Assignment-2 ES
YASHVINAYVANSH
19BCS081

Q1 a) Application specific processors

- AS processors are processors with architecture and instruction set optimized to specific domain / application requirements like n/w processing, automotive, telecom, digital signal processing, control applications etc.
- It is needed when general purpose processor are unable to meet increasing application needs.

Usecase - Washing Machine

Actuators → Motorised Agitator, tumble tub, water drawing pump, inlet valve to control flow of water into the unit

Sensors → water temperature sensor, level sens

Control → Microcontroller based board with interfaces to sensors & Actuators
Sensor data is fed back to control unit and CU generates necessary actuator operations.

CU also provides connectivity to user interfaces like keypad for setting the washing time, selecting type of fabric to be washed and other modes.

User feedback is reflects through the display unit & LEDs connected to the control board.

System operation of washing m/c.

1. user selects a wash pgm on selector dial.
2. user presses start button
3. door lock is engaged
4. water valve is opened to allow water into wash drum
5. If wash pgm involves detergent → detergent hatch is opened. After release, its closed.
6. when full water level is sensed, the water valve is closed.
7. If pgm involves hot water, water heater is turned on.
8. When water reaches ref. temp, heater switched off
9. Washer motor is turned on to rotate the drum. The motor then goes through a series of movements (at various speeds) to wash the clothes.
(Precise set of movements depends on wash pgms)
10. At end of wash cycle, motor is stopped
11. Pump is switched on to drain the drum.
12. When drum is empty, pump switched off.
13. door lock released.
14. during operation, various LEDs are used to indicate where system is in its work cycle.

This can be represented by one of the state models of computation

Q1 b) Hardware software codesign

→ Increasing competition in market and reduced time to market lead to a novel approach for embedded system design in which h/w & s/w are codeveloped instead of independently developing both in the traditional approach.

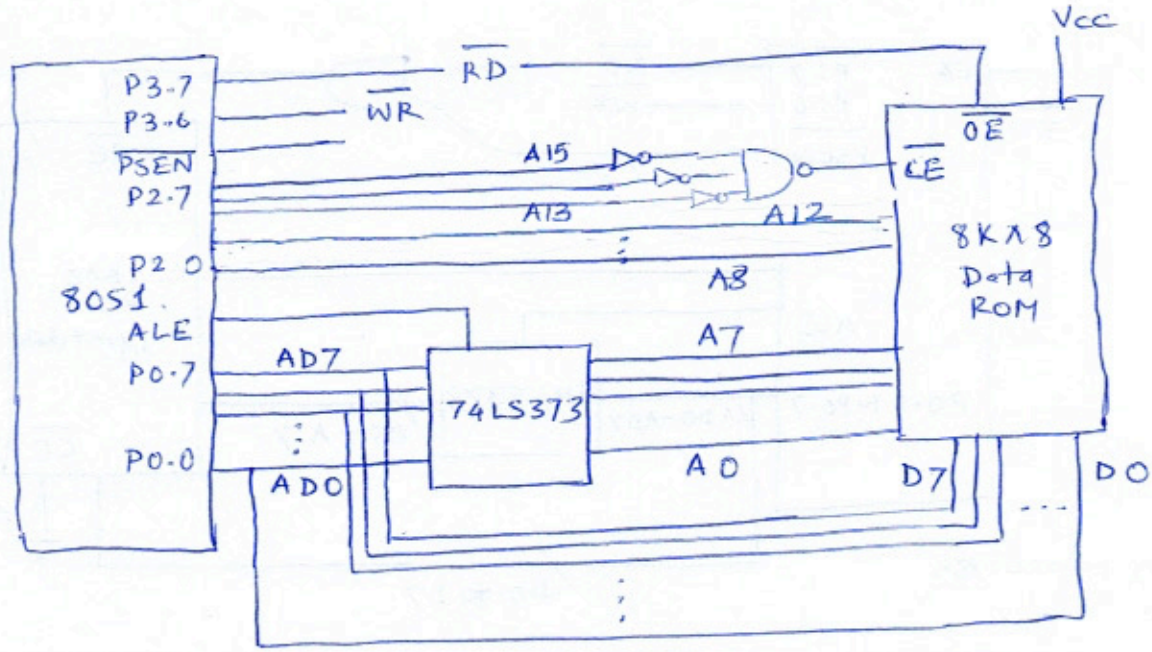
Steps

1. product requirements captured from customer
2. requirements converted to system level needs.
3. sys. level process, requirements transferred to function which can be tested against performance & functionality
4. Architecture design.
   - system level process requirements partitioned in h/w or s/w based on h/w - s/w trade offs.

Issues in h/w - s/w codesign.

1. Selecting the model:
   The model requirement may change with each phase of development which might require switching b/w variety of models.
2. Selecting the architecture.
   how to implement a system in terms of number and types of different components and interconnection among them.
   commonly used architectures in system design.
   - Application specific
     - Controller architecture.
     - data path architecture.
   - General purpose
     - Complex instruction set computing architecture (CISC)
     - Reduced " " (RISC)
   - Parallel processing
     - Very long instruction word computing (VLIW).
     - SIMD
     - MIMD etc.
3. Selecting the language.
   - A PL captures a computational model & maps it into an architecture.
   - It can be procedural, object oriented etc
   - A model can be captured using multiple languages or a single language can capture many models as no hard & fast rule to specify which language to be used for a model.
   - eg. C, C++, C#, Java, VHDL, Verilog, ...
4. Partitioning System requirements into s/w & h/w
   - From implementation perspective, it's possible to implement system requirement in either h/w or s/w (firmware). It's tough to figure which one to opt to due to several s/w - h/w trade offs like high performance of h/w vs high reconfigurability of s/w.

Q2 a) External ROM interfacing to 8051 as data memory



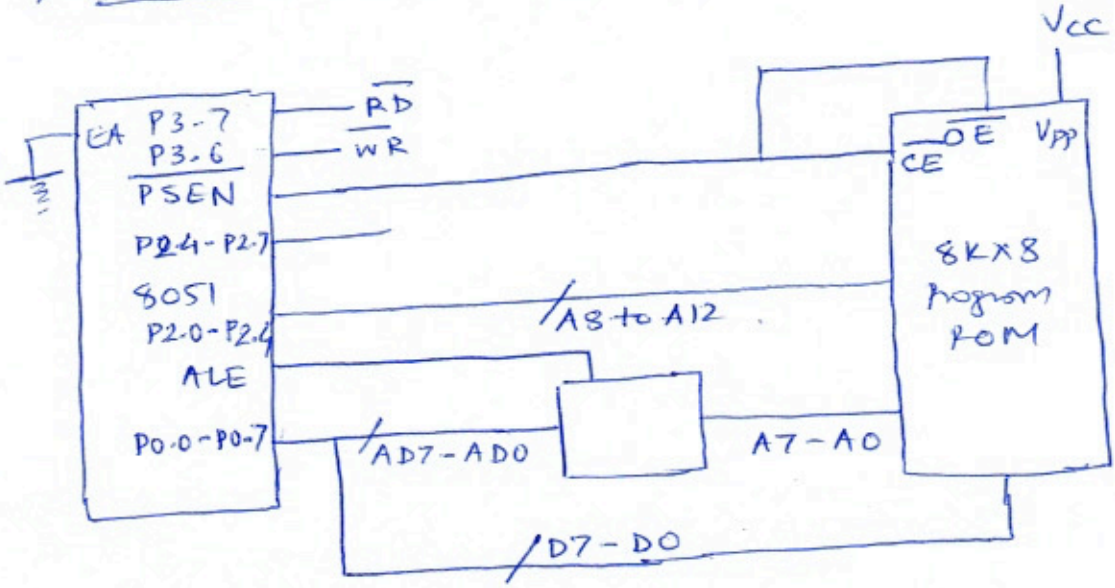
8051 has 216 = 128K bytes of byte addressable address space

74LS373 contains an array of D Flipflops clocked by ALE.

(8 bit)

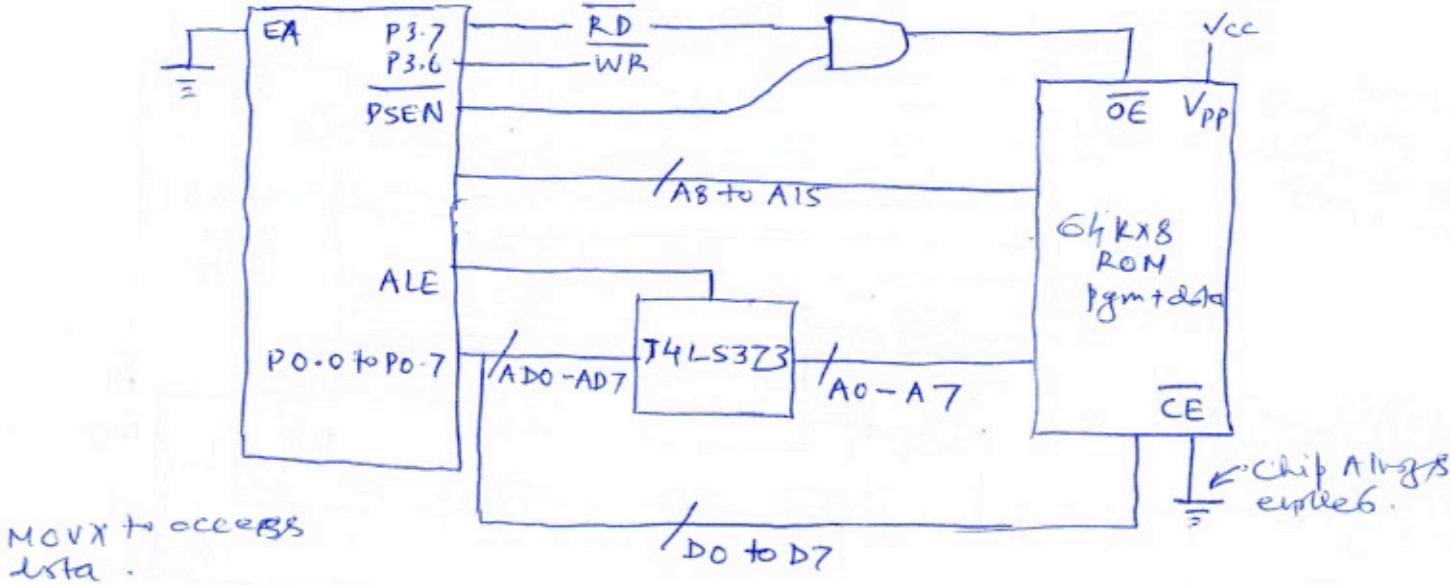
- Databus & lower byte of control bus (16 bit) are multiplexed using Address latch enable ALE signal.
- ALE = 0 ⇝ P0 is used for data path
  ALE = 1 ⇝ P0 is used for control path.
  74LS373 chip latches the address as ALE goes 0 & provides it to A0 - A7 while AD0 to AD7 specialised as data bus now.
- All A13 to A15 are ANDed to achieve chip enable signal. ($\overline{CE}$)
- control signal RD (read) when enabled, the chip outputs data at address (A0 to A12) to (D0 to D7) refer to output enable (OE)

b) External ROM interfacing to 8051 as program memory



- PSEN (Program store enable) signal is an o/p signal in 8051 which is connects to OE (output enable) pin of ROM catering program code
- when EA pin is connected to GND (ground), 8051 fetches opcode from external ROM using $\overline{PSEN}$ irrespective of on-chip ROM.

c) Extend ROM interfacing to 8051 as both data memory and program memory.



- To allow a single ROM chip to provide both code & data space, we use an AND gate to signal the OE pin of ROM chip.
  eg. 0000 - 7FFFH can be allocated to pgm code
  8000 - FFFFH can be used for data.

- When RD or PSEN is enabled it means 8051 wants to access data or code at address provided on A0 to A15. Hence $\overline{RD}$. $\overline{PSEN}$ is connected to output enable of ROM.

Q 2 b) Mechanism in 8052 to resolve collision between upper memory & SFRs.

1. To address upper memory from 80 - FFH, only indirect addressing mode is used which uses R0 & R1 registers as pointers with values of 80H or higher.

   eg MOV @R0, A
   MOV @R1, A

2. The same address space 80 - FFH is assigned to SFRs which can be accessed only using direct addressing mode

   eg MOV 90H, #55H
   ≡ P1.

∴ The addressing mode resolves the conflict of same memory address space assigned to both upper RAM & SFRs in 8052

A pgm to copy a message from on chip ROM code space upper memory & also showing each copied byte to P2

```
        ORG 0000H
        MOV DPTR, #MESSAGE.      // to access upper memory
        MOV R1, #80H.
HERE:   CLR A.
        MOVC A, @A+DPTR          // copy code from ROM to acc
        MOV @R1, A               // place it in upper memory
        MOV P2, A.               // Also move acc to P2
        JZ EXIT                  // exit if its the last byte
        INC DPTR                 // increment ptr of message.
        INC R1                   // increment ptr R1 in upper mem
        SJMP HERE                // repeat until last byte of
EXIT:   SJMP $.                  // message (which is 0)
                                 // stay here when finished.

        ORG 300H.
MESSAGE: DB "YASH VINAYVANSHI", 0
        END
```

Q 3 a)

Approach :
convert Binary (Hexadecimal) Number to decimal no. and then convert decimal number to its ASCII value.

```
RAM_ADDR  EQU  40H
ASCII_RES EQU  50H
CNT EQU 3
```

Aliases.

```
ORG 0
ACALL BIN_TO_DEC
ACALL DEC_TO_ASCII
SJMP $.
```

// main funch
// call binary to dec. conversion
// call dec. to ASCII conversion

```
BIN_TO_DEC : MOV R0, # RAM_ADDR
MOV A, P1
MOV B, #10.
DIV AB
MOV @R0, B
INC R0
MOV B, #10
DIV AB.
MOV @R0, B
INC R0
MOV @R0, A
RET.
```

(converting binary (HEX) to decimal 00-FF to 000-255)

// save decimal digits in these 3 RAM locations
// read data from port 1.
// B = $(10)_D$ or $(0A)_H$.
// divide A by B ie 10.
// save ones digit in 40H
// increment ptr
// B = $(10)_D$ or $(0A)_H$.
// divide A by B ie 10.
// save tens digit in 41H.
// increment pointer
// Save hundredths digit in
// return to main

```
DEC_TO_ASCII : MOV R0, # RAM_ADDR.
MOV R1, # ASCII_RES
MOV R2, # 3
BACK : MOV A, @R0
ORL A, # 30H.
MOV @R1, A
INC R0
INC R1
DJNZ R2, BACK.
RET.

END.
```

(Converts DEC digits to displayable ASCII digits)

// addr of decimal No.
// addr of ASCII No.
// Max. no. of decimal digits in 8 bit binary no = 3. (ob
// get decimal no. digit
// OR it with 30H to get ASCII of decimal digit
// Save it
// next digit of decimal no
// next digit of ASCII equivale
// repeat until MSD of decimal no. is converted to ASCII equivalent

Illustration

$(10101010)_B$

↓

$(AA)_H$ or $(170)_D$.

Integer Arithmetic : floor.

170/10 = 17, 170%10 = 0
17/10 = 1, 17%10 = 7

40H	0
41H	7
42H	1

OR 30H to each digit

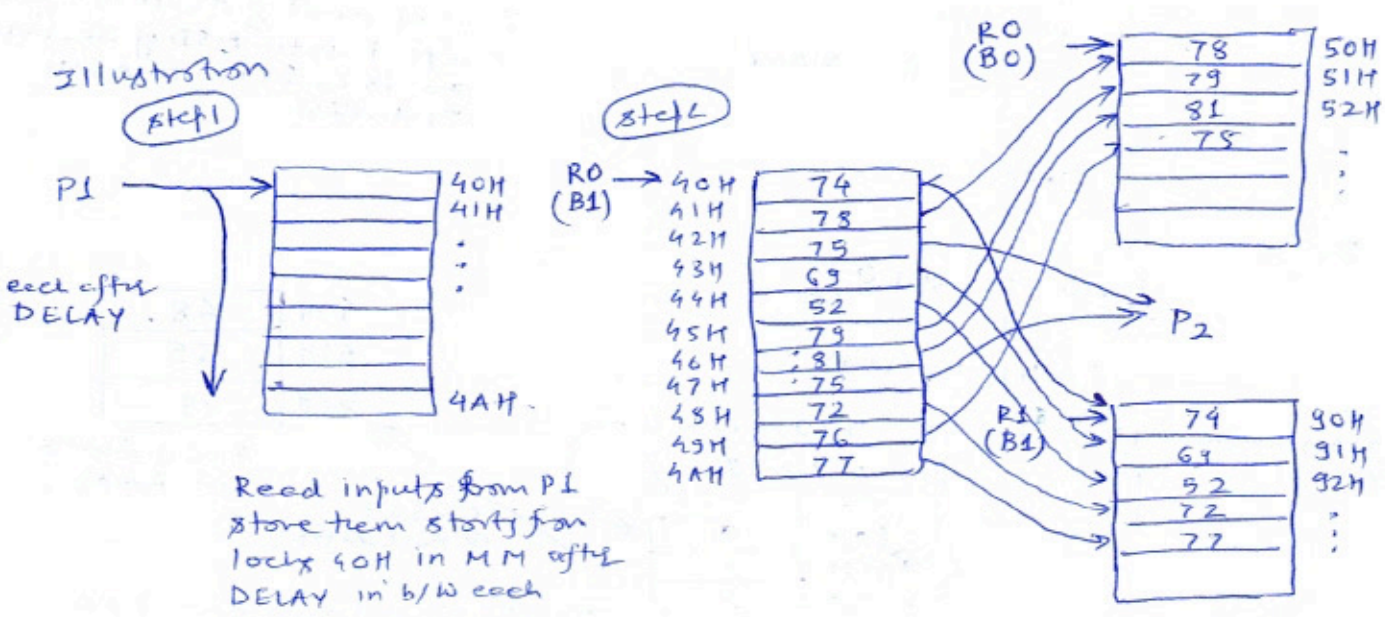
50H	48
51H	55
52H	49

ASCII digits in RAM.

extracted decimal digits in RAM

Q3 b)

```
        MOV  A, #0FFH.        // Make P1 an i/p port by sending all 1s ie FFH
        MOV  P1, A

        MOV  R0, #40H.        // Addr. where i/p is to be stored
        MOV  R1, #0AH         // counter init to 10.
HERE:   MOV  A, P1            // get input from port 1 into acc.
        MOV  @R0, A           // move data from acc. to RAM.
        INC  R0               // increment pointer
        ACALL DELAY           // Assuming Synchronous input take next input after delay caused by function DELAY.
        DJNZ R1, HERE         // Do it 10 times.

        SETB PSW.3            // switch to bank 1
        MOV  R0, #40H.        // R0 of bank 1 is pointer to i/p data in RAM
        CLR  PSW.3            // switch to bank 0
        MOV  R0, #90H         // set R0 of bank 0 as ptr to loc^n 50H
        MOV  R1, #0AH.        // set R1 of bank 0 as ptr to loc^n 90H

HERE:   SETB PSW.3            // switch to bank 1
        MOV  A, @R0           // set ptr to input
        CJNE A, #75, NEXT.    // jump if A ≠ 75.
        MOV  P2, A            // send 75 to P2 (Def. output)
        INC  R0               // increment R0 ptr
        SJMP HERE.            // repeat inspection.

NEXT:   JNC  NEXT1.           // if A < 75
        CLR  PSW.3            // switch to bank 0
        MOV  @R0, A.          // store number < 75 at R0 of bank 0
        INC  R0               // increment R0
        SJMP HERE             // repeat inspection

NEXT1:  CLR  PSW.3            // if A > 75 switch to bank 0
        MOV  @R1, A           // store number > 75 at R1 of bank 0
        INC  R1.              // increment R1
        SJMP HERE             // repeat inspection.
```



Q4 a) Timer's clock period

$f = \frac{1}{12} \times 22MHz \approx 1.833\ MHz$

$T = 1/1.833MHz \approx 0.546\ \mu s$.

Timer initialization for 50 ms delay

$\frac{50\ ms}{1.085\ \mu s} = 46083$. → using Mode 1.

$65536 - 46083 = 19453 = (4BFD)_H$.

Timer initialization for 40 ms delay

$\frac{40\ ms}{1.085\ \mu s} = 36866$

$65536 - 36866 = 28670 = (6FFE)_H$.

Timer initialization for 15 ms.

$\frac{15\ ms}{1.085\ \mu s} = 13825$

$65536 - 13825 = 51711 = (C9FF)_H$.

```
        MOV TMOD, #01H.        // timer 0 Mode 1 (16 bit).
        CLR P1.5.              // waveform starts from low
HERE:   MOV TL0, #FDH          ] Initialise TL0, TH0 with values
        MOV TH0, #4BH.         ] required for timer to create delay of 50
        ACALL DELAY            ] on
        ACALL DELAY            ] off.
        MOV TL0, #FEH          ] Initialise TL0, TH0 with values
        MOV TH0, #6FH.         ] required for timer to create delay of 40
        ACALL DELAY            ] on
        ACALL DELAY            ] off.
HERE1:  MOV R0, #04H.          // Initialise counter to 4.
        MOV TL0, FFH           ] Initialise TL0, TH0 with values
        MOV TH0, C9H           ] required for timer to create delay of 15
        ACALL DELAY            ] on
        ACALL DELAY            ] off
        DJNZ R0, HERE1         // repeat until counter of 4 exhausts
        JMP HERE               // continue whole process again.

DELAY:  CPL P1.5               // invert P1.5.
        SETB TR0               // start timer 0.
AGAIN:  JNB TF0, AGAIN         // timer 0 runs
        CLR TR0
        CLR TF0.
        RET.
```

Q4 b)

timer 1				timer 0			
GATE	C/T	M1	M0	GATE	C/T	M1	M0
0	0	0	0	0	1	0	1

// counter 0 mode 1

TMOD register.

```
        MOV TMOD, #00000101B    // counter 0 Mode 1.
                                // Make T0 input.
        SETB P3.4.
AGAIN : MOV TH0, #00H           ] clear : TH0, TL0
        MOV TL0, #00H.
        SETB TR0.               // start counter 0.
BACK :  MOV A, TL0              // get copy of TL0
                                   (lower byte of current count)
        MOV P0, A               // display on P0
        MOV A, TH0              // get copy of TH0
                                   (upper byte of current count)
        MOV P1, A               // display on P1.
        JNB TF0, BACK           // keep doing till TF = 0
        CLR TR0                 // stop counter 0
        CLR TF0.                // clear TF0 flag.
        SJMP AGAIN              // restore the timer back
                                   to 0000H & start again.
```

Assuming XTAL = 22MHz which is very large than external pulse frequency of 1Hz, therefore the body of program will be executed safely within the next 1hz pulse comes and therefore, the pulses will be counted correctly.

Q5 a) Assuming square wave is of 1 kHz, XTAL = 11.0592 MHz
Assuming baud rate is 4800 bps.

$f = 1\,kHz \rightsquigarrow T = \frac{1}{1kHz} = 1\,ms \rightsquigarrow \frac{T}{2} = \frac{1ms}{2} = 500\,\mu s$.

$\#cycles = \frac{500\,\mu s}{1.085} = 461 \rightsquigarrow 65536 - 461 = 65075 = (FE33)H$

```
        ORG 0000H
        LJMP MAIN
        ORG 000BH
        LJMP WAVE   // jump to interrupt service routine
                    for square wave.
        ORG 0003H
        LJMP LED.   // jump to ISR for LED activity
                    as given in question.
        ORG 0023H
        LJMP SERIAL // jump to serial interrupt ISR.
        ORG 0030H
MAIN:   MOV P2, #0FFH   // Make P1 input port
        MOV TMOD, #21H  // timer 1 mode 2 (auto reload)
        MOV TH1, #0F5H  // 4800 baud rate  timer 0 mode 1 (for sq wave)
        MOV TL0, #033H  ]
        MOV TH0, #0FEH  ] initialise TL0, TH0 for timer 0 used
                          in generating 1 kHz sq. wave.
        MOV SCON, #50H  // 8 bit, 1 stop, ren enabled
        MOV IE, 10010011B // Enable serial port interrupt, timer 0 overflow
                          interrupt, external interrupt 0.
        SETB TR1.       // start timer 1
        SETB TR0        // start timer 0 to generate square wave
HERE:   MOV A, P2       // take data from P2 into Acc.
        MOV SBUF, A     // put data from acc to SBUF to start
                          sending it serially
        SJMP HERE       // continue sending serially.

        ORG 100
LED:    CLR P0.         // called when INT0 activated : clear Port P0
        MOV R0, #255    // for delay
BACK:   DJNZ R0 BACK    // exhaust R0
        MOV P0, 0FFH    // restore P0 to high.
        RETI

SERIAL: JB TI NEXT      // if transfer complete
        MOV A, SBUF     // otherwise due to receive
        CLR RI          // clear RI since CPU doesn't return from ISR
        RETI
NEXT:   CLR TI          // if transfer complete, clear TI flag.
        RETI.           // return & enable interrupts of equal & low
                          priority interrupts than this interrupt.
WAVE:   CPL P1.2.       // generate next pulse of square wave.
        MOV TL0, #033H  ]
        MOV TH0, #FEH   ] reinitialise TH0 & TL0 for timer 0
        RETI              for 1 kHz pulse's delay.
```

Qs b)

```
        ORG 0000H
        MOV TMOD, #22H      // timer 0 & timer 1 both in Mod
        MOV SCON, #50H
        MOV TH1, #-3.       // Set Baud rate to 19200
        MOV TH0, #00H.      // to run for 256 times.
        SETB TR1.           // start timer 1.
        MOV A, #00H         // A = 00H
        CLR P1.2.           // P1.2 = 0.
BACK:   SETB TR0            // start timer 0.
AGAIN:  JNB TF0, AGAIN      // wait for timer 0 to complete one cy
        CPL A.              // complement A
        CPL P1.2.           // complement P1.2 for sq. wave gen
        MOV SBUF, A.        // transfer content of A to SBUF r
                            // so that it can be transferred serial
        CLR TR0             // stop timer 0
        CLR TF0.            // clear TF0 flag.
HERE:   JNB TI, HERE        // wait until TI flag indicates comple
        CLR TI              // clear TI.  transfer of SBUF
        SJMP BACK           // continue.
```