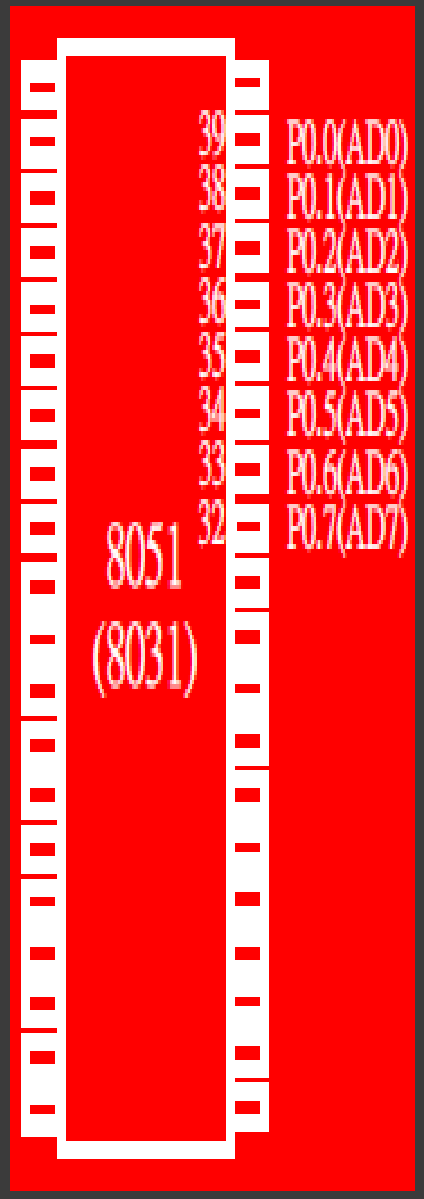
- PSEN, **“program store enable”**, is an output pin
- This pin is connected to the OE pin of the external memory.
- For External Code Memory, PSEN’ = 0
- For External Data Memory, PSEN’ = 1
- ALE pin is used for demultiplexing the address and data.

I/O Port Pins



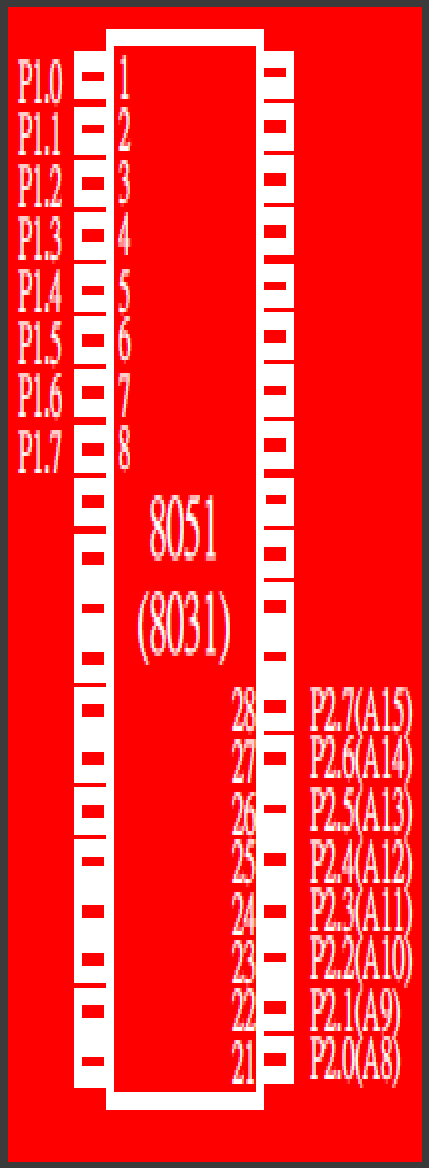
- The four 8-bit I/O ports **P0, P1, P2 and P3** each uses 8 pins.
- All the ports upon RESET are configured as output, ready to be used as input ports by the external device.

Port 0



- Port 0 is **also** designated as **AD0-AD7**.
- When connecting an 8051 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*

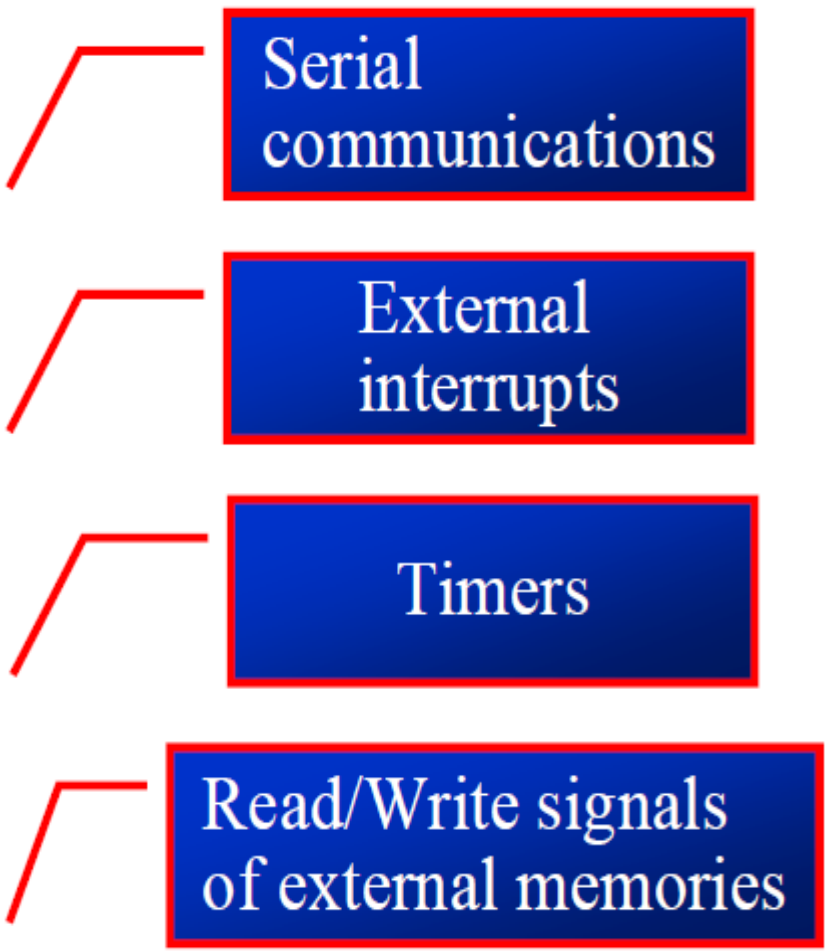
Port 1 and Port 2



- In 8051-based systems **with no external memory connection**:
  - Both P1 and P2 are used as simple I/O.
- In 8051-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.

Port 3

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



Pin Description Summary

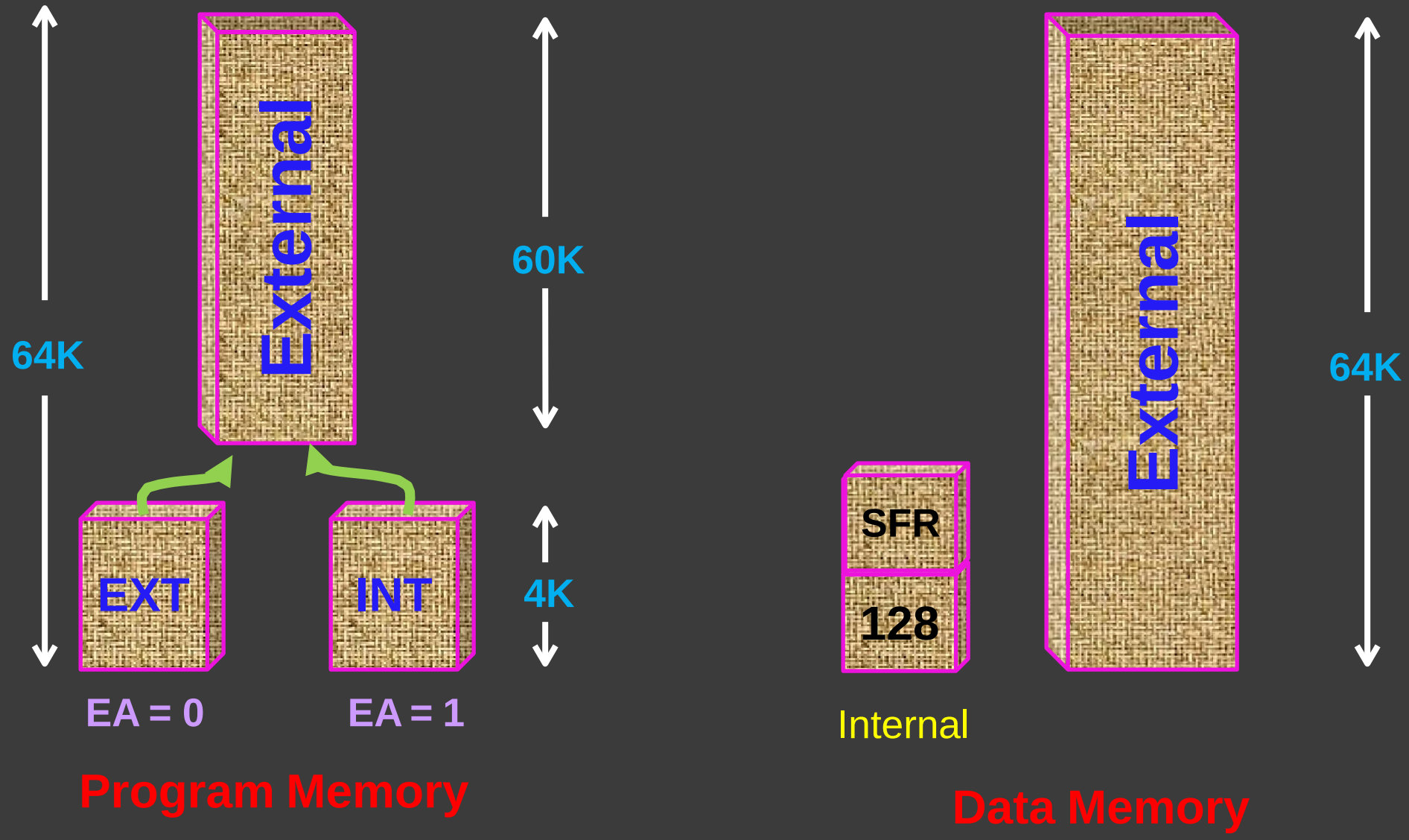
PIN	TYPE	NAME AND FUNCTION
Vss	**I**	**Ground:** 0 V reference.
Vcc	**I**	**Power Supply:** This is the power supply voltage for normal, idle, and power-down operation.
P0.0 - P0.7	**I/O**	**Port 0:** Port 0 is an open-drain, bi-directional I/O port. Port 0 is also the multiplexed low-order address and data bus during accesses to external program and data memory.
P1.0 - P1.7	**I/O**	**Port 1:** Port I is an 8-bit bi-directional I/O port.
P2.0 - P2.7	**I/O**	**Port 2:** Port 2 is an 8-bit bidirectional I/O. Port 2 emits the high order address byte during fetches from external program memory and during accesses to external data memory that use 16 bit addresses.
P3.0 - P3.7	**I/O**	**Port 3:** Port 3 is an 8 bit bidirectional I/O port. Port 3 also serves special features as explained.

Pin Description Summary

PIN	TYPE	NAME AND FUNCTION
RST	**I**	**Reset:** A high on this pin for two machine cycles while the oscillator is running, resets the device.
ALE	**O**	**Address Latch Enable:** Output pulse for latching the low byte of the address during an access to external memory.
PSEN*	**O**	**Program Store Enable:** The read strobe to external program memory. When executing code from the external program memory, PSEN* is activated twice each machine cycle, except that two PSEN* activations are skipped during each access to external data memory.
EA*/VPP	**I**	**External Access Enable/Programming Supply Voltage:** EA* must be externally held low to enable the device to fetch code from external program memory locations. If EA* Is held high, the device executes from internal program memory. This pin also receives the programming supply voltage Vpp during Flash programming. (applies for 89c5x MCU's)

8051 Memory Organisation

8051 Memory Structure

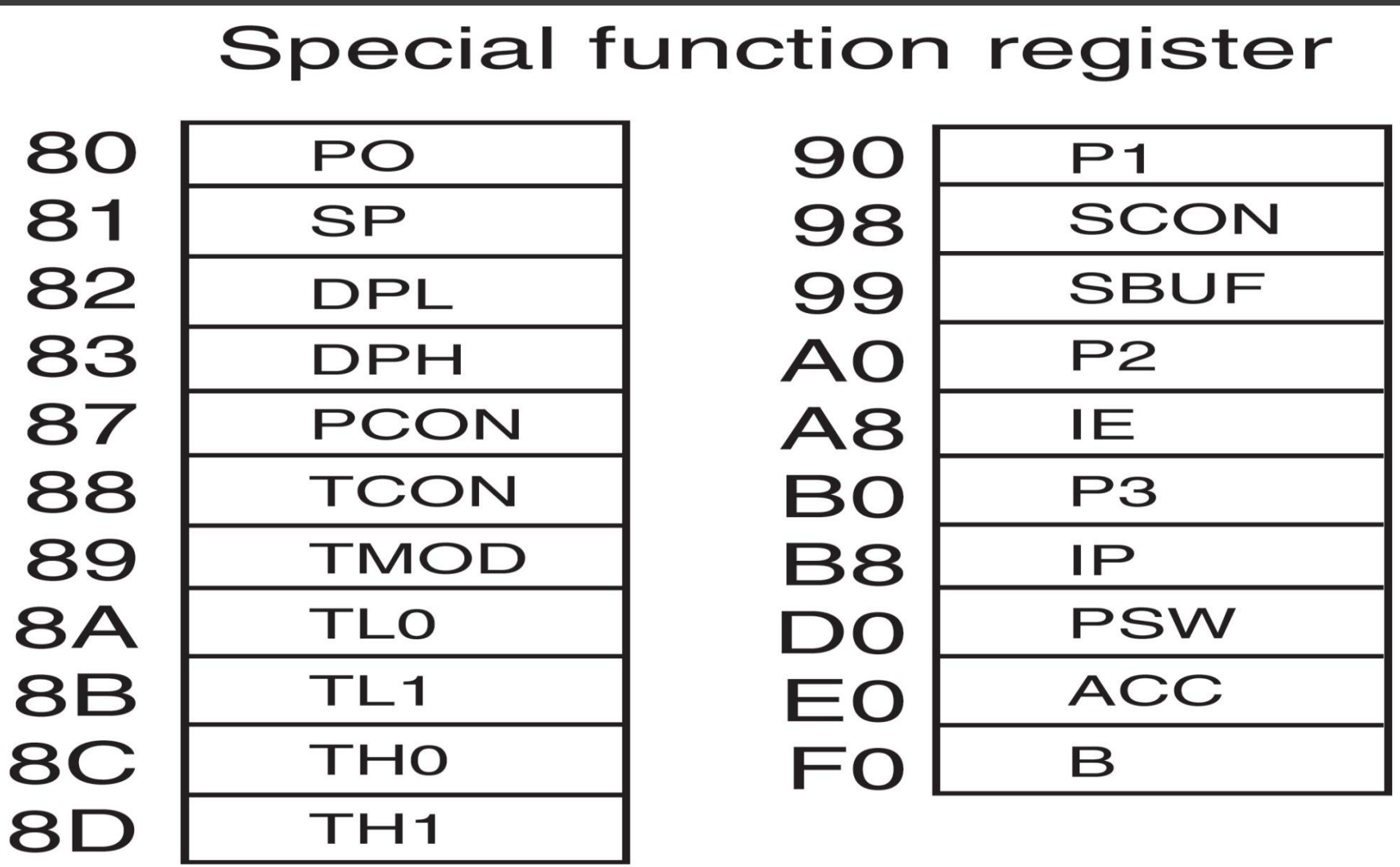


Internal RAM Structure

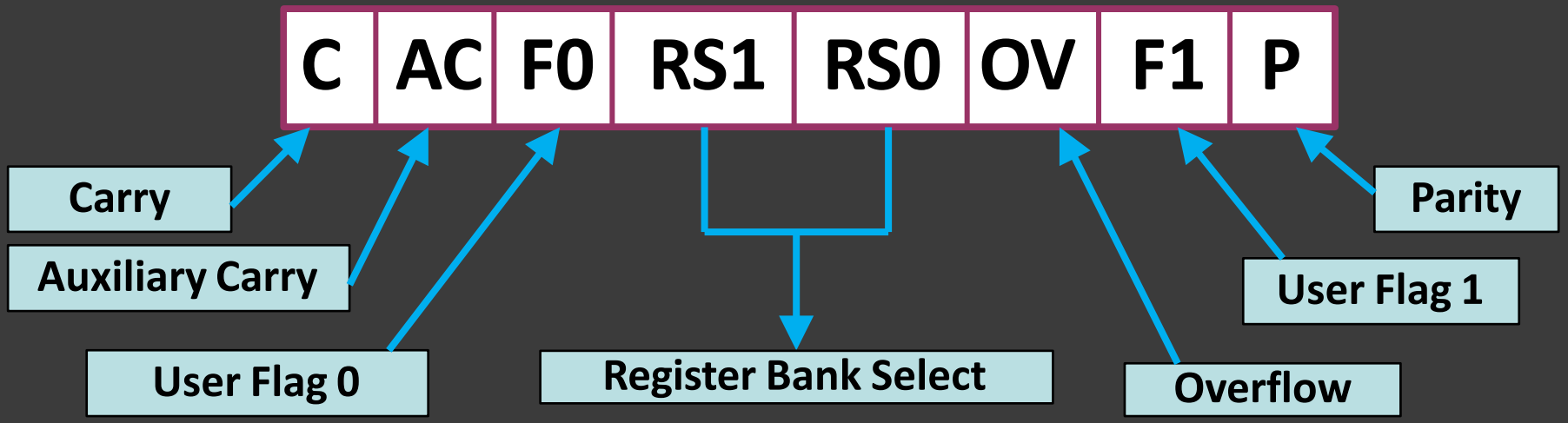
SFR [Special Function Registers]

128 Byte Internal RAM

Special Function Registers [SFR]



Program Status Word [PSW]



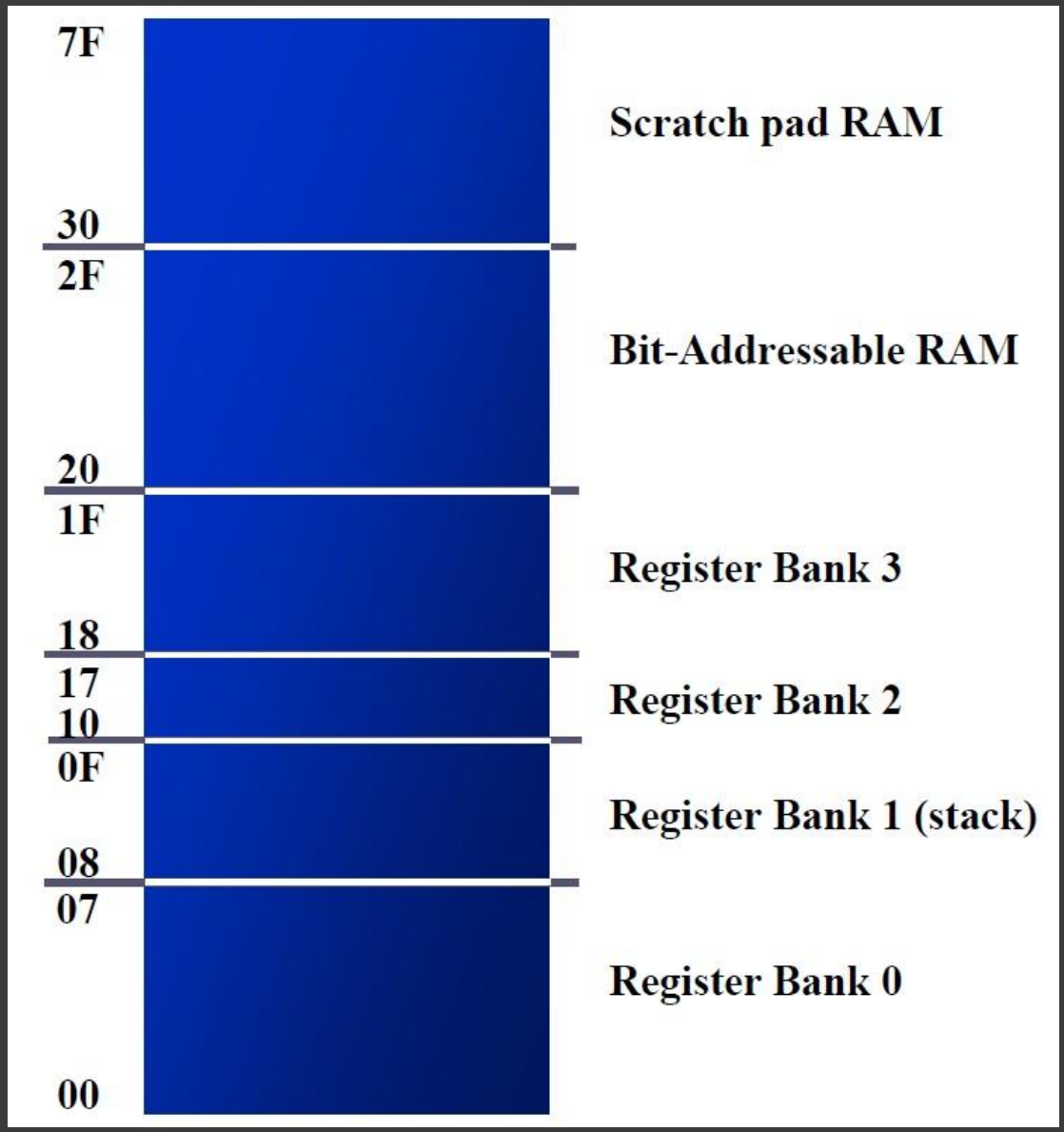
8051 instructions that affects flag

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		

128 Byte RAM

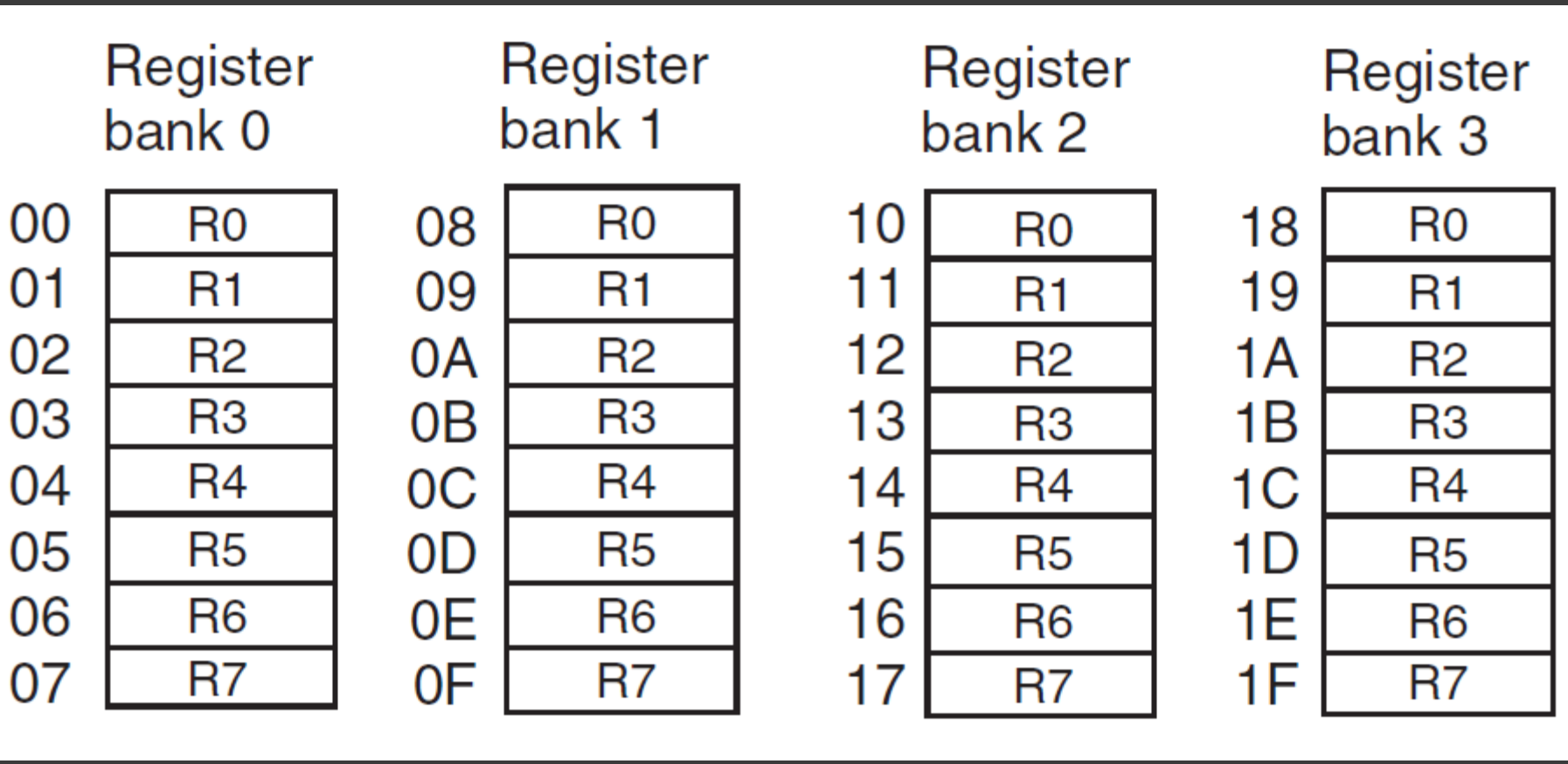
- There are 128 bytes of RAM in the 8051.
  - Assigned addresses 00 to 7FH
- The **128 bytes are divided into 3 different groups** as follows:
  1. A total of **32 bytes** from locations 00 to 1F hex are set aside for ***register banks*** and the ***stack.***
  2. A total of **16 bytes** from locations 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 RAM with addresses



8051 Register Bank Structure

8051 Register Banks with address



8051 Stack

- The **stack** is a section of RAM used by the CPU to store information **temporarily.**
  - This information could be data or an address

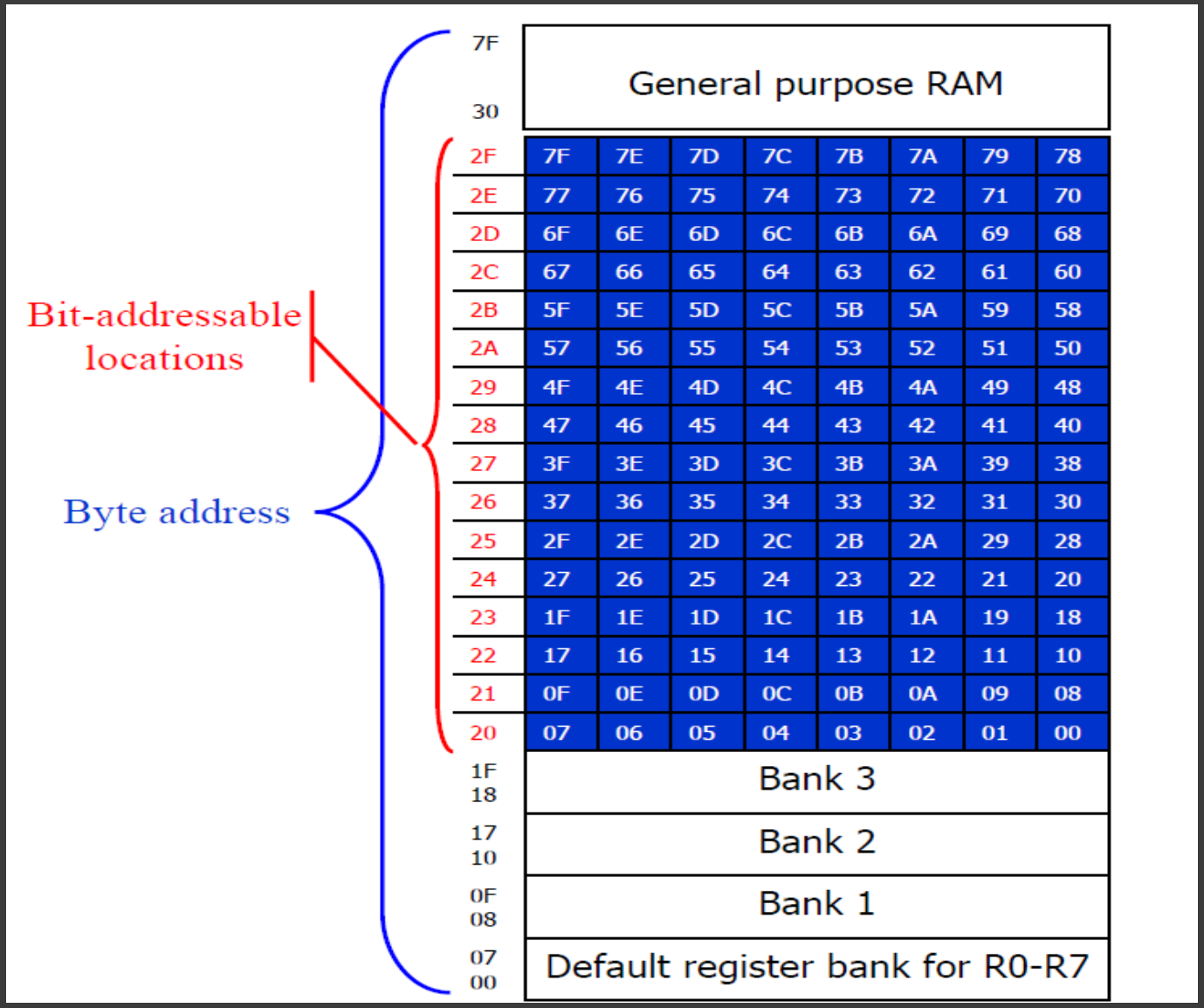
- The register used to access the stack is called the **SP (stack pointer) register**
  - The stack pointer in the 8051 is **only 8 bit wide**, which means that it can take value of 00 to FFH
  - When the 8051 is powered up, the SP register contains value 07
  - RAM location 08 is the first location begin used for the stack by the 8051

8051 Stack

- 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

Bit Addressable & Byte Addressable



Bit Addressable Programming

- **Example:** 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

Chapter 2
Instruction Set

1. Addressing Modes
2. Instruction Set
3. Types of Instructions

8051 Addressing Modes

- The CPU can access data in various ways, which are called addressing modes

1. **Immediate**
2. **Register**
3. **Direct**
4. **Register indirect**
5. **External Direct**

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
```

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

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.
- Contrast this with immediate addressing mode, there is no “#” sign in the operand.

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

SFR Registers & their Addresses

```
MOV   0E0H,#55H     ;is the same as
MOV   A,#55H        ;which means load 55H into A (A=55H)

MOV   0F0H,#25H     ;is the same as
MOV    B,#25H       ;which means load 25H into B (B=25H)

MOV   0E0H,R2       ;is the same as
MOV   A,R2          ;which means copy R2 into A

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

Stack and Direct Addressing Mode

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
```

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
```

Register Indirect Addressing Mode

- 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
```

Register Indirect Addressing Mode

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

```
        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
```

External Direct

- External Memory is accessed.
- There are only two commands that use External Direct addressing mode:
  - **MOVX A, @DPTR**
    MOVX @DPTR, A
- DPTR must first be loaded with the address of external memory.

8051 Instruction Set

- 8051 instructions have 8-bit opcode
- There are 256 possible instructions of which 255 are implemented

8051 Instruction Format

- immediate addressing


```
add a,#3dh      ;machine code=243d
```

- Direct addressing


```
mov r3,0E8h     ;machine code=ABE8
```

8051 Instruction Format

- Register addressing


```
070D E8      mov a,r0        ;E8 = 1110 1000
070E E9      mov a,r1        ;E9 = 1110 1001
070F EA      mov a,r2        ;EA = 1110 1010
0710 ED      mov a,r5        ;ED = 1110 1101
0711 EF      mov a,r7        ;Ef = 1110 1111
0712 2F      add a,r7
0713 F8      mov r0,a
0714 F9      mov r1,a
0715 FA      mov r2,a
0716 FD      mov r5,a
0717 FD      mov r5,a
```

8051 Instruction Format

- Register indirect addressing

mov a, @Ri ; i = 0 or 1

```
070D E7      mov  a,@r1
070D 93      movc a,@a+dptr
070E 83      movc a,@a+pc
070F E0      movx a,@dptr
0710 F0      movx @dptr,a
0711 F2      movx @r0,a
0712 E3      movx a,@r1
```

8051 Instruction Format

- relative addressing


```
here:  sjmp here    ;machine code=80FE(FE=-2)
Range = (-128 ~ 127)
```

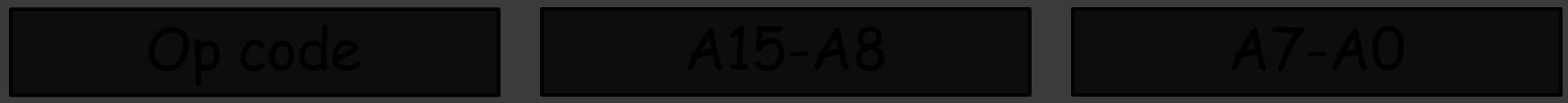
- Absolute addressing (limited in 2k current mem block)


```
0700                1       org 0700h
0700 E106           2       ajmp next    ;next=706h
0702 00             3       nop
0703 00             4       nop
0704 00             5       nop
0705 00             6       nop
                    7   next:
                    8       end
```

07FFh

8051 Instruction Format

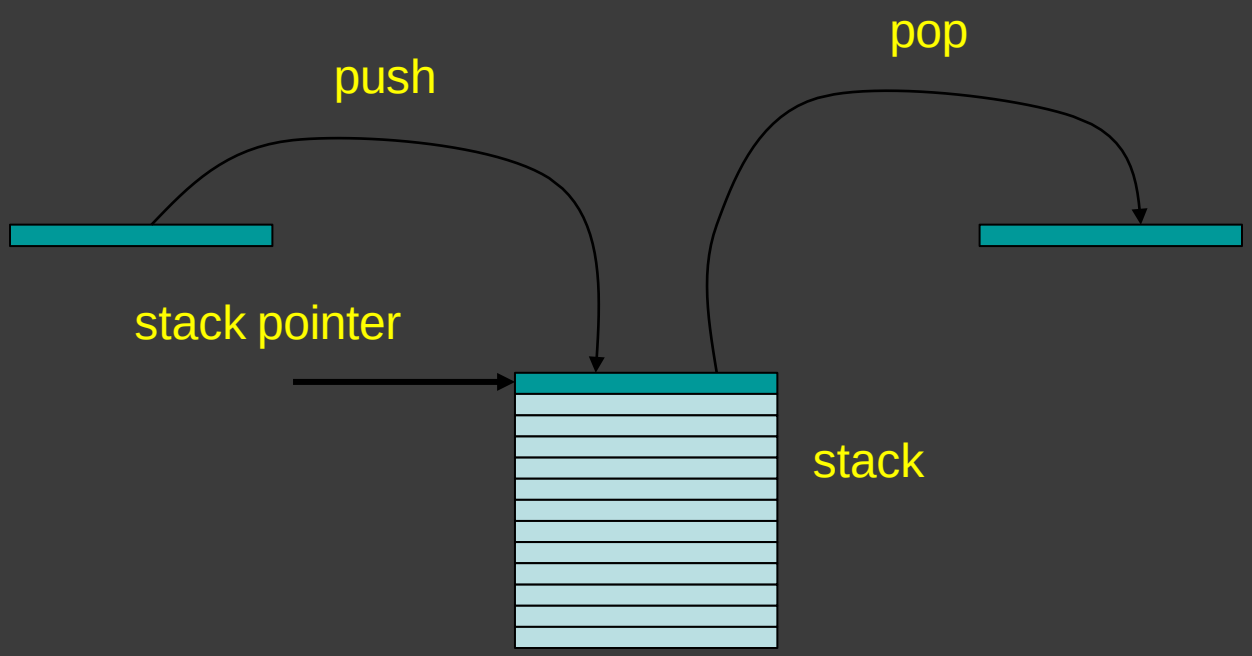
- Long distance address



Range = (0000h ~ FFFFh)

```
0700                    1           org 0700h
0700 020707             2           ajmp next    ;next=0707h
0703 00                 3           nop
0704 00                 4           nop
0705 00                 5           nop
0706 00                 6           nop
                        7     next:
                        8           end
```

Stacks



Go do the stack exercise.....

Stack

- Stack-oriented data transfer
  - Only one operand (direct addressing)
  - SP is other operand – register indirect - implied
- Direct addressing mode must be used in Push and Pop

```
mov sp, #0x40   ; Initialize SP
push 0x55       ; SP ← SP+1, M[SP] ← M[55]
                ; M[41] ← M[55]
pop b           ; b ← M[55]
```

Note: can only specify RAM or SFRs (direct mode) to push or pop. Therefore, to push/pop the accumulator, must use acc, not a

Stack (push,pop)

- Therefore

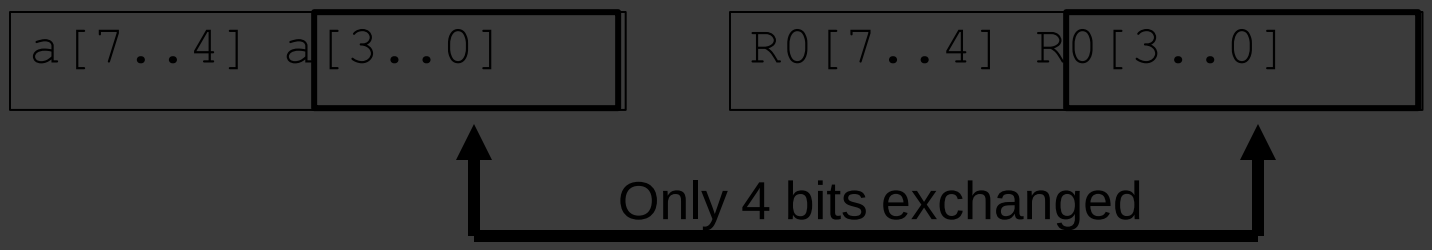
```
Push a      ;is invalid
Push r0     ;is invalid
Push r1     ;is invalid
```

```
push acc    ;is correct
Push psw    ;is correct
Push b      ;is correct
Push 13h
Push 0
Push 1
Pop  7
Pop  8
Push 0e0h   ;acc
Pop  0f0h   ;b
```

Exchange Instructions

two way data transfer

```
XCH a, 30h          ; a ←→ M[30]
XCH a, R0           ; a ←→ R0
XCH a, @R0          ; a ←→ M[R0]
XCHD a, R0          ; exchange "digit"
```



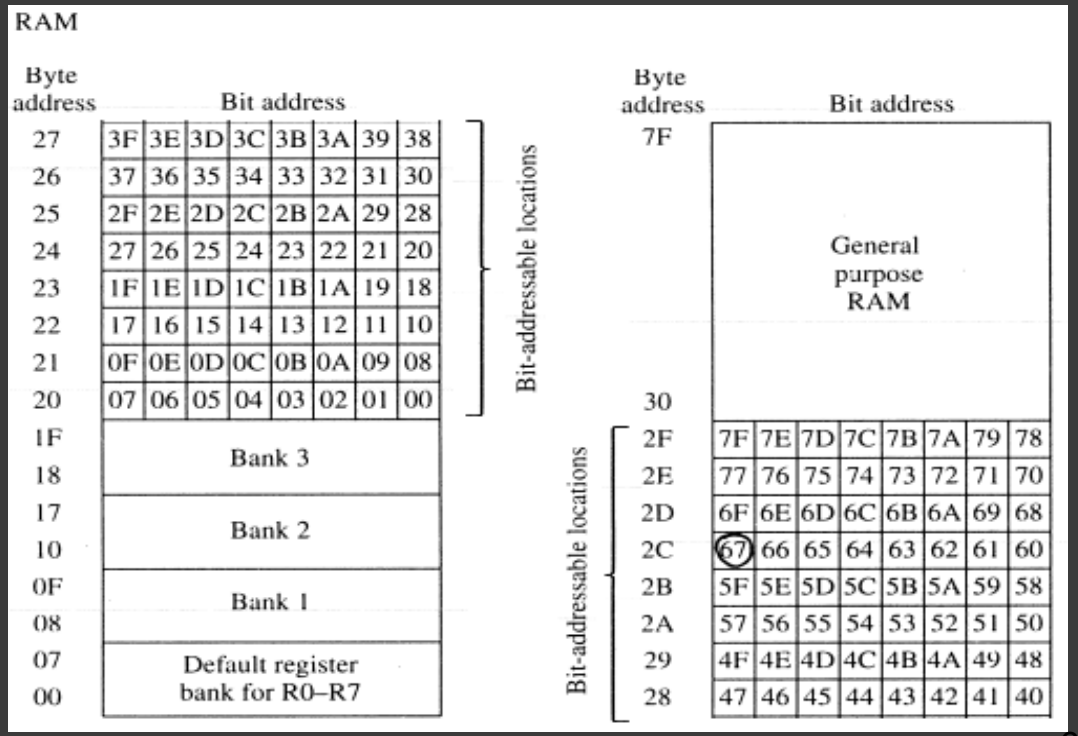
Bit-Oriented Data Transfer

- transfers between individual bits.
- Carry flag (C) (bit 7 in the PSW) is used as a single-bit accumulator
- RAM bits in addresses 20-2F are bit addressable

```
mov C, P0.0

mov C, 67h

mov C, 2ch.7
```

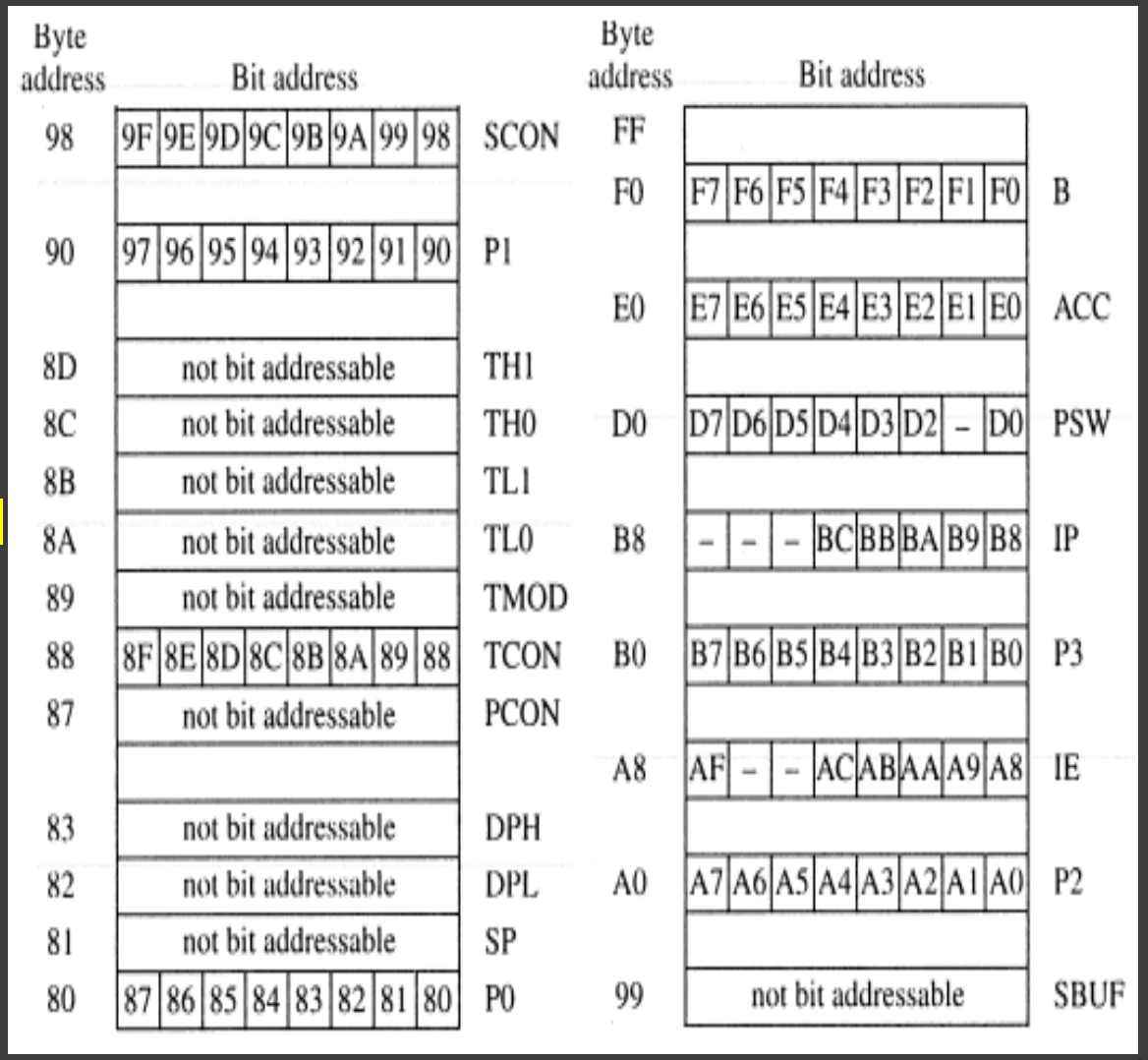


SFRs that are Bit Addressable

SFRs with addresses ending in 0 or 8 are bit-addressable.

(80, 88, 90, 98, etc)

Notice that all 4 parallel I/O ports are bit addressable.



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

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	–	D0	PSW
B8	–	–	–	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	–	–	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	not bit addressable								SBUF

Data Processing Instructions

Arithmetic Instructions

Logic Instructions

Arithmetic Instructions

- Add
- Subtract
- Increment
- Decrement
- Multiply
- Divide
- Decimal adjust

Arithmetic Instructions

Mnemonic	Description
`ADD A, byte`	add *A* to byte, put result in *A*
`ADDC A, byte`	add with carry
`SUBB A, byte`	subtract with borrow
`INC A`	increment *A*
`INC byte`	increment byte in memory
`INC DPTR`	increment data pointer
`DEC A`	decrement accumulator
`DEC byte`	decrement byte
`MUL AB`	multiply accumulator by b register
`DIV AB`	divide accumulator by b register
`DA A`	decimal adjust the accumulator

ADD Instructions

```
add a, byte          ; a ← a + byte
addc a, byte               ; a ← a + byte + C
```

These instructions affect 3 bits in PSW:

C = 1 if result of add is greater than FF

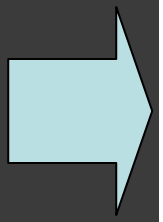
AC = 1 if there is a carry out of bit 3

OV = 1 if there is a carry out of bit 7, but not from bit 6, or visa versa.

Program Status Word (PSW)

Bit	7	6	5	4	3	2	1	0
Flag	**CY**	**AC**	**F0**	**RS1**	**RS0**	**OV**	**F1**	**P**
Name	Carry Flag	Auxiliary Carry Flag	User Flag 0	Register Bank Select 1	Register Bank Select 0	Overflow flag	User Flag 1	Parity Bit

Instructions that Affect PSW bits



Instructions that Affect Flag Settings(1)

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
ADD	X	X	X	CLR C	0		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	0	X		ANL C,/bit	X		
DIV	0	X		ORL C,bit	X		
DA	X			ORL C,/bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						

ADD Examples

```
mov a, #3Fh
add a, #D3h
```

- What is the value of the C, AC, OV flags after the second instruction is executed?

```
  0011 1111
  1101 0011
  ---------
  0001 0010
```

```
C = 1
AC = 1
OV = 0
```

Signed Addition and Overflow

2's complement:

```
0000 0000   00    0
…
0111 1111   7F     127
1000 0000    | 80 |-128
…
1111 1111    | FF |-1
```

```
 0111 1111   (positive 127)
 0111 0011   (positive 115)
 1111 0010   (overflow
cannot represent 242 in 8
bits 2's complement)
```

```
 1000 1111   (negative 113)
 1101 0011   (negative  45)
 0110 0010   (overflow)
```

```
0011 1111   (positive)
1101 0011   (negative)
0001 0010 (never overflows)
```

Addition Example

```
; Computes Z = X + Y
; Adds values at locations 78h and 79h and puts them in 7Ah
;_____________________________________________
X       equ     78h
Y       equ     79h
Z       equ     7Ah
; _____________________________________________
        org 00h
        ljmp Main
; _____________________________________________
        org 100h
Main:
        mov a, X
        add  a,  Y
        mov Z, a
        end
```

The 16-bit ADD example

```
; Computes Z = X + Y     (X,Y,Z are 16 bit)
;_____________________________________________
X           equ         78h
Y           equ         7Ah
Z           equ         7Ch
; _____________________________________________
            org 00h
            ljmp Main
; _____________________________________________
            org 100h
Main:
            mov a, X
            add a, Y
            mov Z, a
            mov a, X+1
            adc a, Y+1
            mov Z+1, a
            end
```

Subtract

SUBB A, byte	subtract with borrow

Example:

```
SUBB A, #0x4F     ;A ← A – 4F – C
```

Notice that
There is no subtraction WITHOUT borrow.
Therefore, if a subtraction without borrow is desired,
it is necessary to clear the C flag.

Example:

```
Clr  c
SUBB A, #0x4F     ;A ← A – 4F
```

Increment and Decrement

INC A	increment A
INC byte	increment byte in memory
INC DPTR	increment data pointer
DEC A	decrement accumulator
DEC byte	decrement byte

- The increment and decrement instructions do NOT affect the C flag.
- Notice we can only INCREMENT the data pointer, not decrement.

Example: Increment 16-bit Word

- Assume 16-bit word in R3:R2

```
mov a, r2
add a, #1 ; use add rather than increment to affect C
mov r2, a
mov a, r3
addc a, #0              ; add C to most significant byte
mov r3, a
```

Multiply

When multiplying two 8-bit numbers, the size of the maximum product is 16-bits

FF x FF = FE01

(255 x 255 = 65025)

```
MUL AB          ; BA ← A * B
```

Note : B gets the High byte
A gets the Low byte

Division

- Integer Division

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

✌👉← Quotient(A/B)
👌👉← Remainder(A/B)

OV - used to indicate a divide by zero condition.
C – set to zero

Decimal Adjust

```
DA a    ; decimal adjust a
```

Used to facilitate BCD addition.
Adds “6” to either high or low nibble after an addition
to create a valid BCD number.

Example:

```
mov a, #23h
mov b, #29h
add a, b          ; a ← 23h + 29h = 4Ch (wanted 52)
DA a              ; a ← a + 6 = 52
```

Logic Instructions

- Bitwise logic operations
  - (AND, OR, XOR, NOT)
- Clear
- Rotate
- Swap

Logic instructions do NOT affect the flags in PSW

Bitwise Logic

ANL → AND

ORL → OR

XRL → XOR

CPL → Complement

Examples:

```
  00001111
ANL 10101100
  00001100

  00001111
ORL 10101100
  10101111

  00001111
XRL 10101100
  10100011

CPL 10101100
  01010011
```

Address Modes with Logic

ANL – AND

ORL – OR

XRL – eXclusive oR

a, byte

direct, reg. indirect, reg, immediate

byte, a

direct

byte, #constant

CPL – Complement

a ex: cpl a

Uses of Logic Instructions

- Force individual bits low, without affecting other bits.

```
anl PSW, #0xE7          ;PSW AND 11100111
```

- Force individual bits high.

```
orl PSW, #0x18          ;PSW OR 00011000
```

- Complement individual bits

```
xrl P1, #0x40                   ;P1 XRL 01000000
```

Other Logic Instructions

CLR - clear

RL - rotate left

RLC - rotate left through Carry

RR - rotate right

RRC - rotate right through Carry

SWAP - swap accumulator nibbles

CLR (Set all bits to 0)

CLR A

CLR byte (direct mode)

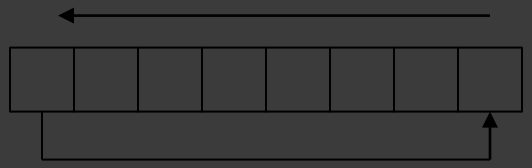
CLR Ri (register mode)

CLR @Ri (register indirect mode)

Rotate

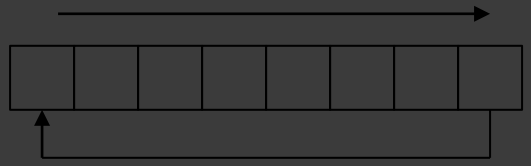
- Rotate instructions operate only on **a**

```
RL a
```



```
Mov a,#0xF0      ; a← 11110000
RR a             ; a← 11100001
```

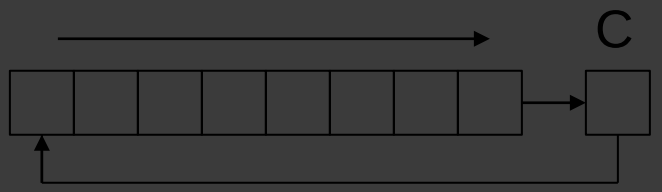
```
RR a
```



```
Mov a,#0xF0      ; a← 11110000
RR a             ; a← 01111000
```

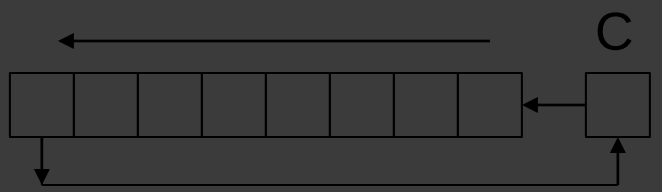
Rotate through Carry

RRC a




```
mov a, #0A9h        ; a ← A9
add a, #14h         ; a ← BD (10111101), C←0
rrc a               ; a ←  01011110, C←1
```

RLC a




```
mov a, #3ch         ; a ← 3ch(00111100)
setb c              ; c ← 1
rlc a               ; a ←  01111001, C←1
```

Rotate and Multiplication/Division

- Note that a shift left is the same as multiplying by 2, shift right is divide by 2

```
mov a, #3     ; A← 00000011 (3)
clr C         ; C← 0
rlc a         ; A← 00000110 (6)
rlc a         ; A← 00001100 (12)
rrc a         ; A← 00000110 (6)
```

Swap

SWAP a

```
mov a, #72h          ; a ← 27h
swap a               ; a ← 27h
```

Bit Logic Operations

- Some logic operations can be used with single bit operands

```
ANL C, bit
ORL C, bit
CLR C
CLR bit
CPL C
CPL bit
SETB C
SETB bit
```

- “bit” can be any of the bit-addressable RAM locations or SFRs.

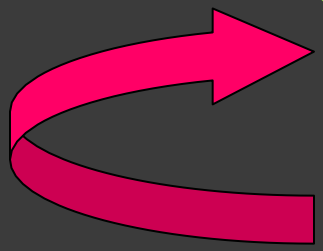
Program Flow Control

- Unconditional jumps (“go to”)
- Conditional jumps
- Call and return

Unconditional Jumps

- **SJMP <rel addr>** ; Short jump, relative address is 8-bit 2's complement number, so jump can be up to 127 locations forward, or 128 locations back.
- **LJMP <address 16>** ; Long jump
- **AJMP <address 11>** ; Absolute jump to anywhere within 2K block of program memory
- **JMP @A + DPTR** ; Long indexed jump

Infinite Loops



```
Start: mov C, p3.7
       mov p1.6, C
       sjmp Start
```

Microcontroller application programs are almost always infinite loops!

Re-locatable Code

Memory specific NOT Re-locatable (machine code)

```
        org 8000h
Start:  mov C, p1.6
        mov p3.7, C
        ljmp Start
        end
```

Re-locatable (machine code)

```
        org 8000h
Start:  mov C, p1.6
        mov p3.7, C
        sjmp Start
        end
```

Jump table

```
            Mov dptr,#jump_table
            Mov a,#index_number
            Rl  a
            Jmp @a+dptr
                ...
Jump_table: ajmp case0
            ajmp case1
            ajmp case2
            ajmp case3
```

Conditional Jump

- These instructions cause a jump to occur only if a condition is true. Otherwise, program execution continues with the next instruction.

```
loop: mov a, P1
      jz  loop      ; if a=0, goto loop,
                    ; else goto next instruction
      mov b, a
```

- There is no zero flag (z)
- Content of A checked for zero on time

Conditional jumps

Mnemonic	Description
JZ <rel addr>	Jump if a = 0
JNZ <rel addr>	Jump if a != 0
JC <rel addr>	Jump if C = 1
JNC <rel addr>	Jump if C != 1
JB <bit>, <rel addr>	Jump if bit = 1
JNB <bit>,<rel addr>	Jump if bit != 1
JBC <bir>, <rel addr>	Jump if bit =1, &clear bit
CJNE A, direct, <rel addr>	Compare A and memory, jump if not equal

Example: Conditional Jumps

```
if (a = 0) is true
    send a 0 to LED
else
    send a 1 to LED
```

```
            jz led_off
            Setb P1.6
            sjmp skipover
 led_off:   clr P1.6
            mov A, P0
skipover:
```

More Conditional Jumps

Mnemonic	Description
CJNE A, #data <rel addr>	Compare A and data, jump if not equal
CJNE Rn, #data <rel addr>	Compare Rn and data, jump if not equal
CJNE @Rn, #data <rel addr>	Compare Rn and memory, jump if not equal
DJNZ Rn, <rel addr>	Decrement Rn and then jump if not zero
DJNZ direct, <rel addr>	Decrement memory and then jump if not zero

Iterative Loops

For A = 0 to 4 do

{...}

```
        clr a
loop:   ...
        ...
        inc a
        cjne a, #4, loop
```

For A = 4 to 0 do

{...}

```
        mov R0, #4
loop:   ...
        ...
        djnz R0, loop
```

Iterative Loops(examples)

```
       mov a,#50h
       mov b,#00h
       cjne a,#50h,next
       mov  b,#01h
next:  nop
       end
```

```
       mov a,#25h
       mov r0,#10h
       mov r2,#5
Again: mov @ro,a
       inc r0
       djnz r2,again
       end
```

```
       mov a,#0aah
       mov b,#10h
Back1:mov r6,#50
Back2:cpl a
       djnz r6,back2
       djnz b,back1
       end
```

```
       mov a,#0h
       mov r4,#12h
Back:  add a,#05
       djnz r4,back
       mov  r5,a
       end
```

Call and Return

- Call is similar to a jump, but
  - Call pushes PC on stack before branching

```
acall <address 11>        ; stack ← PC
                          ; PC ← address 11 bit

lcall <address 16>        ; stack ← PC
                          ; PC ← address 16 bit
```

Return

- Return is also similar to a jump, but
  - Return instruction pops PC from stack to get address to jump to

```
ret                    ; PC ← stack
```

Subroutines

```
Main:       ...
            acall sublabel
            ...
            ...
sublabel:   ...
            ...
            ret
```

call to the subroutine

the subroutine

Initializing Stack Pointer

- SP is initialized to 07 after reset.(Same address as R7)
- With each push operation 1st , pc is increased
- When using subroutines, the stack will be used to store the PC, so it is very important to initialize the stack pointer. Location 2Fh is often used.

```
mov SP, #2Fh
```

Subroutine - Example

```
square:  push b
         mov  b,a
         mul  ab
         pop  b
         ret
```

- 8 byte and 11 machine cycle

```
square: inc a
         movc a,@a+pc
         ret
table:  db 0,1,4,9,16,25,36,49,64,81
```

- 13 byte and 5 machine cycle

Subroutine - another example

```
; Program to compute square root of value on Port 3
; (bits 3-0) and output on Port 1.
        org 0
        ljmp Main

Main:   mov P3, #0xFF    ; Port 3 is an input
loop:   mov a, P3
        anl a, #0x0F     ; Clear bits 7..4 of A
        lcall sqrt
        mov P1, a
        sjmp loop

sqrt:   inc a
        movc a, @a + PC
        ret

Sqrs:   db  0,1,1,1,2,2,2,2,2,3,3,3,3,3,3,3
        end
```

reset service

main program

subroutine

data

Why Subroutines?

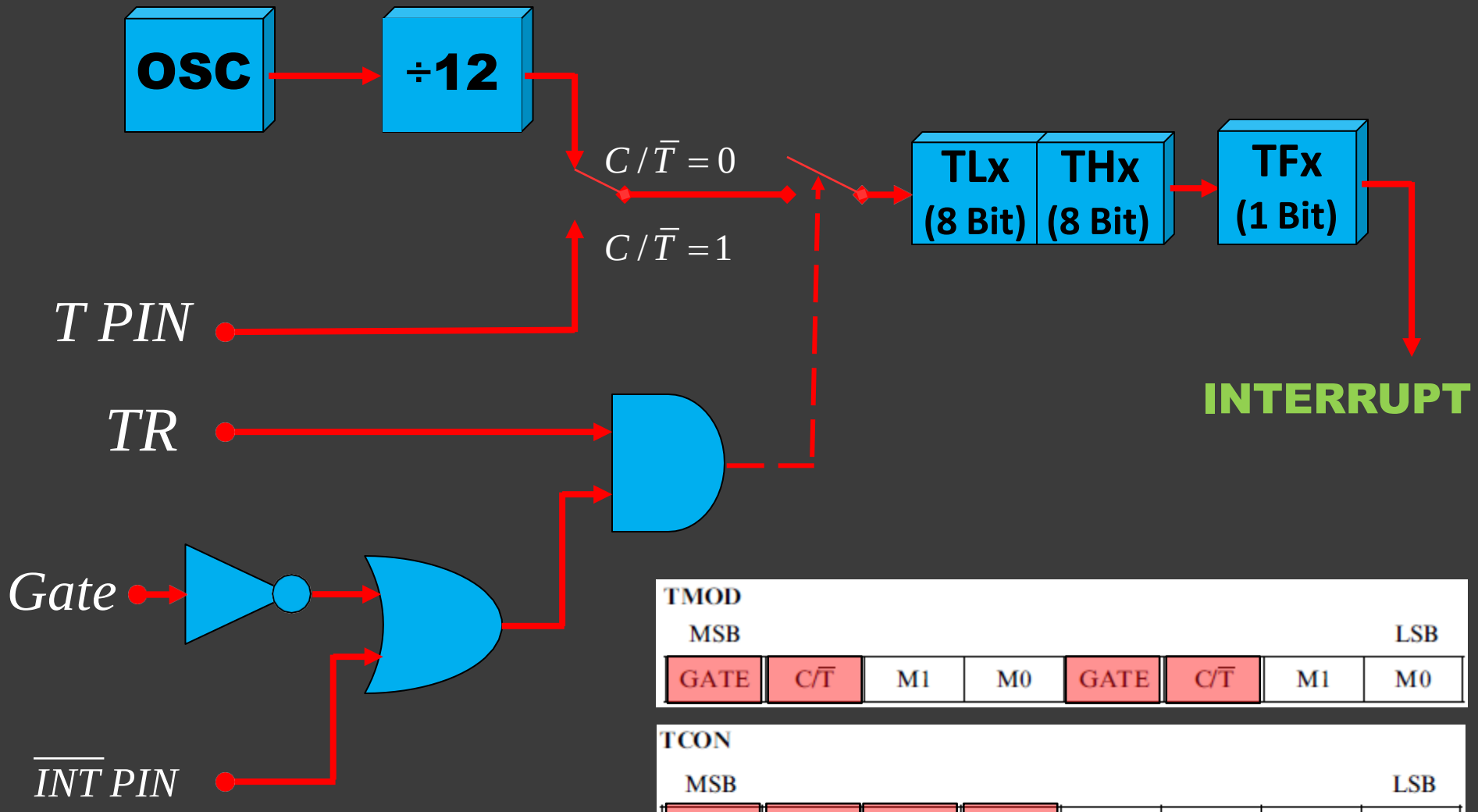
- Subroutines allow us to have "structured" assembly language programs.
- This is useful for breaking a large design into manageable parts.
- It saves code space when subroutines can be called many times in the same program.

Structure of Assembly Language

```
        ORG   0H          ;start (origin) at location 0
        MOV   R5,#25H     ;load 25H into R5
        MOV   R7,#34H     ;load 34H into R7
        MOV   A,#0        ;load 0 into A
        ADD   A,R5        ;add contents of R5 to A
                          ;now A = A + R5
        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               ;end of asm source file
```

2.4
Timer Operation

8051 Timer/Counter



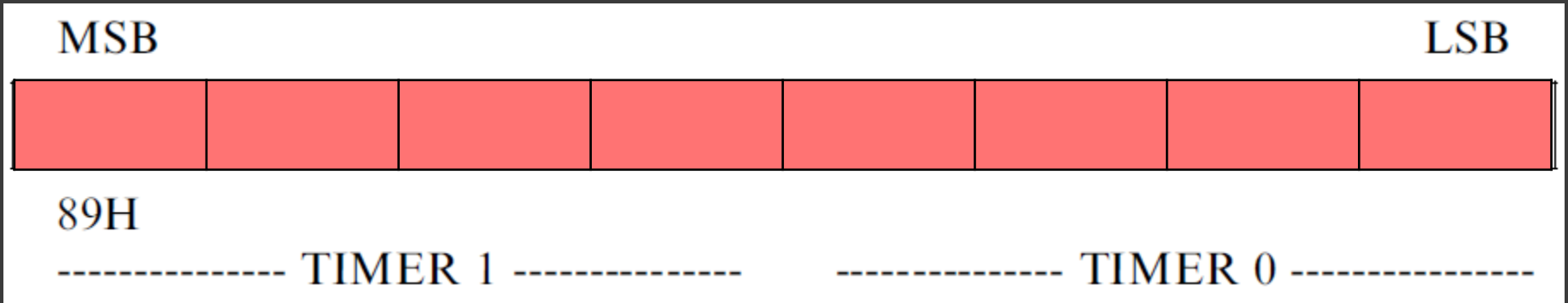
TMOD

MSB							LSB
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0

TCON

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TMOD Register



GATE:

When set, timer/counter x is enabled, **if** INTx pin is high and TRx is set.

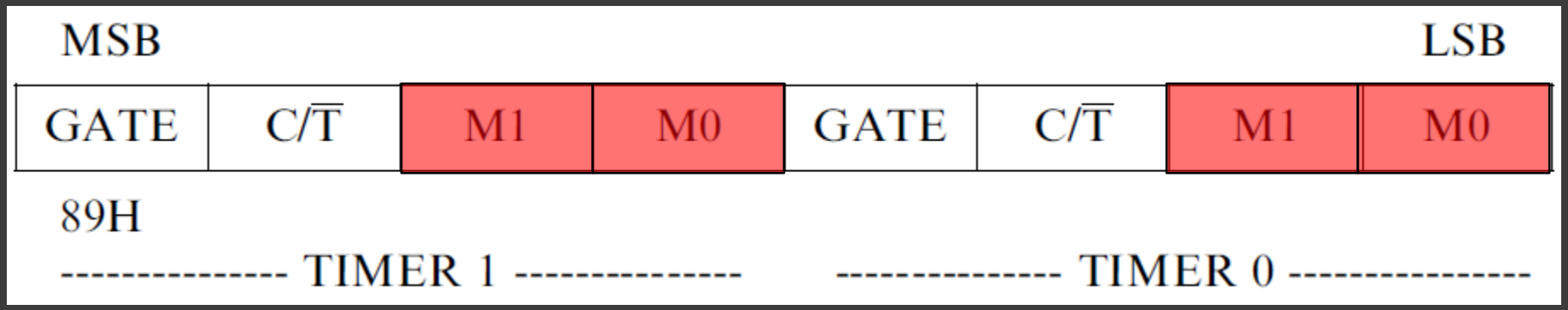
When cleared, timer/counter x is enabled, **if** TRx bit set.

C/T*:

When set, counter operation (input from Tx input pin).

When cleared, timer operation (input from internal clock).

TMOD Register



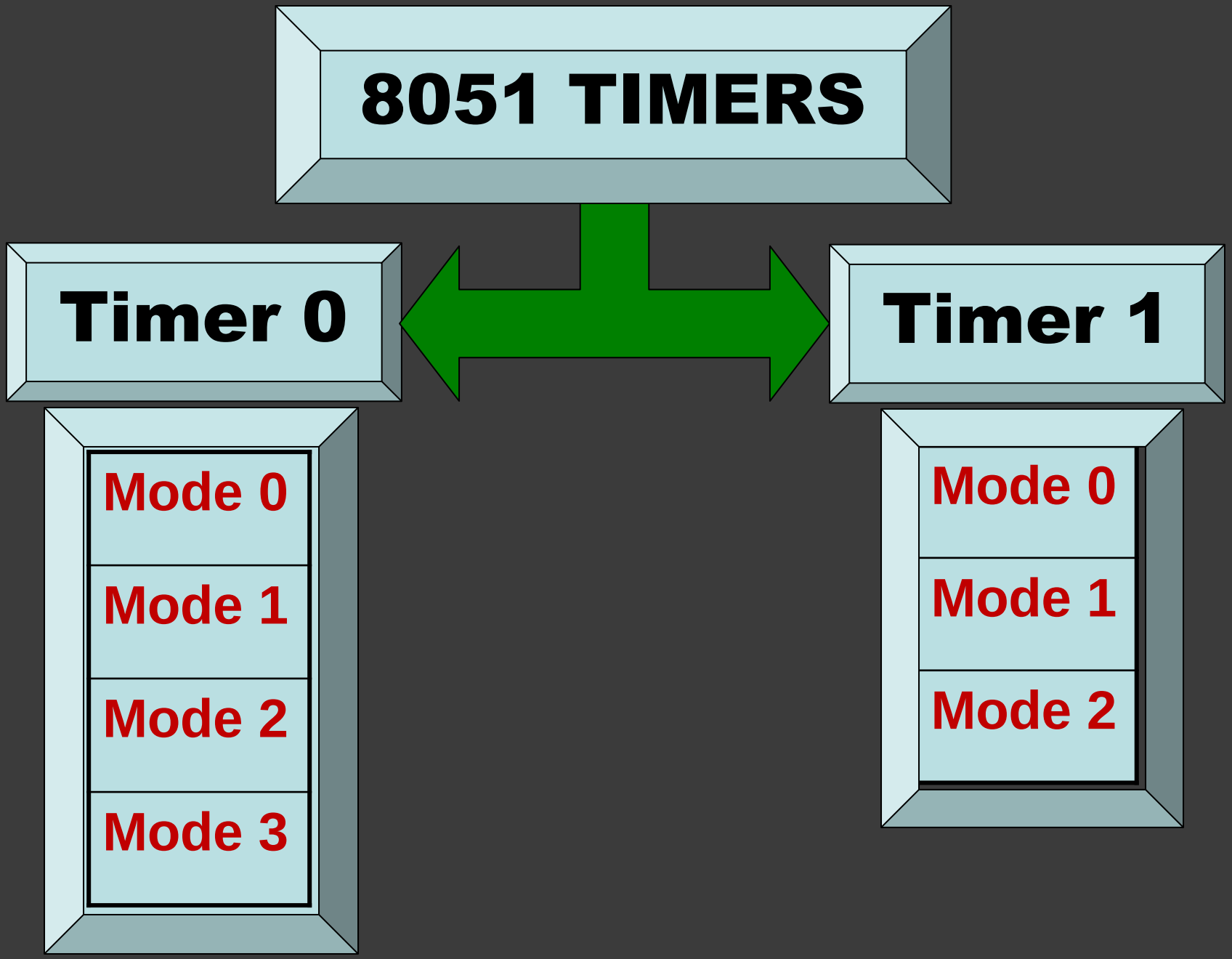
The TMOD byte is not bit addressable.

M1	M0	Operation
0	0	8048 8-bit timer TLx serves as 5-bit prescaler
0	1	16-bit timer/counter. THx and TLx are cascaded. No prescaler
1	0	8-bit autoreload timer/counter. THx contents loaded into TLx when it overflows
1	1	TL0 is 8-bit counter controlled by timer 0 control bits. TH0 is 8-bit timer controlled by timer 1 control bits
1	1	Timer 1 off

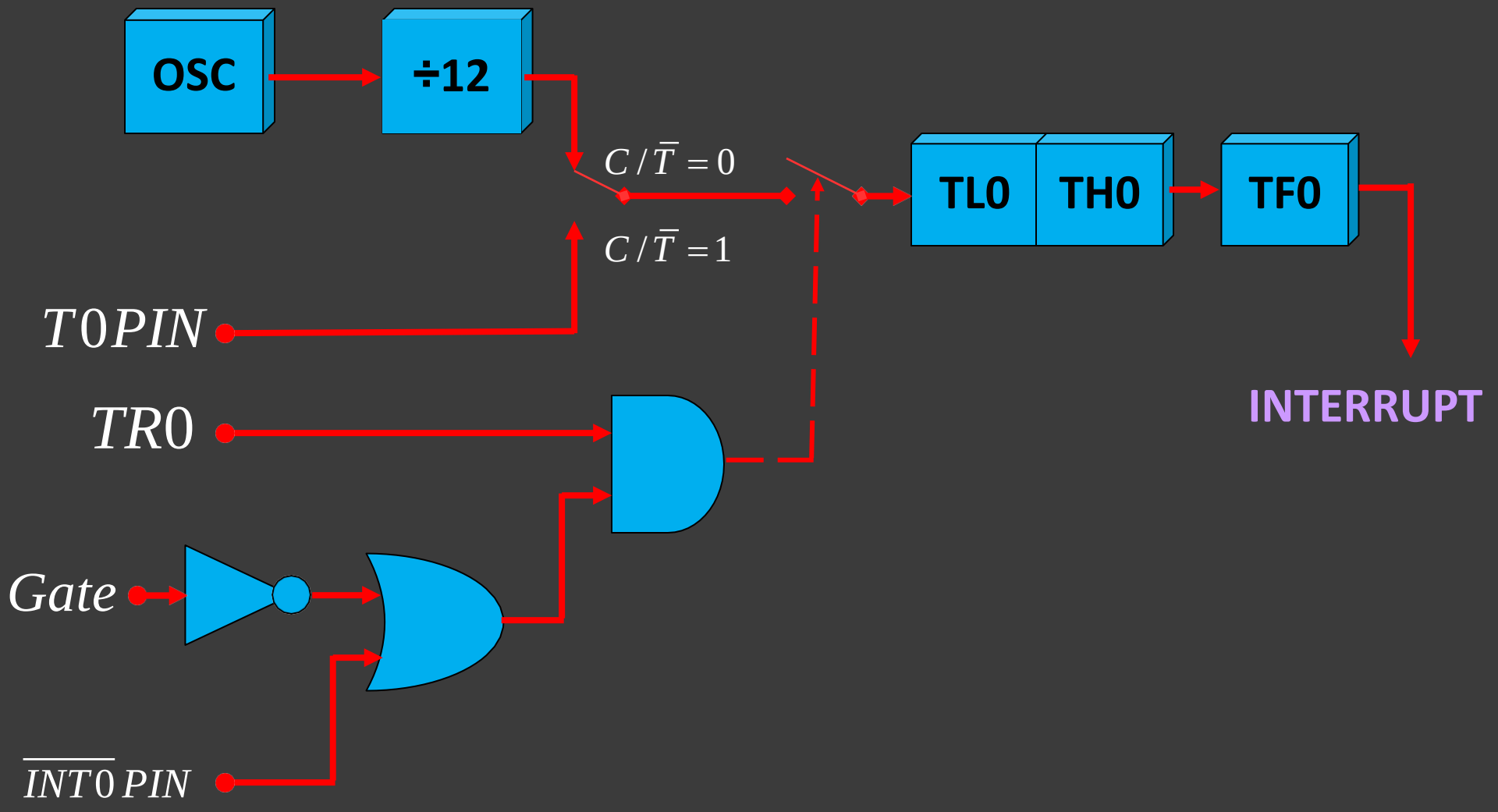
TCON Register

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
8FH	8EH	8DH	8CH	8BH	8AH	89H	88H

Bit	Function
TF1/0	Timer 1/0 overflow flag. Set by hardware on timer/counter overflow. Cleared when CPU vectors to interrupt routine
TR1/0	Timer 1/0 run control bit. Set/cleared by software to turn timer/counter on/off
IE1/0	Interrupt 1/0 edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
IT1/0	Interrupt 1/0 control bit. Set/cleared by software to specify falling edge/ low level triggered external interrupts

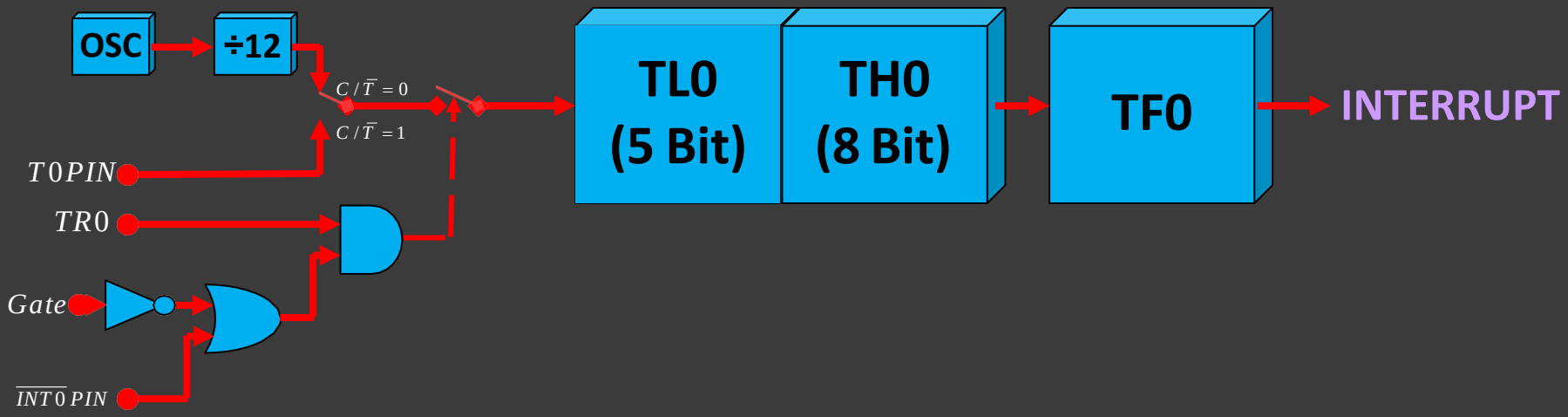

8051 TIMERS
Timer 0
Timer 1
Mode 0
Mode 1
Mode 2
Mode 3
Mode 0
Mode 1
Mode 2

TIMER 0


OSC
÷12
$C/\overline{T}=0$
$C/\overline{T}=1$
TL0
TH0
TF0
INTERRUPT
$T0PIN$
$TR0$
$Gate$
$\overline{INT0}\,PIN$

TIMER 0 – Mode 0

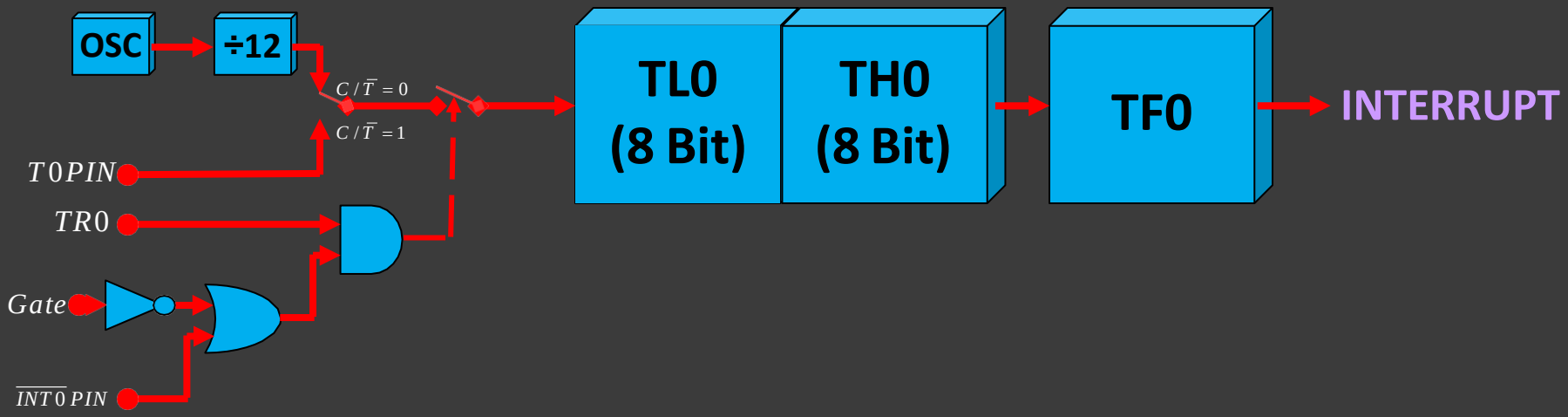
13 Bit Timer / Counter



Maximum Count = 1FFFh (0001111111111111)

TIMER 0 – Mode 1

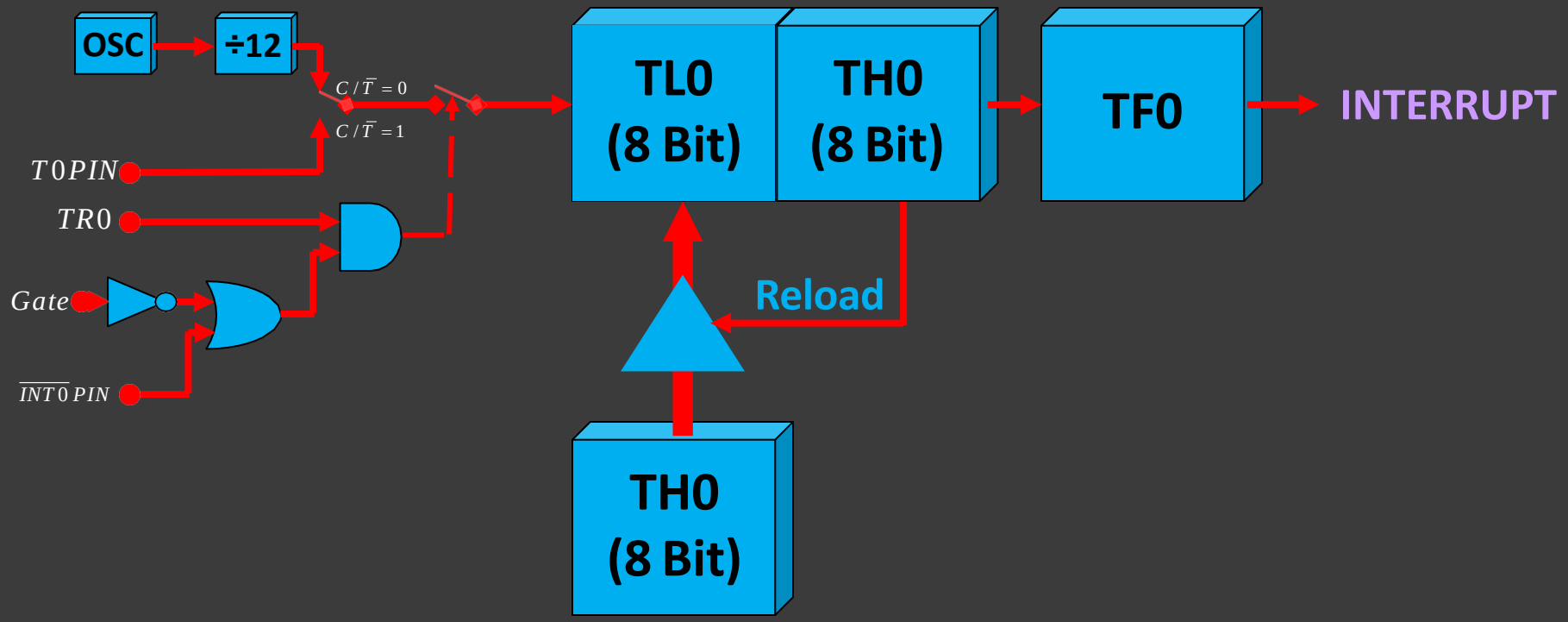
16 Bit Timer / Counter



Maximum Count = FFFFh (1111111111111111)

TIMER 0 – Mode 2

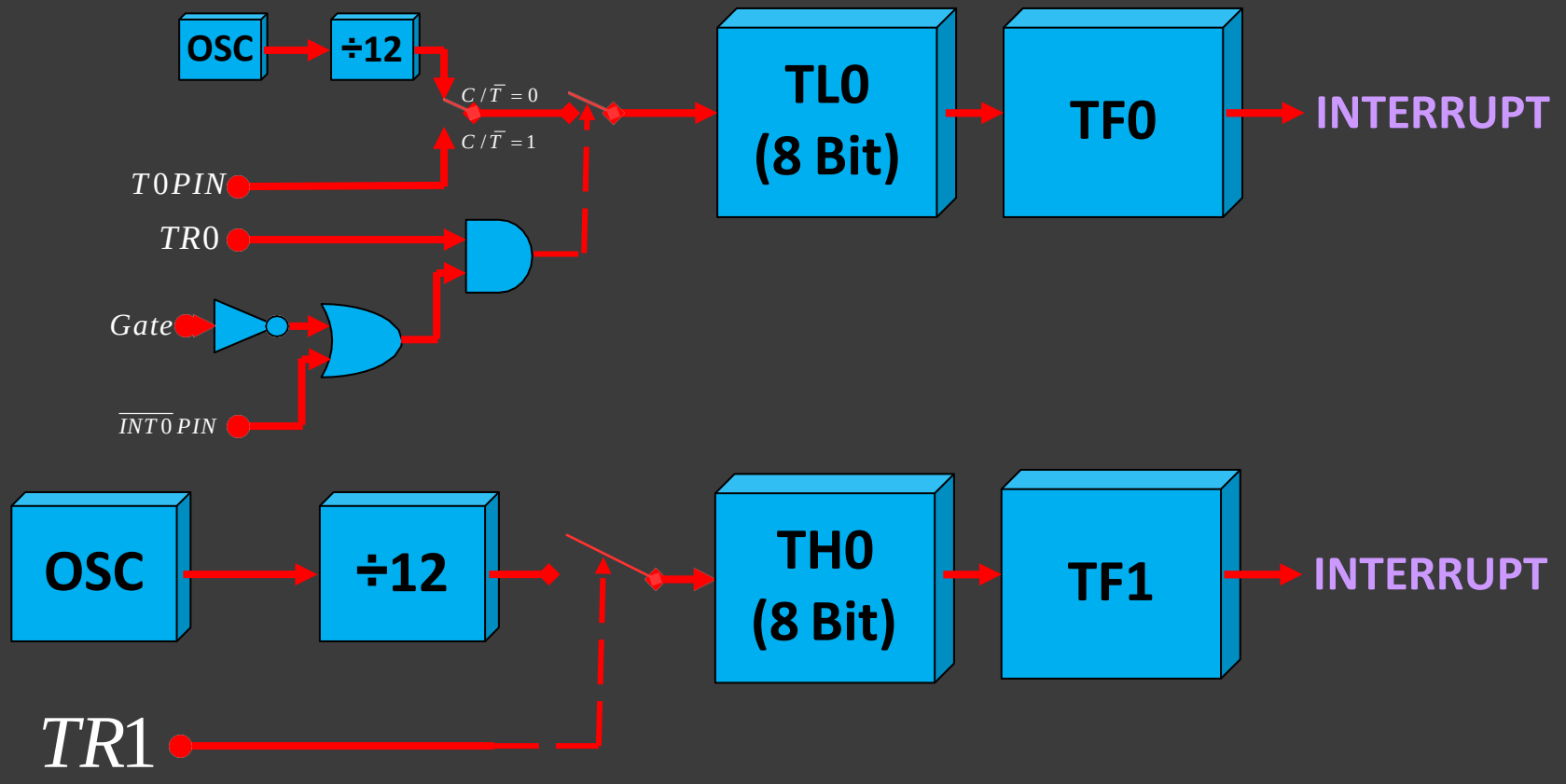
8 Bit Timer / Counter with AUTORELOAD



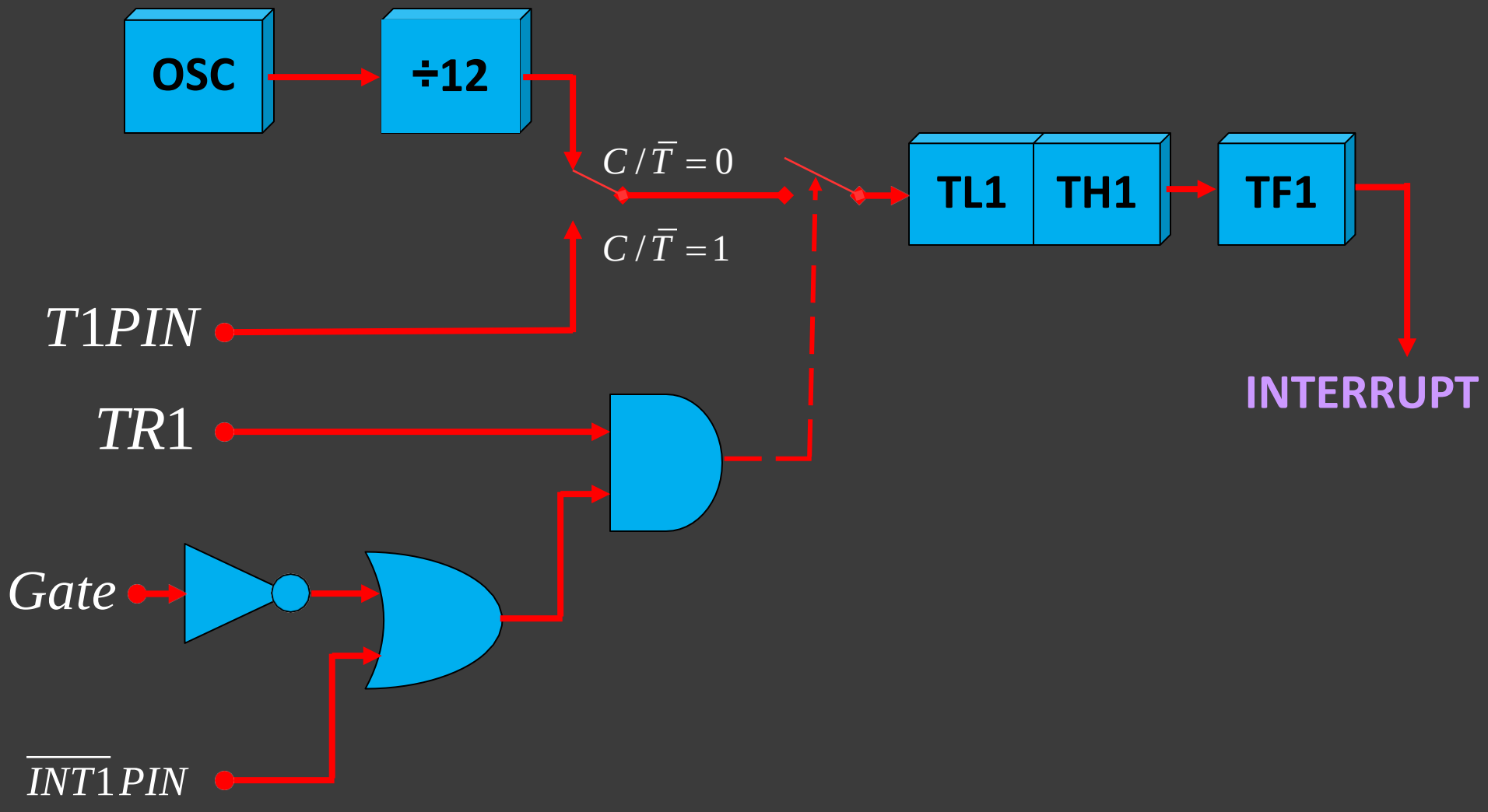
Maximum Count = FFh (11111111)

TIMER 0 – Mode 3

Two - 8 Bit Timer / Counter

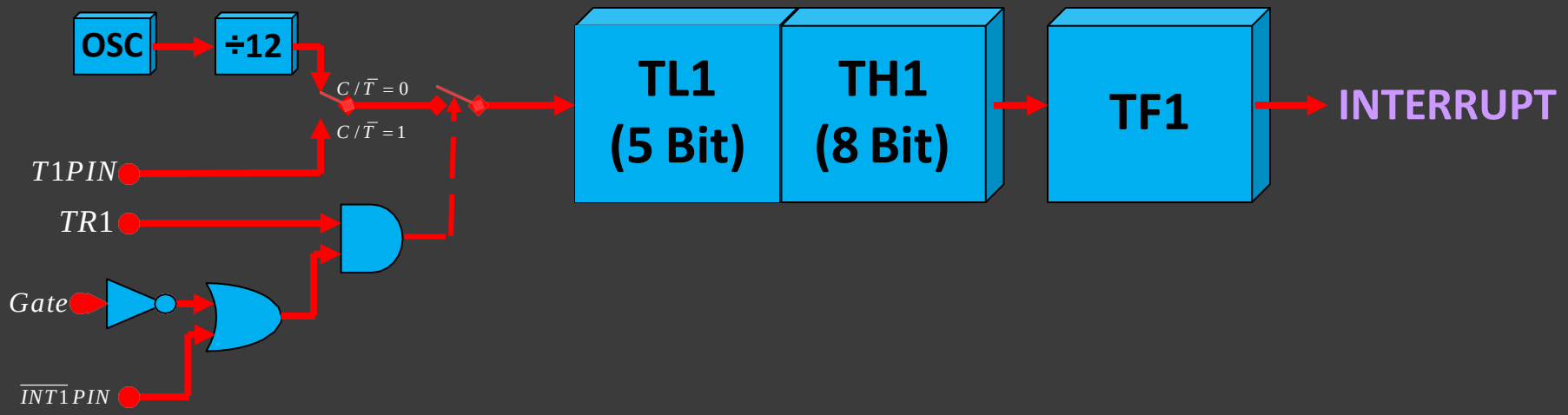


TIMER 1


OSC
÷12
$C/\overline{T}=0$
$C/\overline{T}=1$
TL1
TH1
TF1
INTERRUPT
$T1PIN$
$TR1$
$Gate$
$\overline{INT1}PIN$

TIMER 1 – Mode 0

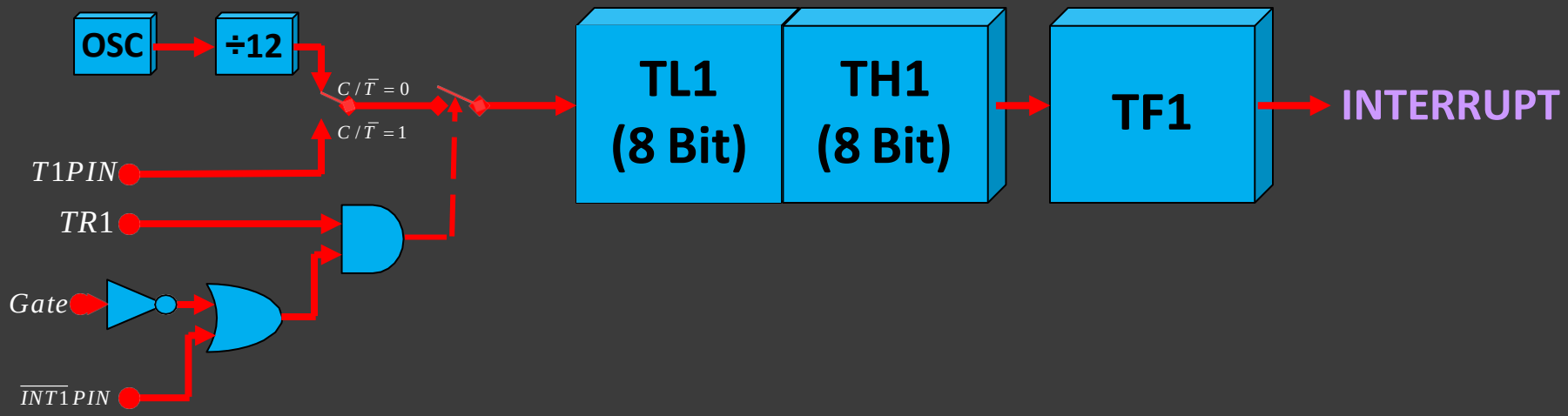
13 Bit Timer / Counter



Maximum Count = 1FFFh (0001111111111111)

TIMER 1 – Mode 1

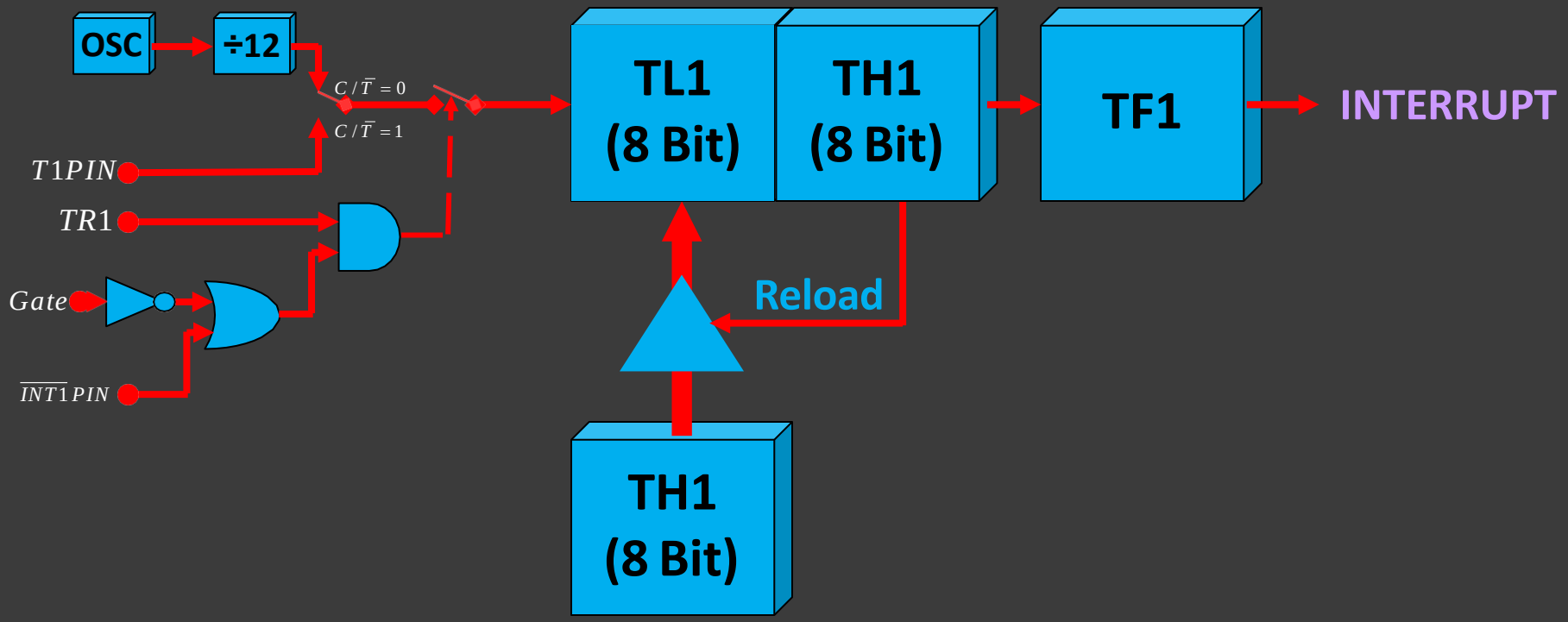
16 Bit Timer / Counter



Maximum Count = FFFFh (1111111111111111)

TIMER 1 – Mode 2

8 Bit Timer / Counter with AUTORELOAD



Maximum Count = FFh (11111111)

Programming Timers

- **Example:** 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.

  (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.

Programming Timers

- 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:**

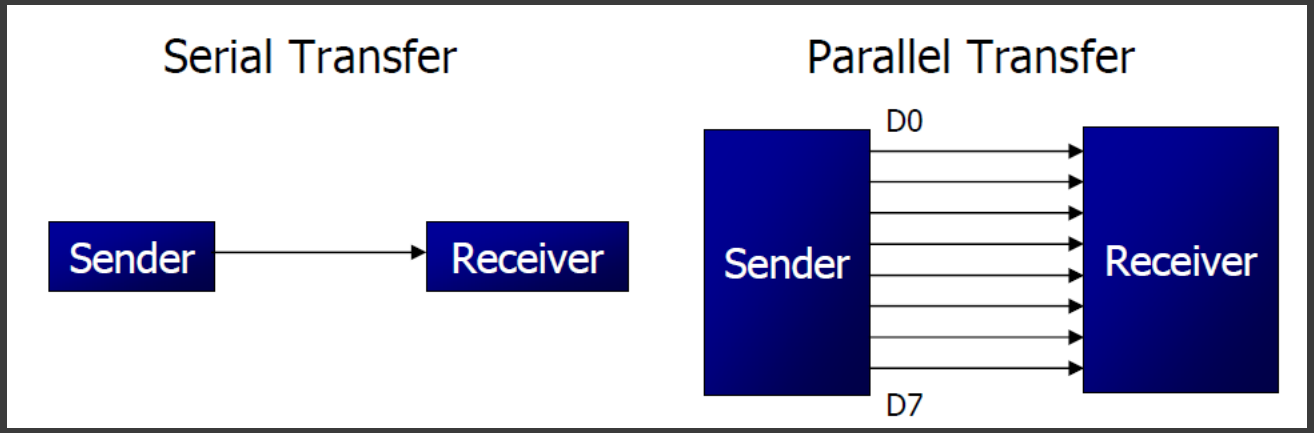
1/12 × 11.0529 MHz = 921.6 MHz;

T = 1/921.6 kHz = 1.085 us

2.5
Serial Port Operation

Basics of Serial Communication

- 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 Communication

- 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

- There are **special IC's** made by many manufacturers for serial communications.
  - **UART** (universal asynchronous Receiver transmitter)
  - **USART** (universal synchronous-asynchronous Receiver-transmitter)

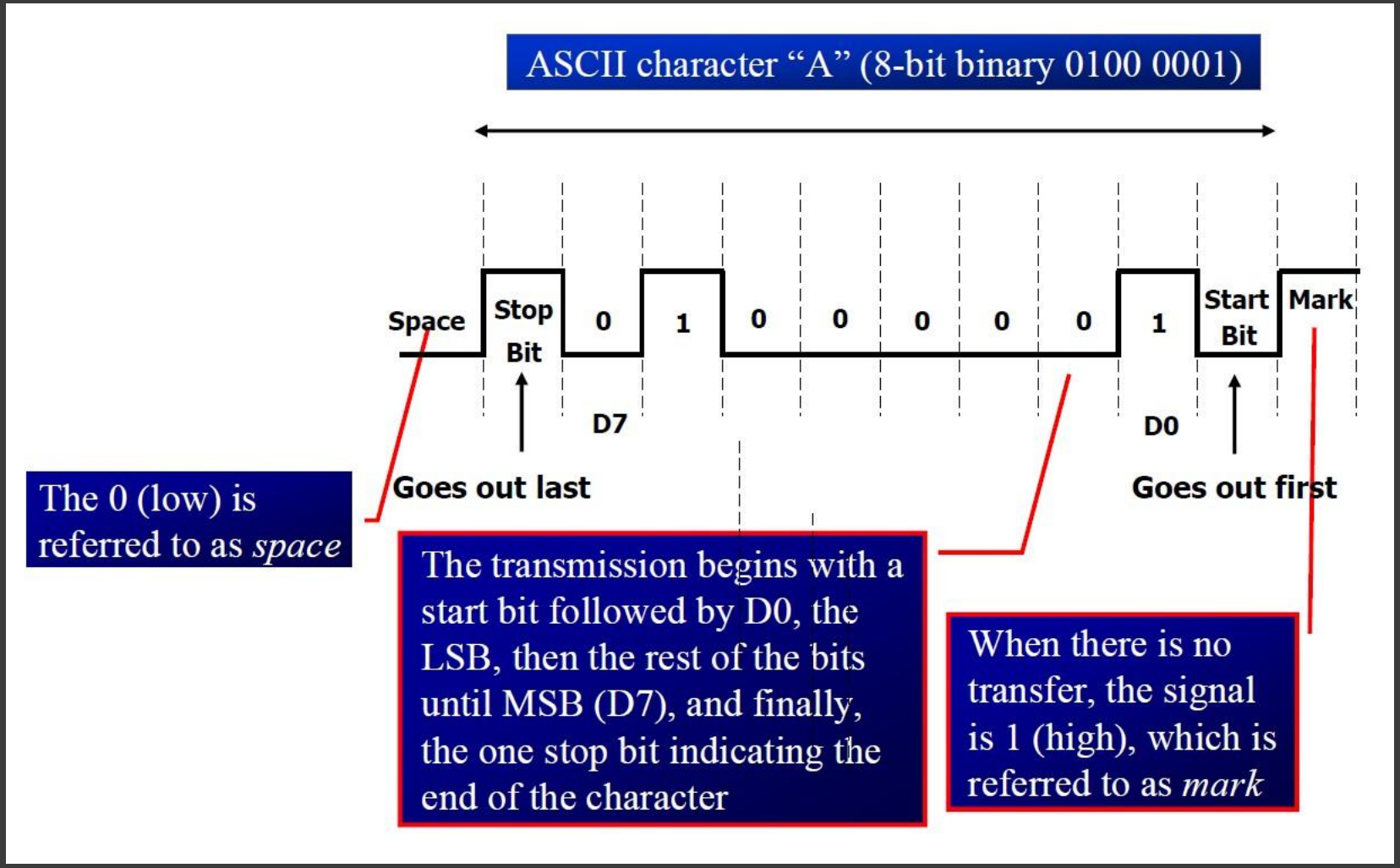
Asynchronous – Start & Stop Bit

- 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**

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

Asynchronous – Start & Stop Bit



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.**

8051 Serial Port

- Synchronous and Asynchronous
- SCON Register is used to Control
- Data Transfer through TXd & RXd pins
- Some time - Clock through TXd Pin
- Four Modes of Operation:

Mode 0	**:Synchronous Serial Communication**
Mode 1	**:8-Bit UART with Timer Data Rate**
Mode 2	**:9-Bit UART with Set Data Rate**
Mode 3	**:9-Bit UART with Timer Data Rate**

Registers related to Serial Communication

1. SBUF Register
2. SCON Register
3. PCON Register

SBUF Register

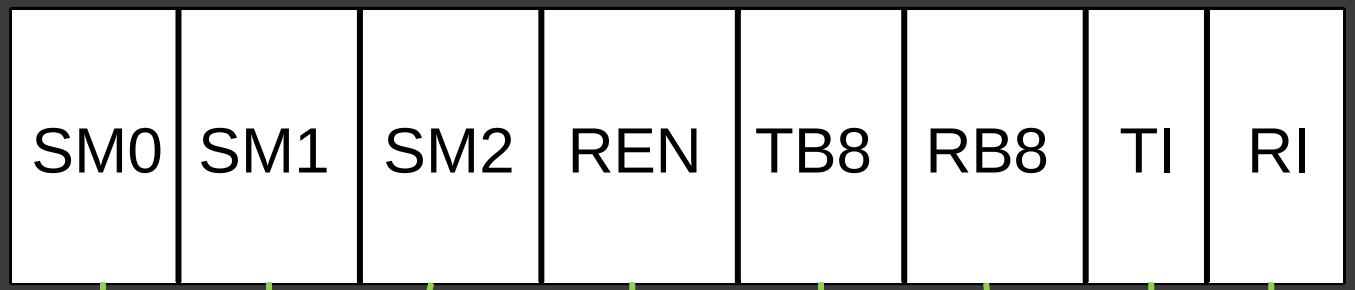
- **SBUF** is an **8-bit register** used solely for serial communication.
- For a byte data to be transferred via the **TxD line**, it must be placed in the **SBUF register**.
- The moment a byte is written into SBUF, it is framed with the start and stop bits and transferred serially via the TxD line.
- 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.

SBUF Register

- **Sample Program:**

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

SCON Register



Serial Mode	Explanation
0	8-bit Shift Register
1	8-bit UART
2	9-bit UART
3	9-bit UART

Set to Enable Serial Data reception

Set when a Cha-ractor received

Set when Stop bit Txed

Enable Multiprocessor Communication Mode

9th Data Bit Sent in Mode 2,3

9th Data Bit Received in Mode 2,3

8051 Serial Port – Mode 0

The Serial Port in Mode-0 has the following features:

1. Serial data **enters and exits through RXD**

2. TXD outputs the **clock**

3. 8 bits are transmitted / received

4. The baud rate is fixed at (1/12) of the oscillator frequency

8051 Serial Port – Mode 1

The Serial Port in Mode-1 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. On receive, the stop bit goes into RB8 in SCON
4. **10 bits** are transmitted / received
    1. *Start bit (0)*
    2. *Data bits (8)*
    3. *Stop Bit (1)*
5. Baud rate is determined by the Timer 1 over flow rate.

8051 Serial Port – Mode 2

The Serial Port in Mode-2 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. 9th data bit (**TB8**) can be assign value 0 or 1
4. On receive, the 9th data bit goes into **RB8** in SCON
5. **11 bits** are transmitted / received
    1. Start bit (0)
    2. Data bits (9)
    3. Stop Bit (1)
6. **Baud rate** is programmable

8051 Serial Port – Mode 3

The Serial Port in Mode-3 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. 9th data bit (**TB8**) can be assign value 0 or 1
4. On receive, the 9th data bit goes into **RB8** in SCON
5. **11 bits** are transmitted / received
    1. Start bit (0)
    2. Data bits (9)
    3. Stop Bit (1)
6. **Baud rate** is determined by Timer 1 overflow rate.

Programming Serial Data Transmission

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**

Programming Serial Data Reception

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.**

Doubling Baud Rate

- There are two ways to increase the baud rate of data transfer
  1. **By using a higher frequency crystal**
  2. **By changing a bit in the PCON register**

- **PCON register** is an 8-bit register.

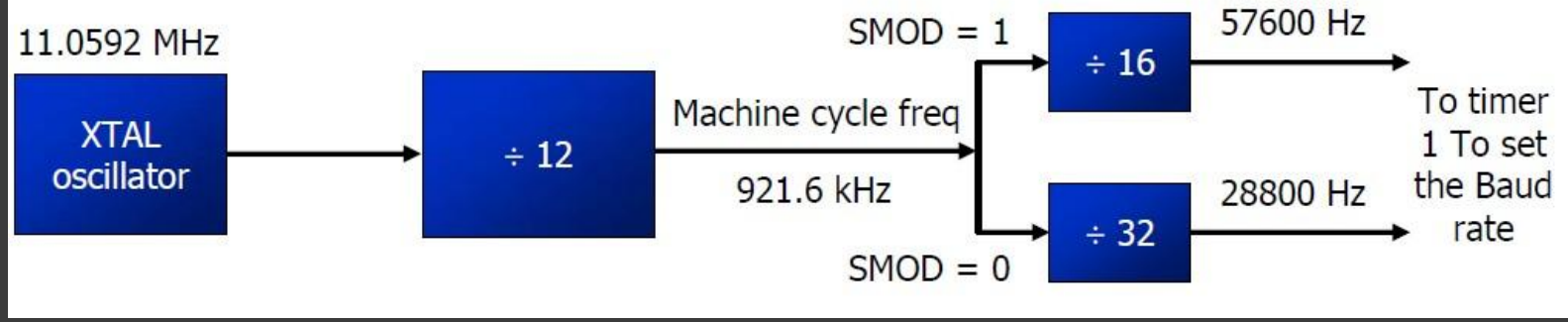
SMOD	--	--	--	GF1	GF0	PD	IDL

- When 8051 is powered up, **SMOD** is zero

- **We can set it to high** by software and thereby **double** the baud rate.

Doubling Baud Rate (cont…)

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



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

2.6
Interrupts

INTERRUPTS

- 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:

1. **Interrupt**
2. **Polling**

Interrupt Vs Polling

1. **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.
2. **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.

Interrupt Vs Polling

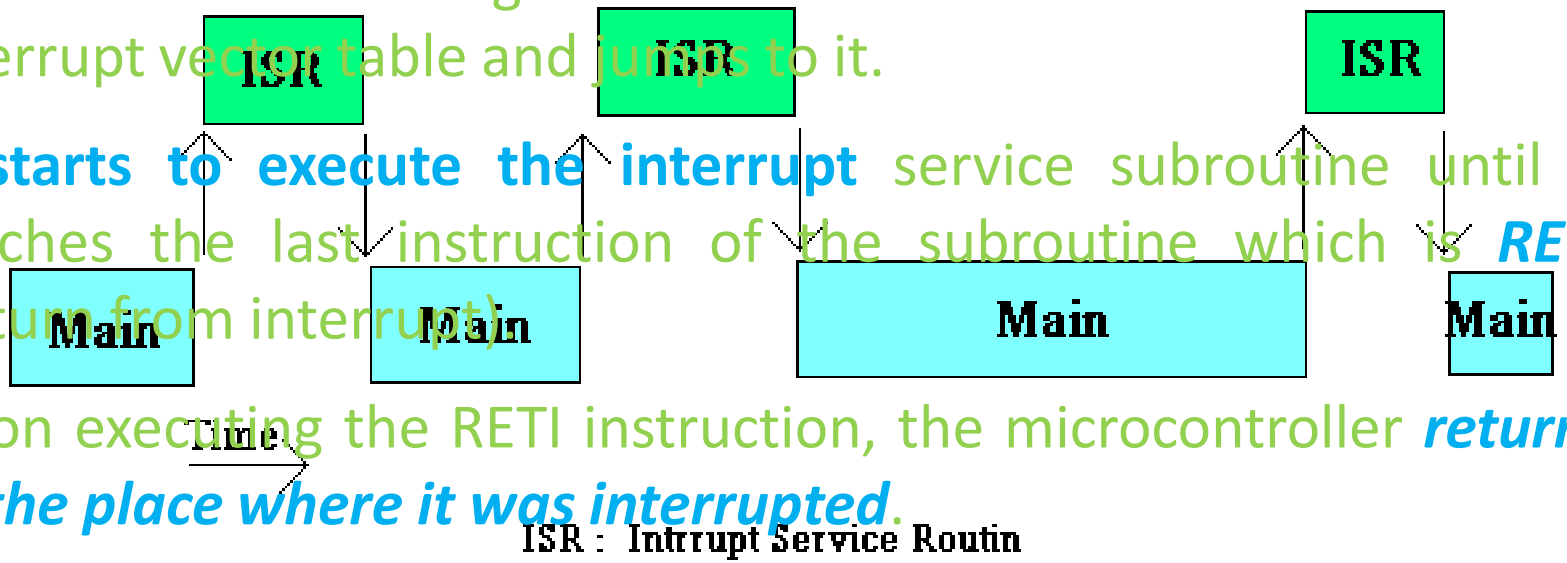
- The **polling method is not efficient**, since it wastes much of the microcontroller's time by polling devices that do not need service.
- The **advantage of interrupts** is that the microcontroller can serve many devices (not all at the same time).
- Each devices can get the attention of the microcontroller 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 in Interrupt.

Program execution without intrrupts :

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.

Program execution with intrrupts :

4. The microcontroller gets the address of the ***ISR*** from the interrupt vector table and jumps to it.
5. It **starts to execute the interrupt** service subroutine until it reaches the last instruction of the subroutine which is ***RETI*** (return from interrupt).
6. Upon executing the RETI instruction, the microcontroller ***returns to the place where it was interrupted***.



Six Interrupts in 8051

Six interrupts are allocated as follows:

1. **Reset – power-up reset.**

2. **Two interrupts are set aside for the timers.**
   - one for timer 0 and one for timer 1

3. **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)

4. **Serial communication has a single interrupt that belongs to both receive and transfer.**

What events can trigger Interrupts?

- We can configure the 8051 so that any of the following events will cause an interrupt:
  - Timer 0 Overflow.
  - Timer 1 Overflow.
  - Reception/Transmission of Serial Character.
  - External Event 0.
  - External Event 1.
- We can configure the 8051 so that when Timer 0 Overflows or when a character is sent/received, the appropriate interrupt handler routines are called.

8051 Interrupt Vectors

INTERRUPT VECTORS

When the original 8051 and 8031 were introduced, only 5 interrupts were provided.

Interrupt Number	Interrupt Vector Address	Description
0	0003h	EXTERNAL 0
1	000Bh	TIMER/COUNTER 0
2	0013h	EXTERNAL 1
3	001Bh	TIMER/COUNTER 1
4	0023h	SERIAL PORT

8051 Interrupt related Registers

- The various registers associated with the use of interrupts are:
  - TCON - Edge and Type bits for External Interrupts 0/1
  - SCON - RI and TI interrupt flags for RS232
  - IE - Enable interrupt sources
  - IP - Specify priority of 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.

Interrupt Enable (IE) Register

- EA : Global enable/disable.
- --- : Reserved for additional interrupt hardware.
- ES : Enable Serial port interrupt.
- ET1 : Enable Timer 1 control bit.
- EX1 : Enable External 1 interrupt.
- ET0 : Enable Timer 0 control bit.
- EX0 : Enable External 0 interrupt.

MOV IE,#08h
or
SETB ET1

Enabling and Disabling an Interrupt

- **Example:** Show the instructions to **(a)** enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 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

Interrupt Priority

- When the 8051 is powered up, the priorities are assigned according to the following.

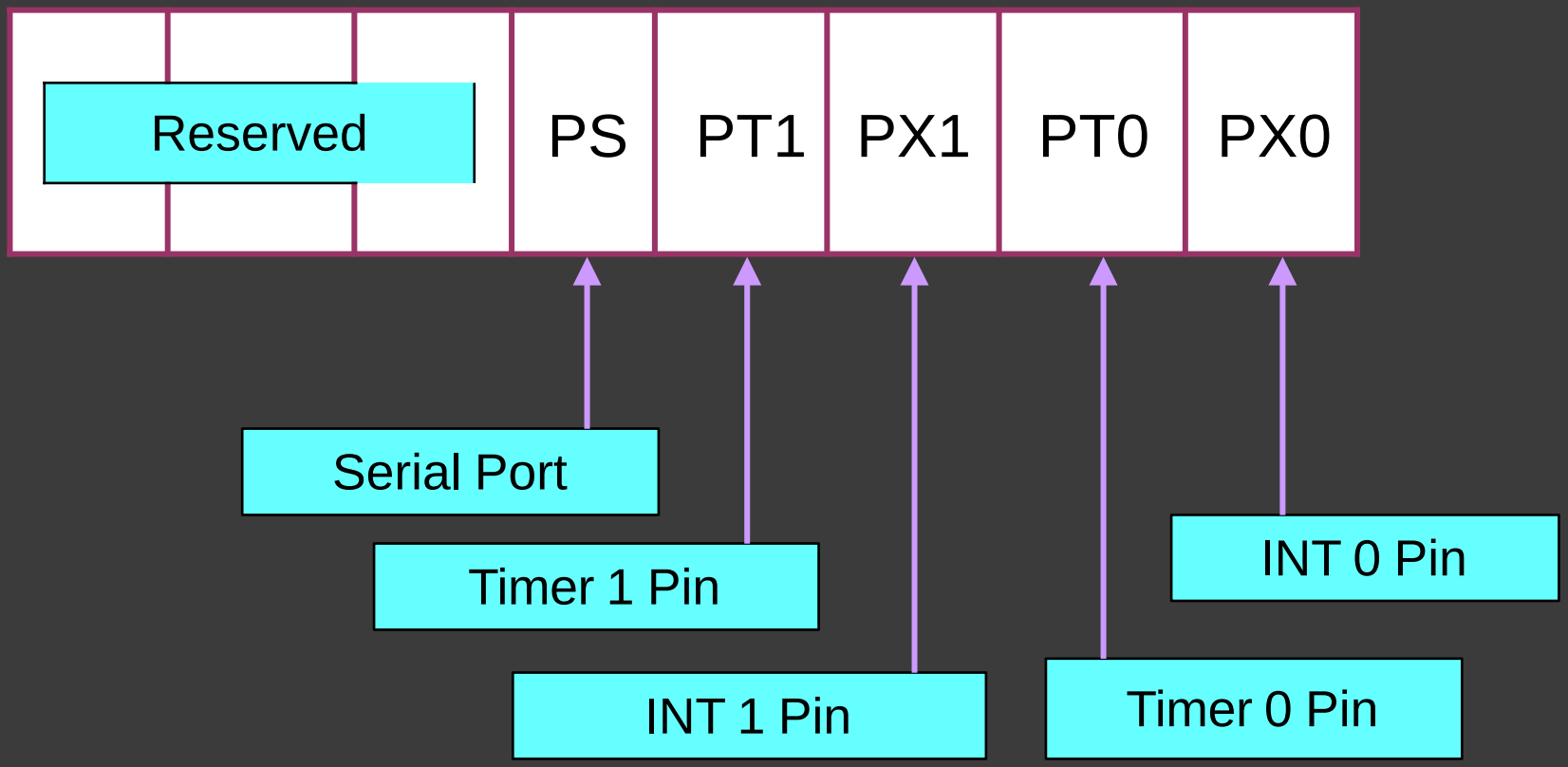
- In reality, the priority scheme is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed and responds accordingly.

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

- **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.**

Interrupt Priority (IP) Register



Priority bit=1 assigns high priority
Priority bit=0 assigns low priority

Chapter 3
Assembly/C Programming for Microcontroller

3.1 Assembler directives

ASSEMBLER DIRECTIVES

- Instructions to the assembler program.
- They are *not* assembly language instructions executable by the target microprocessor.
- However, they are placed in the mnemonic field of the program.
- With the exception of DB and DW, they have no direct effect on the contents of memory.
- Categories of directives:
- Assembler state control (ORG, END, USING)
- Symbol definition (SEGMENT, EQU, SET, DATA, IDATA, XDATA, BIT, CODE)
- Storage initialization/reservation (DS, DBIT, DB, DW)
- Program linkage (PUBLIC, EXTRN, NAME)
- Segment selection (RSEG, CSEG, DSEG, ISEG, BSEG, XSEG)

Assembler State Control

ORG (Set Origin)

End

Using

ORG (Set Origin)

- Format:

  ORG expression

- Alters the location counter to set a new program origin for statements that follow.
- A label is not permitted.

```
ORG 100H                          ;SET LOCATION
                                  ;COUNTER TO 100H
ORG ($ + 1000H) AND 0F000H        ;SET TO NEXT 4K
                                  ;BOUNDARY
```

- Can be used in any segment type.
- If the current segment is absolute, the value will be an absolute address in the current segment.
- If a relocatable segment is active, the value of the ORG expression is treated as an offset from the base address of the current instance of the segment.

End

- Format:

END

- Should be the last statement in the source file. No label is permitted and nothing beyond the END statement is processed by the assembler.

Using

- Format:

  USING expression
- Informs ASM51 of the currently active register bank. Subsequent uses of the predefined symbolic register addresses AR0 to AR7 will convert to the appropriate direct address.

  USING 3

  PUSH AR7

  USING 1

  PUSH AR7
- The first push assembles to PUSH 1FH (R7 in bank 3)
- The second push assembles to PUSH 0FH (R7 in bank 1).
- USING does not actually switch register banks.
- Executing 8051 instructions is the only way to switch register banks.

Symbol Definition

- The symbol definition directives create symbols that represent segments, registers, numbers, and addresses.
- None of these directives may be preceded by a label.
- Symbols defined by these directives may not have been previously defined and may not be redefined by any means.
- The SET directive is the only exception.

Segment

- Format:

symbol SEGMENT segment_type

- The symbol is the name of a relocatable segment.
- Defined 8051 segment types (memory spaces):
  - CODE (the code segment)
  - XDATA (the external data space)
  - DATA (the internal data space accessible by direct addressing, 00H-7FH)
  - IDATA (the entire internal data space accessible by indirect addressing, 00H-7FH, 00H-FFH on the 8052)
  - BIT (the bit space; overlapping byte locations 20H-2FH of the internal data space)

EQU (Equate)

- Format:

symbol EQU expression

- Assigns a numeric value to a specified symbol name.

```
N27       EQU 27              ;SET N27 TO THE VALUE 27
HERE      EQU $               ;SET "HERE" TO THE VALUE
                              ; OF THE LOCATION COUNTER
CR        EQU 0DH             ;SET CR (CARRIAGE RETURN) TO ODH
MESSAGE:          DB 'This is a message'
LENGTH EQU $ - MESSAGE              ;"LENGTH" EQUALS
                                    ;LENGTH OF "MESSAGE"
```

Other Symbol Definition Directives

- The SET directive is similar to the EQU directive except the symbol may be redefined later, using another SET directive.
- The DATA, IDATA, XDATA, BIT, and CODE directives assign addresses of the corresponding segment type to a symbol.
- A similar effect can be achieved using the EQU directive; if used, however, they evoke powerful typechecking by ASM51. Consider the following two directives and four instructions:

```
FLAG1       EQU 05H
FLAG2       BIT 05H
SETB FLAG1
SETB FLAG2
MOV FLAG1, #0
MOV FLAG2, #0
```

- The use of FLAG2 in the last instruction in this sequence will generate a "data segment address expected" error message from ASM51. Since FLAG2 is defined as a bit address (using the BIT directive), it can be used in a set bit instruction, but it cannot be used in a move byte instruction.

Storage Initialization/Reservation

- Initialize and reserve space in either word, byte, or bit units.
- The space reserved starts at the location indicated by the current value of the location counter in the currently active segment.
- These directives may be preceded by a label.

DS (Define Storage)

- Format:

[label:] DS expression

- Reserves space in byte units. It can be used in any segment type except BIT.
- The expression must be a valid assemble-time expression with no forward references and no relocatable or external references.
- The location counter of the current segment is incremented by the value of the expression.

```
                DSEG AT 30H       ;PUT IN DATA SEGMENT
                                  ;(ABSOLUTE, INTERNAL)
LENGTH EQU 40
BUFFER: DS LENGTH         ;40 BYTES RESERVED
```

- BUFFER represents the address of the first location of reserved memory.
- This buffer could be cleared using

```
        MOV     R7, #LENGTH
        MOV     R0, #BUFFER
LOOP:   MOV     @R4, #0
        DJNZ    R7, LOOP
        (continue)
```

- To create a 1000-byte buffer in external RAM starting at 4000H.

```
XSTART          EQU 4000H
XLENGTH         EQU 1000
                XSEG AT XSTART
XBUFFER:        DS XLENGTH
```

- This buffer could be cleared with:

```
        MOV     DPTR,#XBUFFER
LOOP:   CLR     A
        MOVX    @DPTR, A
        INC     DPTR
        MOV     A, DPL
        CJNE    A, #LOW(XBUFFER + XLENGTH + 1), LOOP
        MOV     A, DPH
        CJNE    A, #HIGH(XBUFFER + XLENGTH + 1), LOOP
        (continue)
```

- Since an instruction does not exist to compare the data pointer with an immediate value, the operation must be fabricated from available instructions.

DBIT

- Format:

[label:] DBIT expression

- Reserves space in bit units.
- Used only in a BIT segment.
- The location counter of the current (BIT) segment is incremented by the value of the expression.

```
        BSEG            ;BIT SEGMENT (ABSOLUTE)
KBFLAG: DBIT 1  ;KEYBOARD STATUS
PRFLAG: DBIT 1  ;PRINTER STATUS
DKFLAG: DBIT 1  ;DISK STATUS
```

- If the definitions above were the first use of BSEG, then **KBFLAG** would be at bit address 00H (bit 0 of byte address 20H).
- If other bits were defined previously using BSEG, then the definitions above would follow the last bit defined.

DB (Define Byte)

- Format:

[label:] DB expression [,expression][. . .]

- Initializes code memory with byte values.
- Since it is used to actually place data constants in code memory, a CODE segment must be active.
- Permits character strings longer than two characters as long as they are not part of an expression. Each character in the string is converted to the corresponding ASCII code.

```
                CSEG AT O100H
SQUARES:        DB 0, 1, 4,9, 16, 25    ;SQUARES OF NUMBERS 0-5
MESSAGE:        DB 'Login:',0           ;NULL-TERMINATED
                                        ;CHARACTER STRING
```

DW (Define Word)

- Format:

[label:] DW expression

- Same as the DB directive except two memory locations (16 bits) are assigned for each data item.

```
CSEG  AT 200H
DW    $, 'A', 1234H, 2, 'BC'
```

ASSEMBLER CONTROLS

- Establish the format of the listing and object files by regulating the actions of ASM51.
- For the most part, assembler controls affect the look of the listing file, without having any effect on the program itself.
- They can be entered on the invocation line when a program is assembled, or they can be placed in the source file.
- Assembler controls appearing in the source file must be preceded with a dollar sign and must begin in column 1.
- There are two categories of assembler controls: primary and general.
- Primary controls can be placed in the invocation line or at the beginning of the source program.
- Only other primary controls may precede a primary control.
- General controls may be placed anywhere in the source program.

3.2 Assembler operation

ASSEMBLER OPERATION

- ASM51 is a powerful assembler with all the bells and whistles. It is available on Intel development systems and on the IBM PC family of microcomputers. Since these "host" computers contain a CPU chip other than the 8051, ASM51 is called a **cross assembler**.
- Program may be written on the host computer and assembled to an object file and listing file.
- Execution on the host computer requires either hardware emulation or software simulation of the target CPU. A third possibility is to download the object program to an 8051-based target system for execution.

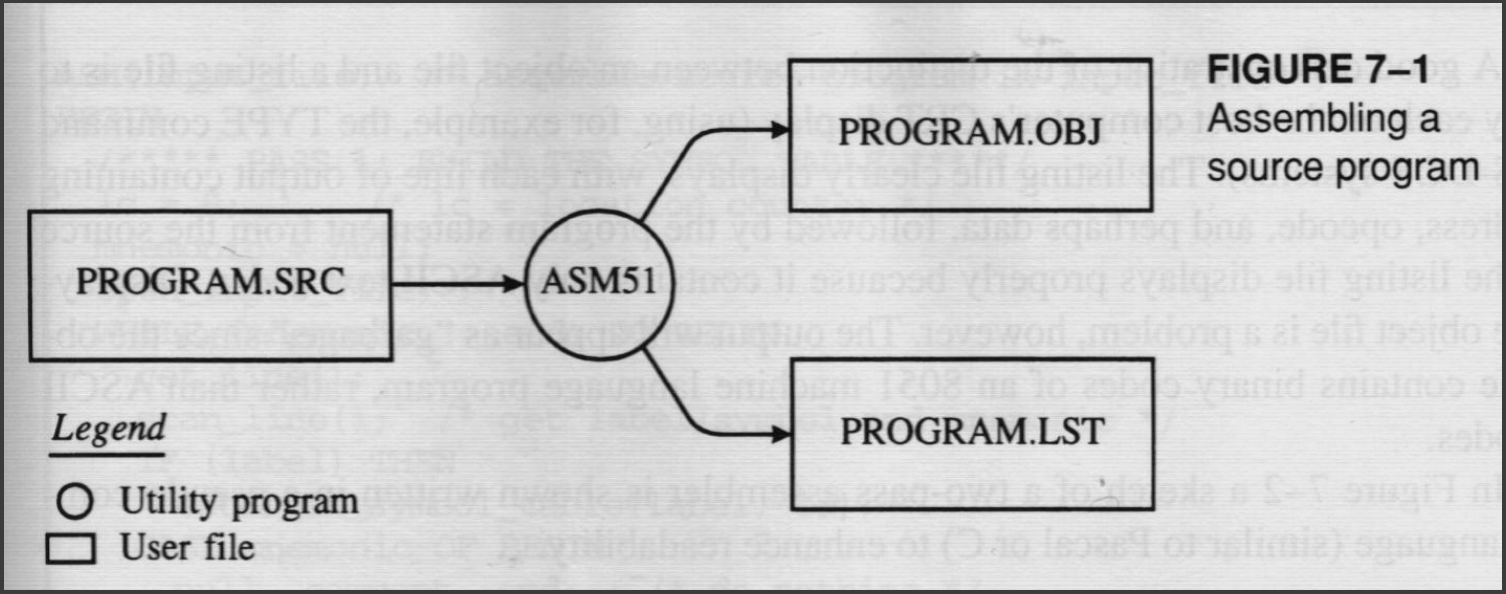


FIGURE 7–1
Assembling a source program

- ASM51 is invoked from the system prompt by
- ASM51 source file [assembler controls]
- The assembler receives a source file as input (e.g., PROGRAM.SRC) and generates an object file (PROGRAM.OBJ) and listing file (PROGRAM.LST) as output.
- Most assemblers scan the source program twice in performing the translation to machine language, they are described as two-pass assemblers. The assembler uses a location counter as the address of instructions and the values for labels.

Pass One

- Source file is scanned line-by-line and a symbol table is built.
- The location counter defaults to 0 or is set by the ORG (set origin) directive.
- As the file is scanned, the location counter is incremented by the length of each instruction.
- Define data directives (DBs or DWs) increment the location counter by the number of bytes defined.
- Reserve memory directives (DSs) increment the location counter by the number of bytes reserved.
- Each time a label is found at the beginning of a line, it is placed in the symbol table along with the current value of the location counter.
- Symbols that are defined using equate directives (EQUs) are placed in the symbol table along with the "equated" value.
- The symbol table is saved and then used during pass two.

Pass Two

- Object and listing files are created.
- Mnemonics are converted to opcodes and placed in the output files.
- Operands are evaluated and placed after the instruction opcodes.
- Symbols values are retrieved from the symbol table.
- Since two passes are performed, the source program may use "forward references," that is, use a symbol before it is defined (for example, in branching ahead in a program).
- The object file, if it is absolute, contains only the binary bytes (OOH-FFH) of the machine language program.
- A relocatable object file will also contain a symbol table and other information required for linking and locating.
- The listing file contains ASCII text codes (20H-7EH) for both the source program and the hexadecimal bytes in the machine language program.

ASSEMBLY LANGUAGE PROGRAM FORMAT

- Assembly language programs contain the following:
  - Machine instructions
  - Assembler directives
  - Assembler controls
  - Comments
- Machine instructions are the familiar mnemonics of executable instructions (e.g., ANL).
- Assembler directives are instructions to the assembler program that define program structure, symbols, data, constants, and so on (e.g., ORG).
- Assembler controls set assembler modes and direct assembly flow (e.g., $TITLE).
- Comments enhance the readability of programs by explaining the purpose and operation of instruction sequences.

- Lines containing machine instructions or assembler directives must be written following specific rules understood by the assembler.
- Each line is divided into "fields" separated by space or tab characters.
- The general format for each line is as follows:

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

3.3 Compiler operations

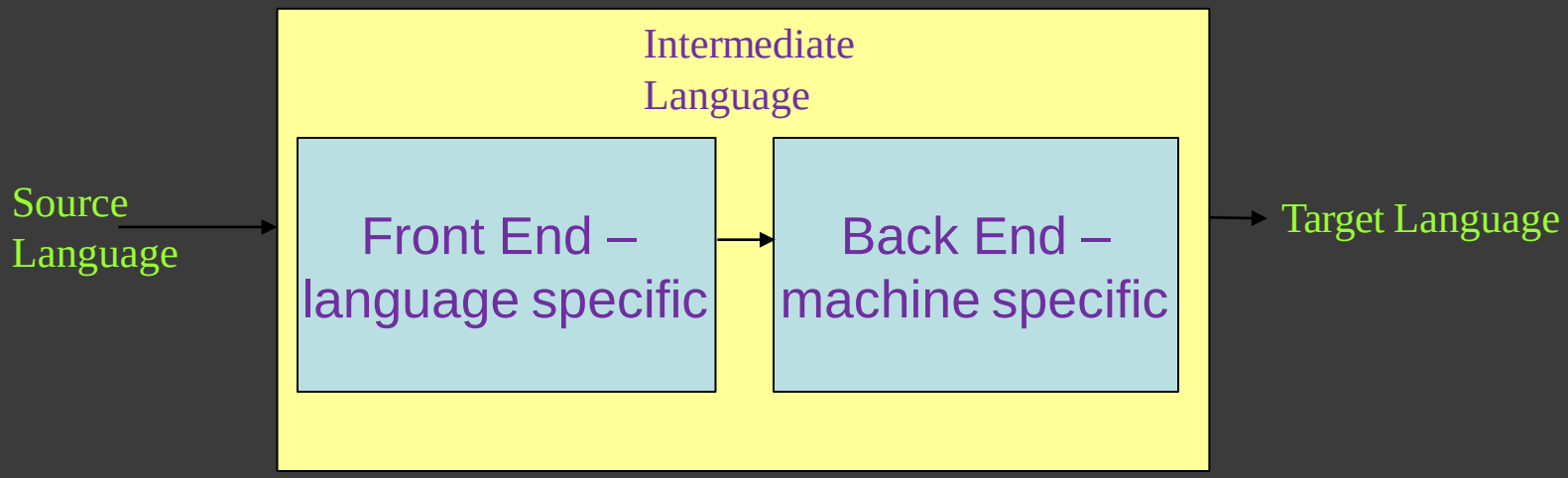
What is a compiler?

A program that reads a program written in one language and translates it into another language.

Traditionally, compilers go from high-level languages to low-level languages.

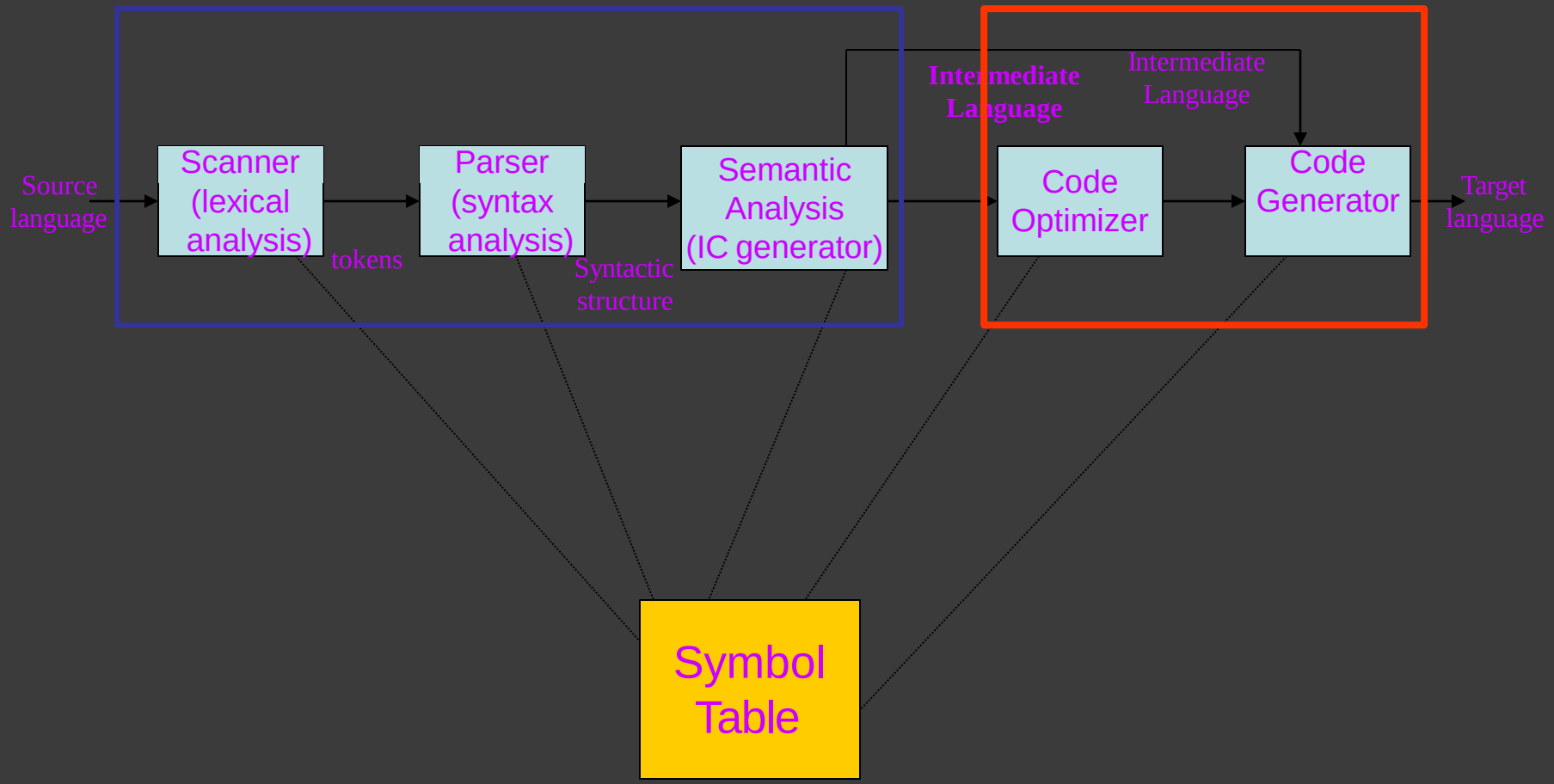
Compiler Architecture

In more detail:

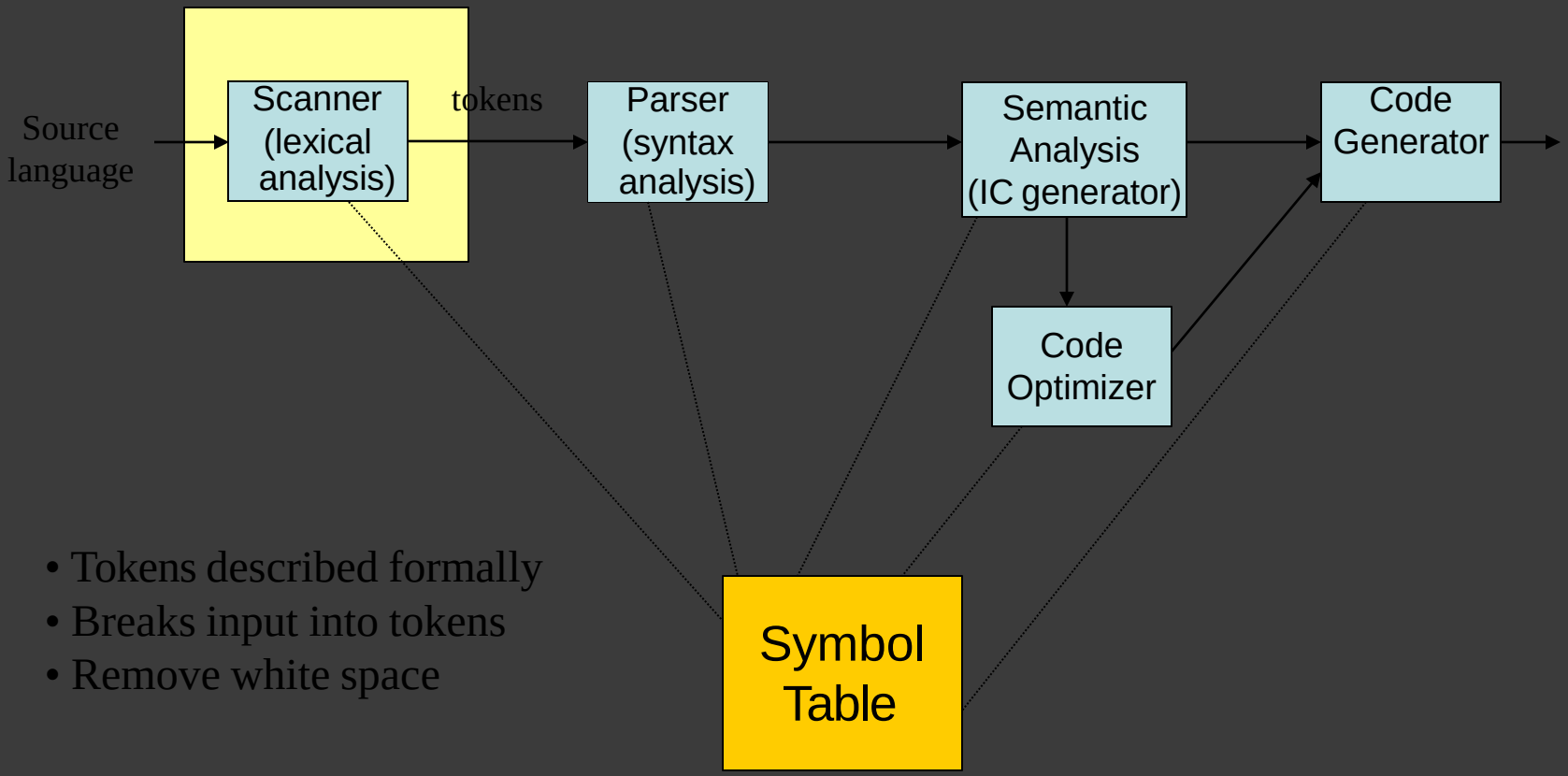


- Separation of Concerns
- Retargeting

Compiler Architecture

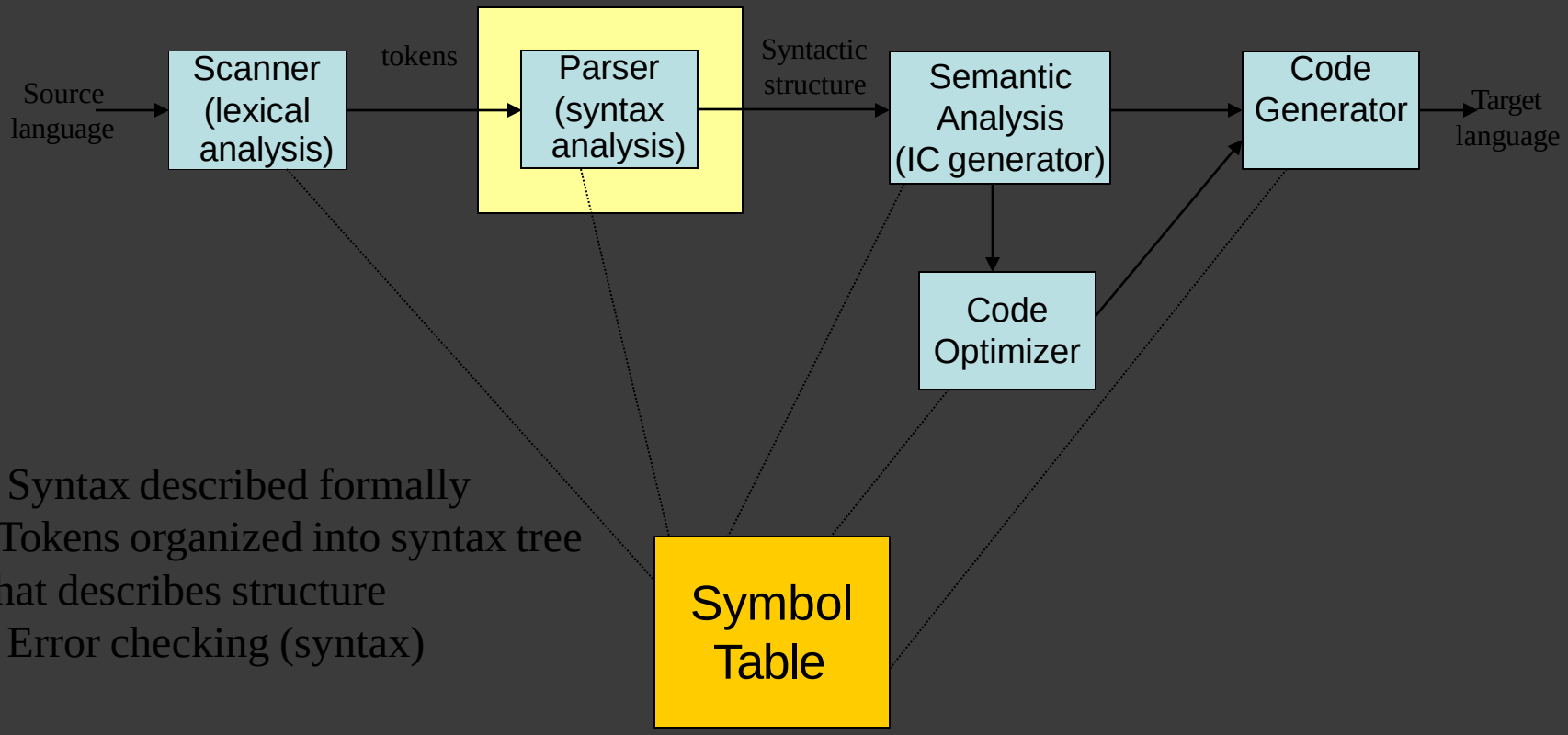


Lexical Analysis - Scanning



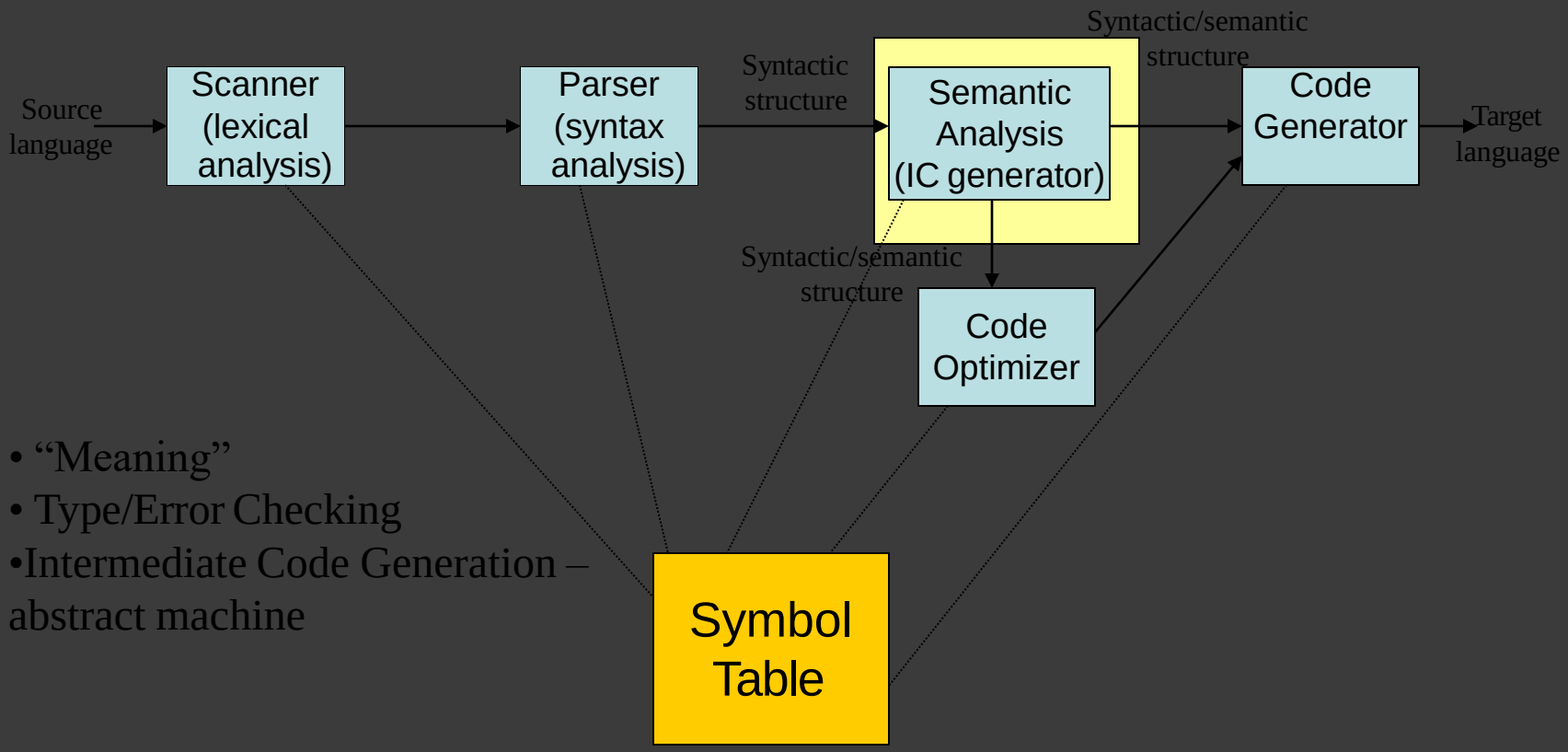
- Tokens described formally
- Breaks input into tokens
- Remove white space

Static Analysis - Parsing



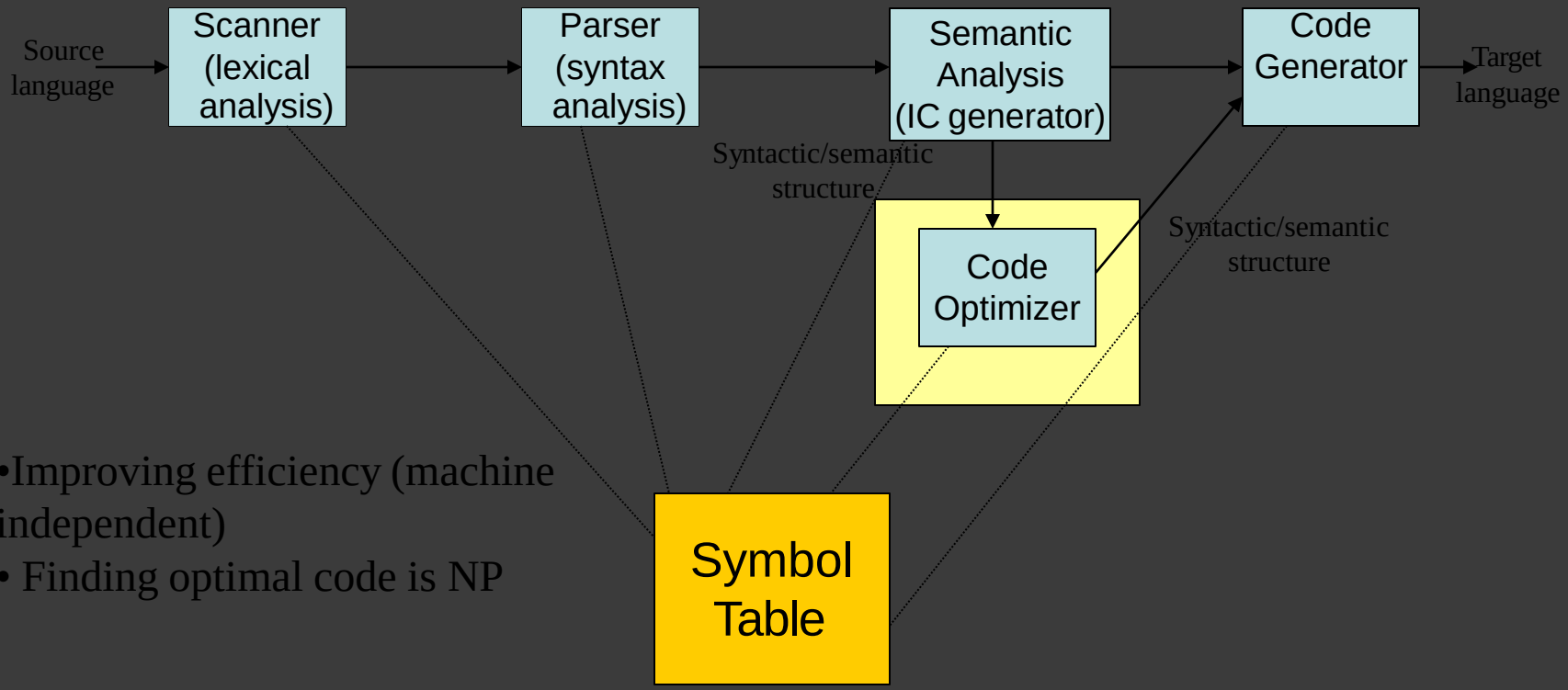
- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

Semantic Analysis



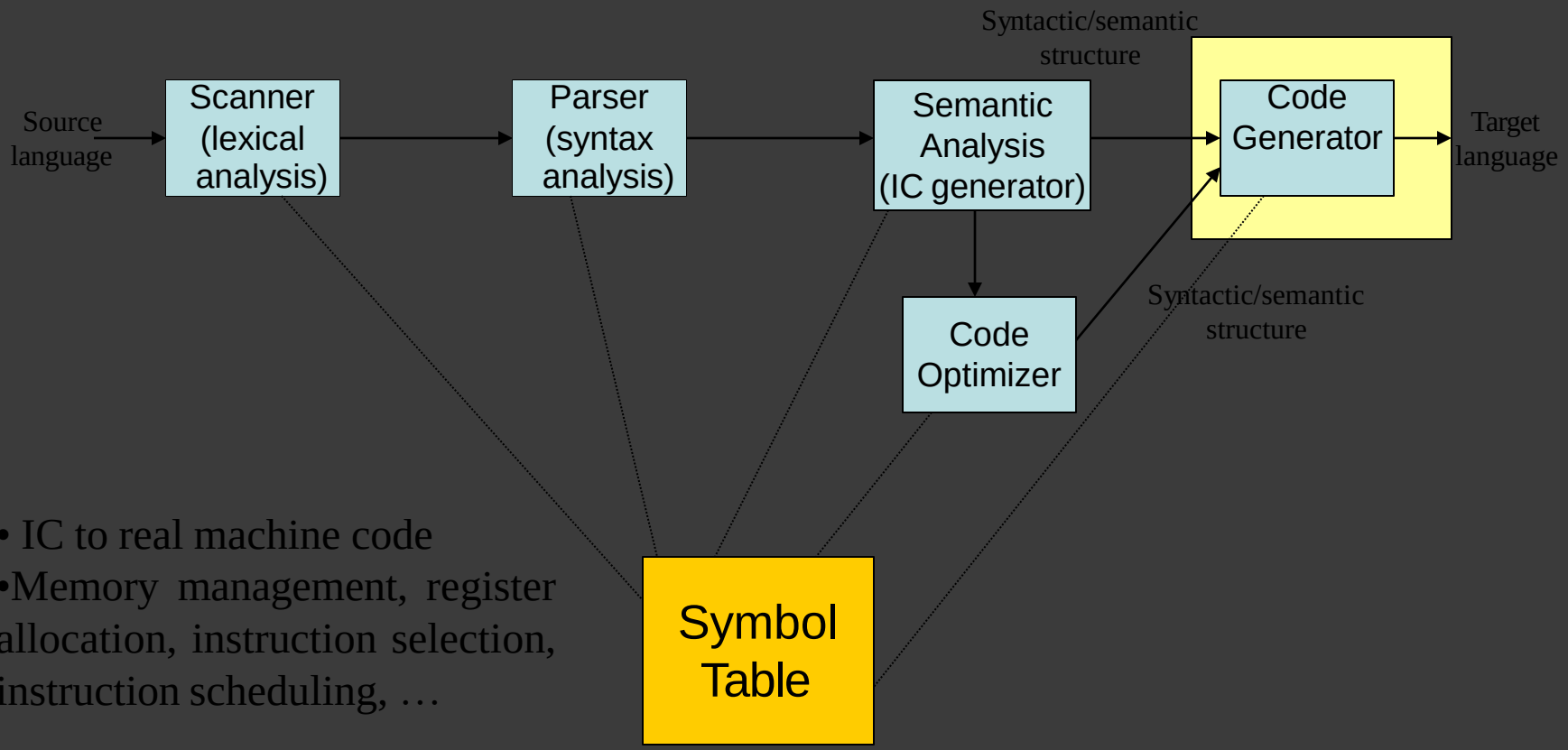
- "Meaning"
- Type/Error Checking
- Intermediate Code Generation – abstract machine

Optimization



- Improving efficiency (machine independent)
- Finding optimal code is NP

Code Generation



- IC to real machine code
- Memory management, register allocation, instruction selection, instruction scheduling, …

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
  - Degrees of optimization
  - Multiple passes
- Space
- Feedback to user
- Debugging

Related to Compilers

- Interpreters (direct execution)
- Assemblers
- Preprocessors
- Text formatters (non-WYSIWYG)
- Analysis tools

Why study compilers?

- Bring together:
  - Data structures & Algorithms
  - Formal Languages
  - Computer Architecture
- Influence:
  - Language Design
  - Architecture (influence is bi-directional)
- Techniques used influence other areas (program analysis, testing, …)

3.4 Debugger

Introduction

- Debugging is methodical process for removing mistakes in program
- So important, whole set of tools to help. Called "debuggers"
  - Trace code, print values, profile
  - New Integrated Development Environments (IDEs) (such as Game Maker) have it built in
- But debugging still frustrating
  - Beginners not know how to proceed
  - Even advanced can get "stuck"
- Don't know how long takes to find
  - Variance can be high
- What are some tips? What method can be applied?

5-Step Debugging Process

Step 1: Reproduce the Problem Consistently

- Find case where always occurs
  - "Sometimes game crashes after kill boss" doesn't help much
- Identify steps to get to bug
  - Ex: start single player, room 2, jump to top platform, attack left, …
  - Produce systematic way to reproduce

Step 2: Collect Clues

- Collect clues as to bug
  - Clues suggest where problem might be
  - Ex: if crash using projectile, what about that code that handles projectile creation and shooting?
- And beware that some clues are false
  - Ex: if bug follows explosion may think they are related, but may be from something else
- Don’t spend too long - get in and observe
  - Ex: see reference pointer from arrow to unit that shot arrow should get experience points, but it is NULL
  - That’s the bug, but why is it NULL?

Step 3: Pinpoint Error

1) Propose a hypothesis and prove or disprove

- Ex: suppose arrow pointer corrupted during flight. Add code to print out values of arrow in air. But equals same value that crashes. *Hypothesis is wrong*. But now have new clue.
- Ex: suppose unit deleted before experience points added. Print out values of all in camp before fire and all deleted. *Yep, that's it*.

Or, 2) *divide-and-conquer* method (note, can use with hypothesis test above, too)

- Sherlock Holmes: "when you have eliminated the impossible, whatever remains, however improbably, must be the truth"
- Setting breakpoints, look at all values, until discover bug
- The "divide" part means break it into smaller sections
  - Ex: if crash, put breakpoint ½ way. Is it before or after?
- Repeat.
- Look for anomalies, NULL or NAN values

Step 4: Repair the Problem

- Propose solution. Exact solution depends upon stage of problem.
  - Ex: late in code cannot change data structures. Too many other parts use.
  - Worry about “ripple” effects.
- Ideally, want original coder to fix
  - If not possible, at least try to talk with original coder for insights.
- Consider other similar cases, even if not yet reported
  - Ex: other projectiles may cause same problem as arrows did

Step 5: Test Solution

- Obvious, but can be overlooked if programmer is sure they have fix (but programmer can be wrong!)
- So, test that solution repairs bug
  - Best by independent tester
- Test if other bugs introduced (beware “ripple” effect)

Debugging Prevention

- Add infrastructure, tools to assist
  - Alter game variables on fly (speed up debugging)
  - Visual diagnostics (maybe on avatars)
  - Log data (events, units, code, time stamps)
- Indent code, use comments (in Game Maker)
- Use consistent style, variable names
  - Ex: spr_boss, obj_boss
- Avoid duplicating code – hard to fix if bug
  - Use a script
- Avoid hard-coded values – makes brittle
- Always initialize variables when declared
- Verify coverage (test all code) when testing

Thank you