Basinatris 300524583

EEEN202 Timer Lab 1

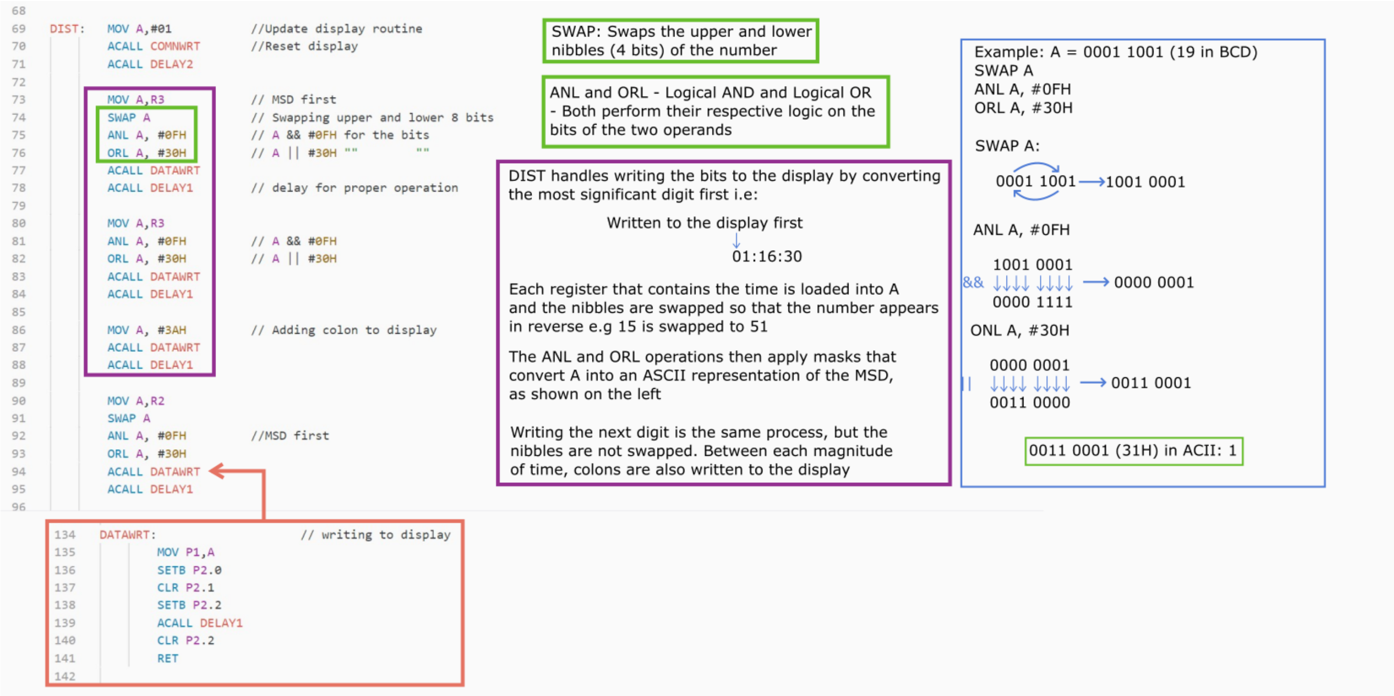
Using the AT89C51AC3 to create a clock displaying seconds, minutes, and hours; coded in assembly in two different modes: polling and interrupt

Operation of Timer Unit: Core/Common functions

At its core, the timer unit operates in a loop of incrementing a set of 3 registers – which represent seconds, minutes, and hours – which cascade into one another and updating the display. How the timer functions differs depending on whether it is polling or using interrupts, but several core parts remain the same:

- Set/update display routine
- Increment routine

To update the display, DIST is called; this routine writes the current time hours first then minutes and second, with colons separating them.



Whenever the display needs to be reset, SETDIS is called which clears the display:

The increment routine handles updating the values of the registers which hold the second, minutes, and hours – R1, R2, and R3 respectively:

```
INCT:   MOV A,R1           //Update time count routine (seconds)
        ADD A, #1
        DA A
        MOV R1, A
        CJNE A, #60H, INCE

        MOV R1, #0       //Update time count routine (minute)
        MOV A,R2
        ADD A, #1
        DA A
        MOV R2, A
        CJNE A, #60H, INCE

        MOV R2, #0       //Update time count routine
        MOV A,R3
        ADD A, #1
        DA A
        MOV R3, A
        CJNE A, #24H, INCE
        MOV R3, #0

INCE:   RET
```

INCT increments R1 by one every time it is called; if 60, then R1 is reset and R2 is increment and so on

DA - Decimal adjust:
- Converts an 8 bit binary digit into two 4 bit BCD digits

E.g: 0001 1000 (24) after DA is 0010 0100

CJNE - Compare then jump if not equal:
- Jumps to indicated branch if first operand not equal with second

This routine is called every second; how the microcontroller achieves this is by using a counter to divide the clock frequency of the microcontroller – which is 12 MHz – to 1 Hz increments. The layout is slightly different for each style, but the execution is the same: Timer 0 is set to 16 counter mode and because 16 bits are not enough to divide the clock frequency down to a 1 second period, R0 is used as another counter to divide the frequency more (NOTE: the following code snippets are a mix from both styles):

```
MAIN:   MOV R0,#20
        MOV R1,#0            // Set time value = 0, seconds
        MOV R2,#0            // minutes
        MOV R3,#0            // hours
        ACALL SETDIS         // initialise the display
        MOV TMOD,#0x01       // set timer 0 to mode 1: 16 bit counter
        MOV TH0,#0x3C        // set lower 8 bits to 0x3C
        MOV TL0,#0xB0        // set upper 8 bits to 0xB0
```

```
                             // resets timer 0 if R0 != 0   &&
REPEAT: MOV TH0,#0x3C        // sets upper 8 bits to 0x3C
        MOV TL0,#0xB0        // sets lower 8 bits to 0xB0
        SETB TR0             // enable timer 0

                             // main program loop    ||
WAIT:   JNB TF0,WAIT         // does Timer 0 flag = 1?
        CLR TR0              // disable timer 0
        CLR TF0              // sets TF0 to 0
        DJNZ R0,REPEAT       // decrease R0 - is it 0?
        MOV TH0,#0x3C        // reset TH0
        MOV TL0,#0xB0        // reset TL0
        SETB TR0             // enable timer 0
```

TH0 = 0x3C
TL0 = 0xB0

T0 is set to 16 bit mode; this means T0 contains: 0x3cB0 or 15536

65535 - 15536 = 49,999

This means T0 counts 50000 times until the timer overflow flag TF0 is triggered

JNB: Jump if bit not set:
- Jumps to indicated address if first operand not set

In this instance, it is waiting for the timer flag

DJNZ called here decreases R0 until zero. Otherwise, Timer 0 is reset and program loops. This means that the timer has run 20 times when the program is allowed to continue

In TMOD 1, the timer triggers every 12 cycles, Therefore:

12 MHz / (20 * 50000 * 12) = 1 Hz

In other words, the seconds register can increment every second,

So, by using a register as a secondary counting stage, the frequency of the microcontroller clock can be stepped down to be used to accurately increment the time registers

Operation of the Timer Unit: Polling Mode

In polling mode, the program

Firstly, timer goes through an initialisation stage where the registers used for counting the time are set and the timer enabled:

```
        ORG 0H             // Indicates what address the code starts
        MOV R0,#20         // Moves decimal 20 into register 0
        MOV R1,#0          // Set time value = 0, seconds
        MOV R2,#0          // minutes
        MOV R3,#0          // hours
        ACALL SETDIS       // initialise the display: Absolute call
        MOV TMOD,#0x01     // Sets timer into mode 1: 16 bit counter
```

- ORG sets the start location of the PC
- R0 is set to 20; R1, R2, and R3 are set to 0.
- SETDIS function is called using ACALL
    - ACALL – Absolute call: Unconditional call to a subroutine located in the same 2K block of memory as the current instruction
- TMOD is set to 16 bit counter mode

Afterwards, the program first executes REPEAT (which enables the timer) and into WAIT, the main timer loop:

```
                          // resets timer 0 if R0 != 0
REPEAT: MOV TH0,#0x3C     // sets upper 8 bits to 0x3C
        MOV TL0,#0xB0     // sets lower 8 bits to 0xB0
        SETB TR0          // enable timer 0

                          // main program loop
WAIT:   JNB TF0,WAIT      // does Timer 0 flag = 1?
        CLR TR0           // disable timer 0
        CLR TF0           // sets TF0 to 0
        DJNZ R0,REPEAT    // decrease R0 - is it 0?
        MOV TH0,#0x3C     // reset TH0
        MOV TL0,#0xB0     // reset TL0
        SETB TR0          // enable timer 0
        MOV R0,#20        // reset R0
        CPL P2.3          // output every second
        ACALL DIST        // Display time
        ACALL INCT        // Increment time
        AJMP WAIT         // reset back to top
```

Here the polling system is shown in effect; JNB TF0 makes the WAIT routine loop until T0 finishes counting. Then the timer is reset and R0 is decremented and the loop starts again until R0 is 0 – upon which the registers are incremented, the display is updated, and the system is reset to start counting again.

Operation of the Timer Unit: Interrupt Mode

In interrupt mode, the program does not wait for the counter to be finished. Once the timer flag is raised, it sends an flag which triggers an interrupt service routine, which loads the current instruction/PC and registers into the stack and then executes the ISR. The code starts out declaring the origin then long jumping to MAIN

```
        ORG 0H
        LJMP MAIN         //bypass interrupt vector table

        ORG 000BH         // Timer-Counter o interrupt vector
        LJMP TINT         // Jump to Timer counter interrupt routine

        ORG 30H

MAIN:   MOV R0,#20
        MOV R1,#0         // Set time value = 0, seconds
        MOV R2,#0         // minutes
        MOV R3,#0         // hours
        ACALL SETDIS      // initialise the display
        MOV TMOD,#0x01    // set timer 0 to mode 1: 16 bit counter
        MOV TH0,#0x3C     // set lower 8 bits to 0x3C
        MOV TL0,#0xB0     // set upper 8 bits to 0xB0
        SETB EA
        SETB ET0          // Enable timer 0 interrupt (your task)
        SETB TR0          // Start timer
```

Main contains all the set up for the registers and timer elements, as well setting up the necessary interrupt enables (EA, ET0). Afterwards, the program sets the PC location to the address of the interrupt vector and executes TINT:

```
TINT:   CLR TF0           // Timer - counter interrupt service rountine ISR
        CLR TR0           // disable timer 0
        MOV TH0,#0x3C     // reset initial timer values
        MOV TL0,#0xB0
        SETB TR0          // enable timer 0
        DJNZ R0,TINTE     // decrease R0 - is it 0?
        ACALL INCT        // if R0 = 0, INCT registers
        MOV R0,#20        // reset R0 count
TINTE:  RETI


        END
```

TINT works like the point in polling when the TF0 is set; T0 is reset and R0 is decremented. If R0 = 0, then INCT is called and R0 is reset. Then the program counter origin is set to 30H, and the main loop, DISPL, is executed:

```

DISPL:  ACALL DIST          // Display time loop
        ACALL DELAY3
        AJMP  DISPL         // loop back to start


```

Compared to polling, this loop is incredibly small, and all it does is ACALL DIST to refresh the display, pauses, then loops back. This is all because of the ISR set up – all the timer resetting and incrementing operations is handled during the interrupt. More importantly, instead of the program checking every cycle if TF0 is set, the program checks if R0 is zero 20 times a second, reducing the idle time of the program.

Pros and cons of both styles

A polling program must wait for the data it requests to be ready, meaning the whole program halts until received; interrupt-based programs simply continue a flag is raised and the interrupt service routine is executed – after which the previous instructions are loaded back in seamlessly. Because of the zero-wait time needed, interrupt routines can be much more efficient, particularly in programs which have much more instructions than just simply updating a clock display. It also means that the CPU is not wasting resources by constantly checking whether the data is done processing or not – it just continues with whatever instruction is next and then processes the ISR when it appears. ISRs also have high priority to the CPU – if the process finishes just as the CPU is finished checking on it, the CPU misses it and must complete the current cycle and go back to pick it up. ISRs are absolute and will immediately flag the CPU, decreasing the likelihood of being missed and making them great for events whose completion times are random.

Nevertheless, polling still has its uses. Its simplicity means it is great when the system is in early development and easy to debug. There are also no big memory management issues associated with polling, whereas ISRs can lead to data loss or stack management issues. ISRs must not be frequently used otherwise one runs the risk of the stack pointer being completely full due to too many ISRs occurring at once. They should also be kept small enough as they still delay the execution of the main code; the goal of ISRs is to be seamless, and larger routines may be noticeable. In non-time sensitive applications, polling can also make much more sense as interrupt just introduces unneeded complexity. It also makes sense in applications where you want the program to run sequentially in an expected order – interrupts would just disrupt that.

So, both styles are useful and can make up for the other's downsides; which one you use just depends on the context of your program.

Pros and cons of Low-level vs High-level language

Assembly provides users with as level of control and efficiency that higher level languages do not. It provides direct and accurate control of the microcontroller's resources such as memory, ports, etc. This fine access means that Assembly code can be written much more efficiently and with higher density than higher level languages, which require a compiler to translate the human friendly language it into machine language. Depending on how intelligent compiler is in recognizing certain short cuts – such as using bit shifting operations in place of more complicated math – the resulting machine code can be more efficient when converted. Assembly on the other hand, is directly translated into binary using assemblers, which generally do not add optimisations and simply convert the instructions into their machine language equivalents.

However, Assembly is much less user friendly than higher level languages like C. Assembly uses mnemonics, whereas C is written using English statements. Assembly instruction sets are also machine dependent, meaning if you want to program another microcontroller, you will need to learn another set of instructions. To program in Assembly effectively, you also need knowledge of the microcontroller and its resources – the vector address or timers, what registers you have access to, the size of flash memory available – whereas C requires very little knowledge of whatever device you are writing to as compilers handle all the required translations into the machine dependent code.

To summarize, Assembly is great for getting as close to metal as possible and producing very efficient programs, but is not user friendly and requires knowledge of the hardware to even begin programming properly. Higher level languages offer the safety net of code statements being legible to most and requiring very little knowledge of the hardware being used in exchange for being slower, much less efficient, and bulkier than Assembly.

Additional Questions

1. Write an instruction sequence that could be used to read bit 1 of Port 0.

MOV A, P0.1

2. What addressing mode is used to access the upper 128 bytes of internal RAM?

Indirect addressing mode

3. Show how the content of internal address 6BH could be transferred to the accumulator.

MOV R0, #6BH
MOV A, @R0

4. What is the difference between the following instructions: ADD A,@R5 and ADD A,R5

@ is the indirect address operator, used to signify that the following bits are an address pointing to another location which contains the values. ADD A, @R5 essentially means add the contents of the memory location pointed to by R5, which could contain an 8-bit address, to the accumulator. ADD A, R5 just means add the contents of R5 to A.

5. Below shows a sequence of instructions, give the result of accumulator before and after the DA instruction.

```
MOV A,#13H
MOV R2,#18H
ADD A, R2
DA A
```

Before DA (2B): 00101011
After DA (43): 0100 0011

6. Explain the difference between AJMP, SJMP, and LJMP instructions.

All three do the same instruction of transferring the program execution to another part of the memory; the main difference comes from the maximum distance the program can jump:

- AJMP (Absolute short range jump): contains an 11 bit address, so destination must be within the same 2kB block of memory
- LJMP (Long range jump): contains an 16 bit address, meaning destination can be anywhere in the full 64kB space
- SJMP (Short range jump): contains a signed 8 bit address, meaning destination must be within -128 and 127 of the instruction

7. Describes what happens when the ACALL instruction is executed.

ACALL unconditionally calls the subroutine indicated by the address in the line. The program counter (PC) increases by 2, pushes the currently stored address onto the stack, and increments the stack pointer twice. The PC is then loaded with address called and the program continues from the new address

8. Write an instruction which is able to complement bit 7, 6, 2 and 0 of Port 2

CPL P0.7

CPL P0.6
CPL P0.2
CPL P0.0

9. What is the advantage of using EQU directive in an assembly language program?

Using EQU directives to create symbolic constants streamlines the coding flow if there is a particular value that is repeated often or if mathematical constant is needed – for example, if the program is calculating circles. It also makes changing the value of said symbolic constants easy as you only need to change one line instead of however many.

10. Describe what happens when an enabled interrupt is detected.

When the interrupt flag is triggered, the PC and registers are pushed to the stack while the interrupt service routine is executed. Once finished, the registers and PC are popped off the stack and the program continues from where the interrupt occurred.

11. Describe how you would implement a 20ms delay using timer 0. Assume the clock into the timer is 1MHz.

12. What is the vector address of Timer 1?

001BH

13. What is the next available memory where the user can write a program without interfering with the interrupt vector? Give an example of program code.

14. Upon exiting Reset, what is the contents of the stack pointer?

Upon reset, the stack pointer is set to the address 07H

15. Upon exiting Reset, which instruction is first executed

Instruction at the address 0H