XMUT202 Digital Electronics

8051 Assembly Languages

Week __ Lecture __

Agatha Rachmat

School of Engineering and Computer Science

Victoria University of Wellington

Today's topic

- Introduction to programming.
- Assembly language program.
- 8051 instruction sets.
- Program examples.
- Addressing modes.
- Register programming.
- Bit programming.

Conceptualising A Program

As engineers, we're often given tasks and are expected to practically realise them.

- "Make a robot that sweeps the floor!"

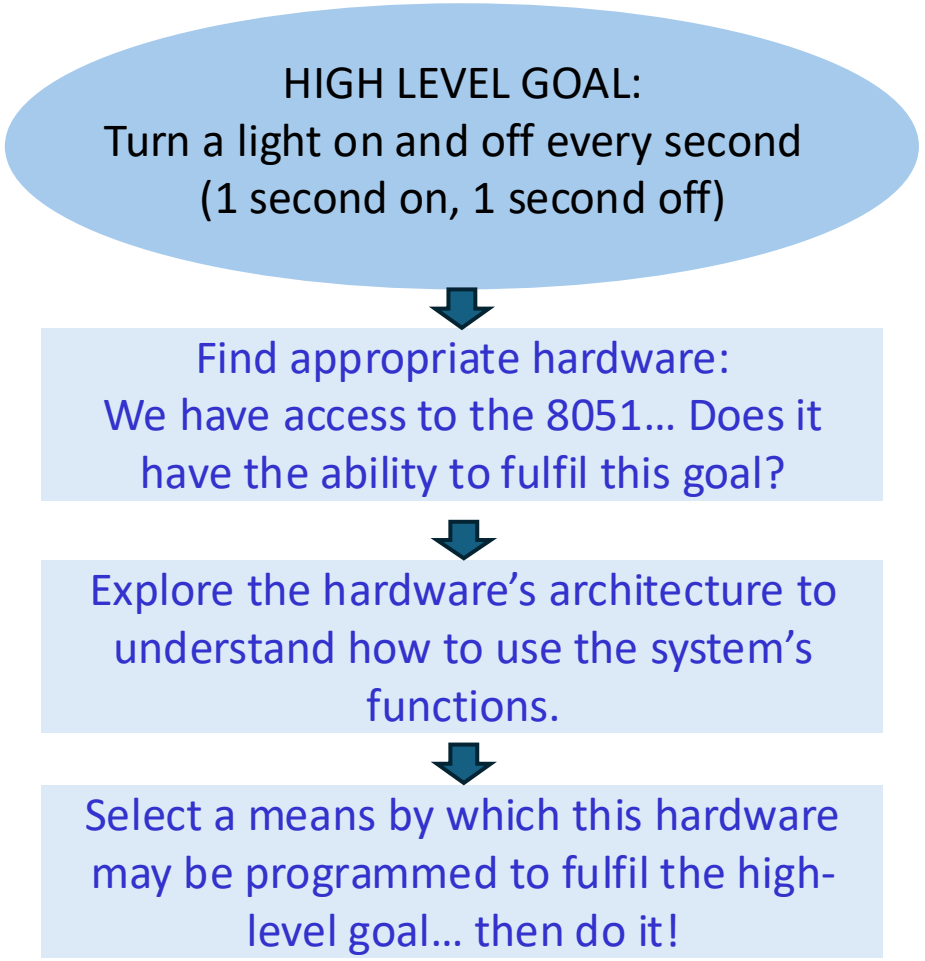
As embedded systems engineers, we need to be able to:

- Take a very high-level goal…
- and rationalise it with the tools that we have available.

If we are told: "Turn a light on and off

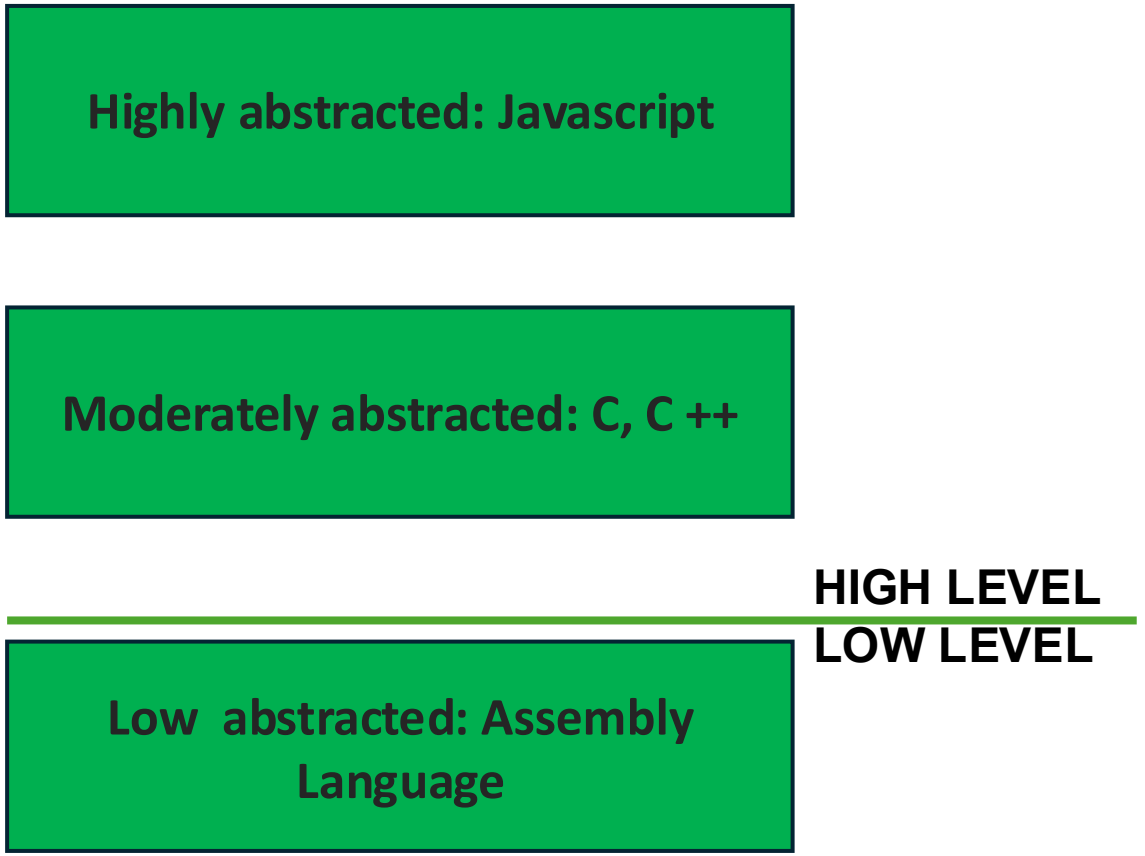
every one second!"…

- We must figure out how to do that with the available tools.



Programming Language Hierarchies

- Computer programming languages may be thought of as *high-level* or *low-level*.
- High-level: high levels of abstraction between language and hardware.
- C, Python, Javascript, LabView, Lua, C#, Java, Ada, Smalltalk, Swift, Forth…
- There are varying levels of abstraction: C has less than, say, Javascript
- Low-level: Programming commands are very closely related to the hardware's actual operation.
- Assembly languages are considered low-level: often one line per instruction cycle.
- Machine code is the lowest of all.



Assembly Languages

- Every different computer architecture has unique commands that allow it to execute programs.
- At the very lowest level, these are electrical signals that correspond to LOGIC HIGH and LOGIC LOW (0 or 1). Groups of these form instructions used by the computer.
- Machine code might look like this: 10010111 10011101 11101110 10011111
- This is very hard for most mortals to read, understand, debug, and expand
- These instructions and memory locations may be represented by *mnemonics*: easier-to-read, easier-to-remember codes that correspond to the machine code.

Mnemonic	Machine Code (Binary)
NOP	0000 0000
ADD A, R3	0010 1011

Assembly Languages

- We're used to counting in base-10 (decimal, 0d) because we have 10 fingers!
- Computers that use HIGH/LOW logic utilise a base-2 scheme: binary (0b)
- It can get very clunky and verbose to represent digital systems using binary: you end up with lots and lots of 0's.
- To make things more compact, we often use hexadecimal (0x or #nH).
- This is a counting system that goes from 0d0 to 0d16 (0x0 to 0xF)
- 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- 1 hexadecimal digit represents 4 binary digits.

Assembly Languages

- These 4 binary numbers form half of a byte, often called a ‘nibble’
- 0b0000 0000 is the same as 0x0 0x0
- Hexadecimal is commonly used in low-level software development: machine codes and addresses are easily represented this way.

Mnemonic	Machine Code (0b)	Machine Code (0x)
DIV AB	1000 0100	0x84
MOV A, #data	0111 0100	0x74

CISC vs. RISC

Instruction set: the collected instructions (in machine code) that are executable by a CPU.

CISC

Complex instruction set.

Many detailed operations that allow for single instructions to have fine- grained control over the computer.

RISC

Reduced instruction set

As high-level languages became prevalent in the 1980's and 1990's, there was less need for very specific processor instructions.

CISC vs. RISC

CISC

- Common in many early computers
- Good for low-level programming, as fewer commands are needed.
- Pentium, 8051, etc. are CISC
- Assembly language programs for CISC systems are often relatively human-readable

RISC

- Compilers could make use of a restricted (and fast!) instruction set to efficiently realise high-level programs.
- ARM processors are the most prevalent RISC processors today.
- RISC assembly language programs are significantly less human-readable.

8051 Instructions

8051 instructions are one byte, and are often organised in a table with the high nibble on one axis and the low nibble on another:
https://www.win.tue.nl/~aeb/comp/8051/set8051.html

- In the Lab Exercise, you will find a PDF called at_c51ism.pdf (note that the axes in this document are arranged opposite to those in the above URL)
- This contains a detailed list of the 8051's instructions. You are strongly encouraged to explore this document.

Further resources:

- KEIL 8051 instruction set reference:
http://www.keil.com/support/man/docs/is51/is51_instructions.htm
- Instructions organised by function:
https://www.engineersgarage.com/tutorials/8051-instruction-set

8051 Selected Instructions

You aren't expected to memorise the 8051's instruction set.

• Should the test contain instruction set-related questions, you will be provided with reference materials that you may consult during the test.

INSTRUCTION MNEMONIC	SYNTAX	Example OpCode	Number of bytes	Notes
NOP	NOP	0x00	1	Causes no operation that cycle
MUL	MUL AB	0xA4	1	Multiply accumulator by B
MOV	MOV operand1, operand2	0x74	1-3	Copies the val. of operand1 to operand2
INC	INC register	0x04	1-2	Increments the value of the register by 1

8051 Flags Intro

When the 8051's processor enters certain states, it raises 'flags' to indicate these states.

- These flags are stored in the Program Status Word register (PSW),
- which uses six of the register's eight bits.
- We'll explore some examples of these flags.

PSW.7 CY (Carry flag, raised when the processor needs to carry in addition)	PSW .6 AC (Aux. carry, used during BCD math)	PSW.5 F0 (User-assignable flag)	PSW.4 RS1 (Register bank selector, don't worry about for now)	PSW.3 RS2 (Register bank selector, don't worry about for now)	PWS.2 OV (Overflow, raised when a signed number overflows into the sign bit)	PSW.1 - (User assignable)	PSW.0 P (Parity: 0 if acc. holds even number of 1's)

Program Status Word Structure

- PSW flags are usually flags associated with the arithmetic operations.
- There are also other flags:
  - Specific flags used by special registers.
  - User-defined flags.
- Flags are useful for monitoring the condition of the execution of your program.

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
CY	AC	F0	RS1	RS0	OV	--	P

Bit	Symbol	Flag name and description		
7	C (or CY)	Carry; Used in arithmetic, logic and Boolean operations		
6	AC	Auxiliary carry ; useful only for BCD arithmetic		
5	F0	Flag 0; general purpose user flag		
4	RS1	Register bank selection bit 1		
3	RS0	Register bank selection bit 0		
		RS1	RS0	
		0	0	Bank 0
		0	1	Bank 1
		1	0	Bank 2
		1	1	Bank 3
2	0V	Overflow; used in arithmetic operations		
1	--	Reserved; may be used as a general purpose flag		
0	P	Parity; set to 1 if A has odd number of ones, otherwise reset to 0		

Example 1: A Simple Recursive Program

HIGH LEVEL GOAL:
Turn a light on and off every millisecond.
(1ms on, 1ms off)

Let's connect the LED to Port 1 and then toggle Port 1 between 0 and 1 every 1 ms.

The biggest challenge will probably be figuring out how to get a good precise timer to let the light stay on/off for 1 ms

```
START:
MOV A,#0FFH          ;Move 0xFF(1) to accumulator
MOV P1,A             ;Move accumulator value to P1

;TODO: delay for 1 ms!
MOV A,#00H           ;Move 0x0(0) to accumulator
MOV P1,A             ;Move accumulator value to P1

;TODO: delay for 1 ms again
SJMP START           ;Jump back to 'START'
```

SimpleDelay.asm

Note: Practical 8051 assembly language programs need a few other things to get working (e.g., setting the start address, specifying when the program has ended, etc.)

Example 2: Subroutine 1

In high-level languages, we often use functions to compartmentalise blocks of code that we might reuse.

- This allows us to avoid copy+paste of code.
- Somewhat similar to this is the assembly language concept of subroutines
- We can jump to particular blocks of code, execute them, and then jump back to our 'main' program.
- Let's try to do this with the 1 ms delay…

```
//pseudocode, high-level example of port writing
main(){
  Port1.write(HIGH);
  delay1Ms(); //call function routine
  Port1.write(LOW);
  delay1Ms();
}

function delay1Ms(){
   //code to make the CPU wait for 1 ms
}
```

Example 3: Subroutine 2

```
START:
MOV A, #OFFH     ;Move OxFF (1) to accumulator
MOV PI, A        ;Move accumulator value to P1
ACALL DELAY      ;Calls subroutine at 'delay'
MOV A, #00H      ;Move 0x0 (0) to accumulator
MOV PI,A         ;Move accumulator value to PI
ACALL DELAY      ;Delay for another 1 ms
SIMP START       ;Jump back to 'START'

DELAY:           ;1 ms delay Subroutine
MOV R6,#250D     ; Place 0d250 into Register 6
MOV R7,#250D     ;Place Od250 into Register 7
DEL1 :

DJNZ R6, DELI    ; DJNZ: Decrement R6 & jump if not 0
DEL2:
DJNZ R7, DEL2    ;DJNZ is 2-cycles, 2us to run. 2x500us-1ms
RET              ; Return to ACALL
```

MultiDelay.asm

Example 3: Subroutine 2

Challenge: Change this 1 ms delay to a 1 second delay.

Hint: call the delay 4 times in a row (4ms), then repeat this 4x call 250 times.

Example 3: Subroutine 2

Call the 1 ms delay **4 times** → gives about **4 ms**

Repeat that **250 times**

$4\ ms \times 250 = 1000\ ms = 1\ second$

```
ORG 0000H

START:

    MOV A, #0FFH        ; Load FFH into accumulator
    MOV P1, A           ; Output to Port 1
    ACALL DELAY1S       ; 1 second delay

    MOV A, #00H         ; Load 00H into accumulator
    MOV P1, A           ; Output to Port 1
    ACALL DELAY1S       ; Another 1 second delay

    SJMP START          ; Repeat forever
```

Example 3: Subroutine 2

```
;---------------------------------
; 1 ms DELAY SUBROUTINE
;---------------------------------

DELAY1MS:

    MOV R6, #250
D1:
    MOV R7, #250
D2:
    DJNZ R7, D2
    DJNZ R6, D1

    RET

END
```

This creates approximately a 1 ms delay using nested loops.

Example 3: Subroutine 2

```
;--------------------------------
; 1 SECOND DELAY SUBROUTINE
;--------------------------------

DELAY1S:

    MOV R5, #250        ; Outer loop count

REPEAT:

    ACALL DELAY1MS
    ACALL DELAY1MS
    ACALL DELAY1MS
    ACALL DELAY1MS      ; 4 × 1ms = 4ms

    DJNZ R5, REPEAT     ; Repeat 250 times

    RET
```

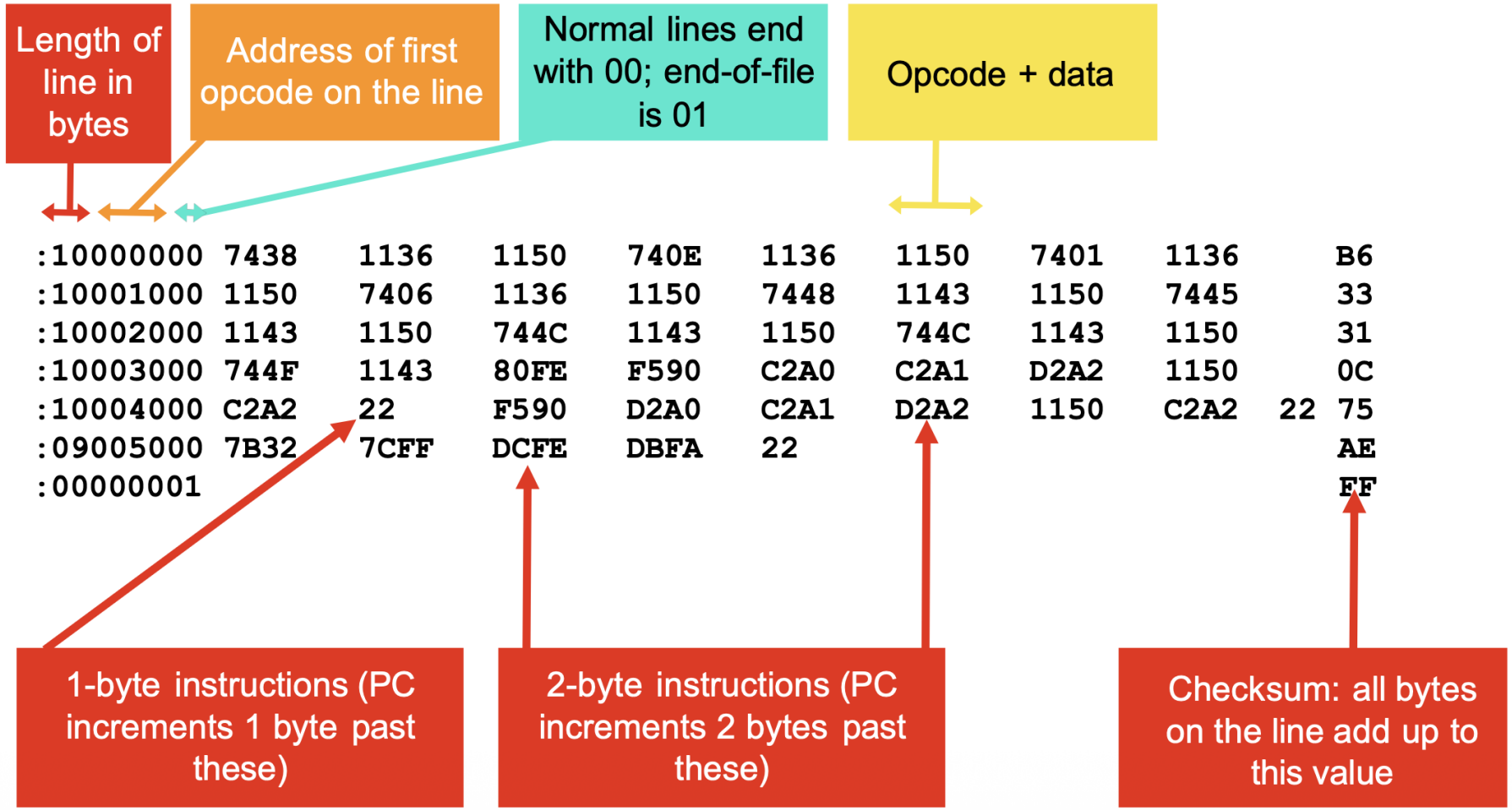
This saying 250 times

Understanding Hex Files

- Once written and carefully checked over, the assembly language program is assembled.
- We'll use the KEIL IDE to do this.
- The result is a Hex file (.hex), with opcodes and accompanying data represented as hex numbers.
- This hex file is in the Intel Hex format.
- More good info about this here: https://www.edsim51.com/intelHex.html
- If you are going to do a lot of Hex file editing, a dedicated hex editor is recommended https://mh-nexus.de/en/hxd/

Understanding Hex Files

Hexadecimal file has specific format and structure when viewed with special programming editor.



Programming Standard and Conventions

Turn in: a commented Hex file at start of your Lab 4.

- This needn't have many additional notes.
- 1 or 2 lines up at the top explaining the changes that you have made.
- A brief comment on each line explaining the line-by-line changes.
- See further handout provided in the lab for standard and conventions

– similar to those used in programming courses

8051 Assembly Language Instruction set

Basic 8051 assembly language instruction sets:

a. Memory addressing – interacting and working with data in the program. The highlight of the assembly language programming. Mostly with byte enable registers

b. Bit addressing – This is special type of memory addressing instruction sets with bit enable registers.

8051 Addressing Modes

IMMEDIATE ADDRESSING MODE

- The data is included in the instruction.
- MOV A,#48H
- The # shows that the data is 'immediate'
- In a sense, this data is hard-coded into the instruction. Fast but less flexible.

REGISTER ADDRESSING MODE

- The data operand is in a specified register.
- Only some registers may be used: R0 through R7 of each of the 8051's banks.
- MOV A,R7
- Contents of R7 are copied to ACC.

DIRECT ADDRESSING MODE

- The address of a location in RAM is specified, and its contents are operated upon. Only works with internal RAM & SFR's
- MOV A,10H
- Contents of address are copied to ACC.

INDIRECT ADDRESSING MODE

- Slower: the contents of a location of the address stored in a register are fetched.
- MOV A,@R7
- The @ indicates an address
- The upper 128 bytes of RAM are accessible this way.

INDEXED ADDRESSING MODE

- Used to step through data (as in lookup tables).
- We won't be exploring this in depth (and you won't be tested on it!), but see details about the MOVC instruction in C8051F02xC3.pdf

8051 Addressing Modes

Unlike PC programming that typically frees the developers from accessing memory, assembly language uses memory addressing very closely in its programming.

Eight addressing modes are available with the 8051.

The different addressing modes determine how the operand byte is selected.

Addressing Modes	Instruction
Register	MOV A, B
Direct	MOV 30H,A
Indirect	ADD A,@R0
Immediate Constant	ADD A,#80H
Relative*	SJMP AHEAD
Absolute*	AJMP BACK
Long*	LJMP FAR_AHEAD
Indexed	MOVC A,@A+PC

8051 Addressing Modes

- The register addressing instruction involves information transfer between registers.
- Example:

MOV R0, A

- The instruction transfers the accumulator content into the R0 register.
- The register bank (Bank 0, 1, 2 or 3) must be specified prior to this instruction.

Lower 128 – Register Banks and RAM

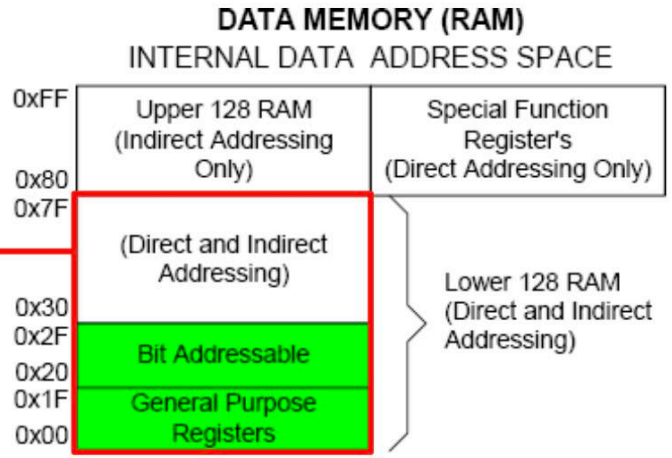
Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	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
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

Bit-addressable Area (16 bytes)

General Purpose RAM (80 bytes)

Register Banks (8 bytes per bank; 4 banks)

- Bits 3 and 4 of PSW (RS1 and RS2) determine which register bank (Bank 0, 1, 2 or 3) is used.



- Addresses of stack programming overlap with the default register bank (Bank 0).

Direct Addressing

This mode allows you to specify the operand by giving its actual memory address (typically specified in hexadecimal format) or by giving its abbreviated name (e.g. P3).

Note: Abbreviated SFR names are often defined in a header file

- Example:

```
MOV A, P3      ;Transfer the contents of
               ;Port 3 to the accumulator
MOV A, 020H ;Transfer the contents of RAM
               ;location 20H to the accumulator
```

Indirect Addressing

This mode uses a pointer to hold the effective address of the operand.

- Only registers R0, R1 and DPTR can be used as the pointer registers.
- The R0 and R1 registers can hold an 8-bit address, whereas DPTR can hold a 16-bit address.
- Examples:

```
MOV @R0,A       ;Store the content of accumulator into
                ;the memory location pointed to by
                ;register R0. R0 could have an
                ;8-bit address, such as 60H.
MOVX A,@DPTR    ;Transfer the contents from
                ;the memory location pointed to by DPTR
                ;into the accumulator. DPTR could have a
                ;16-bit address, such as 1234H.
```

Immediate Constant Addressing

- This mode of addressing uses either an 8- or 16-bit constant value as the source operand.
- This constant is specified in the instruction, rather than in a register or a memory location.
- The destination register should hold the same data size which is specified by the source operand.
- *Examples*:

```
ADD A,#030H          ;Add 8-bit value of 30H to
                     ;the accumulator register
                     ;(which is an 8-bit register).
MOV DPTR,#0FE00H     ;Move 16-bit data constant
                     ;FE00H into the 16-bit Data
                     ;Pointer Register.
```

Relative Addressing

This mode of addressing is used with some type of jump instructions, like SJMP (short jump) and conditional jumps like JNZ.

- These instructions transfer control from one part of a program to another.
- The destination address must be within -128 and +127 bytes from the current instruction address because an 8-bit offset is used (28 = 256).
- Example:

```
GoBack:    DEC A ;Decrement A
           JNZ GoBack ;If A is not zero, loop back
```

Absolute Addressing

Two instructions associated with this mode of addressing are ACALL and AJMP instructions.

- These are 2-byte instructions where the 11-bit absolute address is specified as the operand.
- The upper 5 bits of the 16-bit PC address are not modified. The lower 11 bits are loaded from this instruction. So, the branch address must be within the current 2K byte page of program memory (211 = 2048).
- Example:

```
ACALL PORT_INIT      ;PORT_INIT should be
                  ;located within 2k bytes.
PORT_INIT:
MOV P0, #0FH      ;PORT_INIT subroutine
```

Long Addressing

This mode of addressing is used with the LCALL and LJMP instructions.

- It is a 3-byte instruction, and the last 2 bytes specify a 16-bit destination location where the program branches.
- It allows use of the full 64 K code space.
- The program will always branch to the same location no matter where the program was previously.
- Example:

```
LCALL TIMER_INIT ;TIMER_INIT address (16-bitslong)
                 ;is specified as the operand;
                 ; In C, this will be a function call:
                 ;Timer_Init().
;TIMER_INIT subroutine
TIMER_INIT: ORL TMOD,#01H
```

Indexed Addressing

The Indexed addressing is useful when there is a need to retrieve data from a look-up table

- A 16-bit register (data pointer) holds the base address, and the accumulator holds an 8-bit displacement or index value
- The sum of these two registers forms the effective address for a JMP or MOVC instruction
- Example:

```
MOV A,#08H              ;Offset from table start
MOV DPTR,#01F00H        ;Table start Address
MOVC A,@A+DPTR          ;Gets target value from the table
                        ;start address + offset and puts it in A.
```

After the execution of the above instructions, the program will branch to address 1F08H (1F00H+08H) and transfer into the accumulator the data byte retrieved from that location (from the look-up table)

Example 4: Memory Addressing

The following assembly language program shows the most common memory addressing programming in 8051 used in the course

```
;Register addressing
MOV R0, A

;Direct addressing
MOV A, P3           ;Transfer the contents of
                    ;Port 3 to the accumulator
MOV A, 020H         ;Transfer the contents of RAM
                    ;location 20H to the accumulator

;Indirect addressing
MOV @R0,A           ;Store the content of accumulator into
                    ;the memory location pointed to by
                    ;register R0. R0 could have an
                    ;8-bit address, such as 60H.
```

Example 4: Memory Addressing

```
;Immediate constant addressing
ADD A,#030H          ;Add 8-bit value of 30H to
                     ;the accumulator register
                     ;(which is an 8-bit register).

;Relative addressing
ADD A,#030H          ;Add 8-bit value of 30H to
                     ;the accumulator register
                     ;(which is an 8-bit register).
MOV DPTR,#0FE00H     ;Move 16-bit data constant
                     ;FE00H into the 16-bit Data
                     ;Pointer Register.

;Absolute addressing
ACALL PORT_INIT      ;PORT_INIT should be
                     ;located within 2k bytes.
PORT_INIT:
```

Example 4: Memory Addressing

```
Long addressing
LCALL TIMER_INIT      ;TIMER_INIT address (16-bits
                      ;long) is specified as the
                      ;operand;
TIMER_INIT:


;Index addressing
MOV A,#08H            ;Offset from table start
MOV DPTR,#01F00H      ;Table start Address
MOVC A,@A+DPTR        ;Gets target value from the
;table start address + offset ;and puts it in A.
```

MemoryAddressing.asm

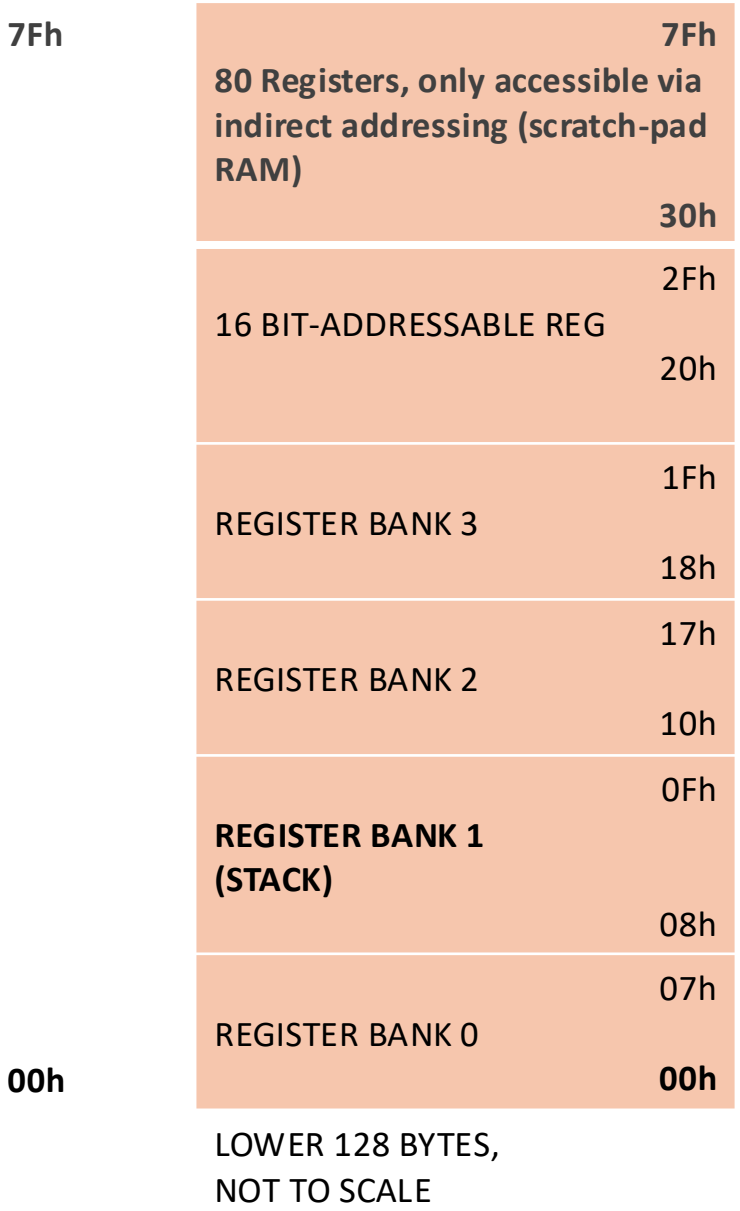
Note: Index addressing might not be covered in the course

Bit Addressing

BYTE ADDRESS	BIT ADDRESS								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	D1	D0	PSW
B8	BF	BE	BD	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	AE	AD	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit-addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	
8D	Not bit-addressable								TH1
8C	Not bit-addressable								TH0
8B	Not bit-addressable								TL1
8A	Not bit-addressable								TL0
89	Not bit-addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit-addressable								PCON
83	Not bit-addressable								DPH
82	Not bit-addressable								DPL
81	Not bit-addressable								SP
80	87	86	85	84	83	82	81	80	P0

Byte Address	Bit Address							
7F	General Purpose RAM							
30								
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
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0-R7							
00								

Bit Addressing



The 8051 has 16 bit-addressable registers.• Bit addressable: each bit within each register maybe independently manipulated.

[_ _ _ _ _ _ _ _] < each of those bits may be toggled, allowing fine-grained control.

Region from 0x20 to 0x2F is bit addressable.
• Some of the SFR's (special function registers) are also bit-addressable.
• This allows us to edit settings (e.g., PSW) bit-by- bit using the SETB instruction and the CLR instruction.

In a byte-addressable register, only the whole byte can be manipulated.
• If you wish to manipulate a byte-addressable register bit-by-bit, you typically must move it to a bit-addressable register, edit it, and then move it back.
• R0-R7 in each of the register banks are byte-addressable, as is the region from 0x30 to 0x7F

Bit Addressing

Bit addressing with symbols is by dot notation.

```
CLR 97H      ;Clear bit 7 of port 1 in SFR (bit addressing)
CLR P1.7     ;Same as the above (symbol with dot notation)
```

- The assembler performs translation and the machine code contains the appropriate operand value.
- Generally, any mnemonic listed in the datasheet can be used as a symbol in an assembly language program.

Bit Addressing

```
CLR 97H      ;Clear bit 7 of port 1 in SFR (bit addressing)
CLR P1.7     ;Same as the above (symbol with dot notation)
```

Port 1 is an **8-bit Special Function Register (SFR)**.
Its address is:
P1 = 90H

Bit	Address
P1.0	90H
P1.1	91H
P1.2	92H
P1.3	93H
P1.4	94H
P1.5	95H
P1.6	96H
P1.7	97H

Example 5: Bit Addressing

```
MOV A,#11000011B     ;A=11000011B
RL A                 ;A=10000110B
RR A                 ;A=01000011B

MOV A,#11000011B     ;A=10000000B
RLC A                ;A=00000000B and Carry=1
RLC A                ;A=00000001B and Carry=0
RRC A                ;A=00000000B and Carry=1
RRC A                ;A=10000000B

MOV P1, #10101111B   ;P1=10101111B
SETB P1.6            ;P1=11101111B
CLR P1.2`            ;P1=11101011B
CPL P1.O             ;P1=10101010B

MOV A, 11111010B     ;A=11111010B
SWAP A               ;A=10101111B
```

Example 5: Bit Addressing

```
                    A=10000110B
RR A                ;A=01000011B
```

Bit Position	7	6	5	4	3	2	1	0
Value	1	0	0	0	0	1	1	0

Visual Movement

Starting:

[1 0 0 0 0 1 1 0]

↑ ↑

bit7 bit0

Move everything right:

[? 1 0 0 0 0 1 1] 0

↑ ↑

bit7 bit0

Now place old bit0 (1) into bit7:

[0 1 0 0 0 0 1 1]

Final result:

01000011

Example 5: Bit Addressing

```
                    ;A=11111010B
SWAP A              ;A=10101111B
```

SWAP A

Swap upper 4 bits and lower 4 bits.

Split into nibbles:

1111 | 1010
Swap:
1010 | 1111

Example 5: Bit Addressing

```
MOV P2,#10101111B    ;P2=10101111B
ANL C,P2.0           ;As P2.0 is 0, so Carry=0
ORL C, P2. 7         ; As P2.7 is 1, so Carry=1
ANL P2.6,C           ;C=1 and P2.6=1,so bit P2.6=1, P2=11101111B
ANL P1,#00000001B    ;P1=00000001B
ORL P1,#00000010B    ;P1=#00000011B

MOV A, #80H          ;A=10000000H
RLC A                ;A=00000000H and Carry=1
JC LOOP              ;Program will not jump to LOOP if C=1
JNC LOOP             ;Program will not jump to LOOP if C=0
JB P0.2,LOOP         ;If P0.2=1 program jump to LOOP
JNB P1.4,LOOP        ;If P1.4=0 program jump to lOOP
JBC P0.2,LOOP        ;If P0.2=1 program jump to LOOP, P0.2=0
LOOP:
```

BitAddressing.asm

Example 5: Bit Addressing

```
ANL C,P2.0              ;As P2.0 is 0, so Carry=0
```

ANL (AND Not)
performs a logical AND operation between the Carry flag (C) and the bit P2.0 of P2.

The value of P2 is [10101111]B
P2.0 is 1
Carry flag (C) = 1. – *Check slide 43*

1 AND 1 = 1. Therefore, the Carry flag (C) is set to 1.

Result:
The Carry flag (C) is set to 1.

Example 5: Bit Addressing

```
MOV P2,#10101111B    ;P2=10101111B
ANL C,P2.0           ;As P2.0 is 0, so Carry=0
ORL C, P2. 7         ; As P2.7 is 1, so Carry=1
ANL P2.6,C           ;C=1 and P2.6=1,so bit P2.6=1, P2=11101111B
ANL P1,#00000001B    ;P1=00000001B
ORL P1,#00000010B    ;P1=#00000011B

MOV A, #80H          ;A=10000000H
RLC A                ;A=00000000H and Carry=1
JC LOOP              ;Program will not jump to LOOP if C=1
JNC LOOP             ;Program will not jump to LOOP if C=0
JB P0.2,LOOP         ;If P0.2=1 program jump to LOOP
JNB P1.4,LOOP        ;If P1.4=0 program jump to lOOP
JBC P0.2,LOOP        ;If P0.2=1 program jump to LOOP, P0.2=0
LOOP:
```

BitAddressing.asm

Example 5: Bit Addressing

Instruction	Action
RL A	Rotate left
RR A	Rotate right
RLC A	Rotate left through carry
RRC A	Rotate right through carry
SETB bit	Set bit = 1
CLR bit	Clear bit = 0
CPL bit	Toggle bit
SWAP A	Swap upper/lower nibble

Example 5: Bit Addressing

Instructions	Action
MOV	Load binary value into
ANL C	AND Carry
ORL C	OR Carry
ANL P2.6,C	AND bit P2.6 with Carry
ANL	AND
ORL P1,#00000010B	OR Port 1
MOV A,#80H	Load accumulator with 80H
RLC A	Rotate accumulator left through Carry
JC LOOP	Jump if Carry = 1
JNC LOOP	Jump if Carry = 0
JB P0.2,LOOP	Jump if bit is set
JNB P1.4,LOOP	Jump if bit is not set
JBC P0.2,LOOP	Jump if bit is set and clear it
LOOP:	Label definition (target location)

Review

A microcontroller system uses: Port 1 (P1) and Port 2 (P2) for input and output operations.

```
MOV P2,#10101010B
ANL P2,#11110000B
MOV A,P2
SWAP A
MOV P1,A
END
```

Part A: Conceptual Questions

1. What is the initial binary value loaded into Port 2?
2. What does the instruction

   ANL P2,#11110000B

   do?
3. What is the purpose of the SWAP A instruction?
4. Which register acts as the accumulator in this program?
5. What is the final binary value stored in Port 1 (P1)?

Part B: Step-by-Step Execution

6. Write the binary value loaded into P2.
7. Perform the AND operation below and show the result:

   10101010

   11110000
8. What value is transferred into accumulator A after:.
9. After SWAP A, what is the new value of A?
10. What is the final binary value written to P1?

Part C: Application Question

11. Modify the program so that:

•Lower nibble of P2 is sent to upper nibble of P1