University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers CPEN312

Lecture 13a: Introduction to 8051 Assembly I

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 8, 2023

Objectives

- Get acquainted to the 8051 instruction set.
- Understand how an assembler compiler is used and the files it generates.
- Assemble mnemonics by hand.
- Disassemble machine code by hand.
- Calculate machine code execution time.
- Write simple assembly programs.
- Load machine code by hand.

8051 Instruction Set

1. Arithmetic Operations
2. Logical Operations
3. Data Transfer
4. Boolean Variable Manipulation
5. Program Branching and Machine Control

8051 Instruction Set

- The instructions are all described in chapter 2 of "MCS-51 Microcontroller Family User's Guide".
- The number of cycles per instruction for the CV-8052 are listed in the one page handout distributed:
  - Side 1: Instructions sorted by opcode number
  - Side 2: Instructions sorted by mnemonic name.

Assembly Language Instruction

- **An Assembly language instruction consists of four fields:**

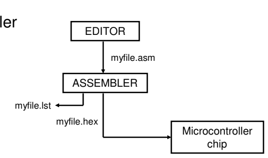
[label :] mnemonic [operands] [;comment]

- **Examples:**

```
L1:  MOV A, #'*' ; Load Acc with ASCII for '*'
     NOP
     ; We can just have comments!
L3: ; We can also place labels anywhere
     DJNZ R0, L3
```


Writing and compiling an assembly language program

Using A51 assembler by yours truly!



Hand writing, compiling, and loading an assembly program

By hand: Good for educational purposes, but very, very impractical for real applications!

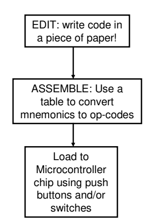


Table of mnemonics is available as an Excel spreadsheet in the course web page: *'CV-8052_opcodes.xls'*

Exercise: Assembling Machine Code by Hand

Assemble by hand the following piece of machine code:

```
E9   MOV A, R1     ; Low 8-bits first number
2B   ADD A, R3     ; Add to low 8-bits second number
FD   MOV R5, A     ; Save result-low to R5
E8   MOV A, R0     ; High 8-bits first number
3A   ADDC A, R2    ; Add  with carry to high 8-bits second number
FC   MOV R4, A     ; Save result-high to R4
```

Answer: E9 2B FD E8 3A FC

This type of question often appears in exams

Disassembling

- In microcomputer jargon, disassembling is the process of getting the source code from the machine code.
- Many debuggers disassemble machine code to trace program execution.

Exercise: Disassembling by hand

- What is the value of R2 after executing this piece of 8051 machine code?
  C3 74 88 94 10 FA

C3	CLR C	; Set carry to zero
74 88	MOV A, #88H	; Load Acc. With 88H
94 10	SUBB A, #10H	; 88H-10H=78H
FA	MOV R2, A	; R2=78H

Answer!

This type of question often appears in exams

Number representation

- Decimal (default): 2957 or 2957D
- Binary: 101110001101B
- Octal: 5615o or 5615Q
- Hexadecimal: 0B8DH, 0b8dh, 0x0B8D, 0x0b8d

The maximum number that can be input in any radix is 65535 which corresponds to 16 bits. If we have a 'equ' formula, the result should fit in 16 bits.

Machine cycles for the 8051 Instructions

- The microcontroller takes a certain number of clock cycles to execute an instruction.
- These clock cycles are referred to as *machine cycles*.
- In the 8051 family, the length of the machine cycle depends on the frequency of the crystal oscillator.
- The frequency of the crystal connected to the 8051 family can range from 32kHz to over 1000 MHz.
- In the original 8051, one machine cycle lasts 12 oscillator periods. Therefore, to calculate the machine cycle for the original 8051, we take 1/12 of the crystal frequency, then take its inverse.
- For newer 8051 derivatives, one machine cycle can take 12, 6, 4, 2, or 1 oscillator period(s).
- Warning: The same instruction can take a different number of machine cycles in 8051's fabricated by different manufacturers:

Machine cycles for the 8051 Instructions

Instruction	Intel 8051	CPEN312 CV-8052	Atmel AT89LP52	Silicon Labs C8051F38x
MOV R3, #value	1	2	2	2
DEC Rx	1	1	1	1
DJNZ	2	2/3	2	2/4
LJMP	2	3	3	5
SJMP	2	2	2	4
NOP	1	1	1	1
MUL AB	4	1	2	4

Machine cycles for the CV_8052

- Available in Excel format in the course web page.
- The CV_8052 takes one clock per cycle!
- The clock of the CV_8052 is 33.333333MHz. Therefore, each cycle is 1/33.333333MHz=30ns.
- The opcode table for ALL 8051 compatible microcontrollers is the same. The number of bytes per instruction is the same. The number of cycles per instruction MAY NOT BE the same.

Example 1: Instruction Timing for the original 8051 from Intel

- How long will it take this program to run in a standard 8051 (12 clocks per cycle) with a 12MHz crystal? Use the cycles per instruction of Appendix G in the textbook.

CODE: C3 74 88 94 10 FA

C3	CLR C	; Set carry to zero (1 cycle)
74 88	MOV A, #88H	; Load Acc. With 88H (1 cycle)
94 10	SUBB A, #10H	; 88H-10H=78H (1 cycle)
FA	MOV R2, A	; R2=78H (1 cycle)

1 cycle=12/12MHz=1µs, therefore 4 cycles take 4µs

Example 2: Instruction Timing for the CV_8052

- How long will it take this program to run in CV_8052 (1 clock per cycle) with a 33.3333 MHz clock?

CODE: C3 74 88 94 10 FA

C3	CLR C	; Set carry to zero (1 cycle)
74 88	MOV A, #88H	; Load Acc. With 88H (2 cycles)
94 10	SUBB A, #10H	; 88H-10H=78H (2 cycles)
FA	MOV R2, A	; R2=78H (1 cycle)

1 cycle=1/33.33MHz=30ns, therefore 6 cycles take 0.18µs

This type of question often appears in exams

8051 Registers

- The 8051 is LOADED with registers, all of them (but one) are 8-bit:
  – 32 general purpose registers arranged in 4 banks of 8 register each: R0 to R7. R0 and R1 can be used as “index” registers for indirect access.
  – N (depends on the derivative, but usually 27 or more) Special Function Registers or SFR.
- All the hardware resources of the 8051 are accessed through SFRs. More on to this later.

8051 Registers

- Just a few registers are related to ALU operations:
  – Accumulator (Acc).
  – Register B.
  – Data Pointer DPTR (two eight bit registers put together: DPH and DPL)
  – Stack Pointer SP.
  – Program Counter PC.
  – Program Status Word PSW.

Registers R0 to R7

	Bank 0		Bank 1		Bank 2		Bank 3
7	R7	F	R7	17	R7	1F	R7
6	R6	E	R6	16	R6	1E	R6
5	R5	D	R5	15	R5	1D	R5
4	R4	C	R4	14	R4	1C	R4
3	R3	B	R3	13	R3	1B	R3
2	R2	A	R2	12	R2	1A	R2
1	R1	9	R1	11	R1	19	R1
0	R0	8	R0	10	R0	18	R0

Accumulator

- It is the most versatile of all the registers. All the ALU operations place the result in the accumulator.
- Curiously, the accumulator is also a Special Function Register (SFR) in the 8051.
- Bit Addressable. We can use the setb, clr, jb, etc. with the accumulator bits: setb acc.3
- Many opcodes accept only the accumulator as operand.
- Example usage:
  – MOV a, #20H ; Load accumulator with 20H
  – INC a ; Add one to accumulator
  – SWAP A ; Swap bits 0 to 3 with bits 4 to 7

Register B

- Used together with the Accumulator to perform 8-bit multiplication/division operations.
- Also bit addressable.
- Example usage:
  – MOV A, #140
  – MOV B, #150
  – MUL AB ; After this inst. A=08H, B=52H

Data Pointer DPTR

- Formed by two 8-bit SFRs: DPH and DPL.
- Together with some especial opcodes, it is used to access CODE and XRAM memory.
- This is the only 16-bit register in the 8051.
- There is a dedicated opcode to increment it! Unfortunately there is no opcode to decrement it!
- Example:
  – MOV DPTR, #0AAAAH
  – INC DPTR ; DPL=0ABH, DPH=0AAH

How to decrement the DPTR

```
    mov a, dpl
    jnz skip_dph
    dec dph
skip_dph:
    dec dpl
```

Exercise: modify the code above so that the value of the accumulator remains unchanged.

Stack Pointer

- Need to be set for the ‘CALL’/’RET’ instructions to work properly.
- The stack will be cover with more detail in future lectures.
- For now, at the beginning of your code place the following instruction:
  – MOV SP, #7FH ; Set sp to beginning of IDATA!

Program Counter

- We can not access this 16-bit register directly. Increments by one, two, or three bytes (one for most instructions), depending on the opcode length.
- We can only use the “jump”, “call”, or “return” instructions to change the program counter.
- Example:
  – LJMP 34AAH ; Set PC to 34AAH

Program Status Word

- It is where the ALU are stored:

CY	AC	FO	RS1	RS0	OV	--	P

CY	Carry flag
AC	Auxiliary Carry flag (For BCD Operations)
FO	Flag 0 (Available to the user for General Purpose)
RS1, RS0	Register bank select bits.
OV	Overflow flag
P	Parity flag

Example: JC 8 ; If the carry is set jump 8 bytes ahead

Instructions that modify the ‘Flag’ bits in the PSW:

Instruction	CY	OV	AC
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RRC	X		
RLC	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		

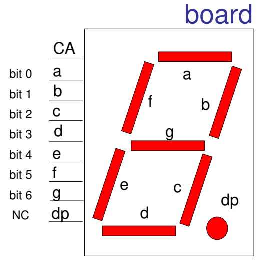
Example 3

- Write an assembly program that writes ‘JESUS’ to the 7-segment displays HEX4 to HEX0. Compile and load the program manually. The Special Function Register (SFR) addresses of HEX4 to HEX0 are:

HEX0: 91H
HEX1: 92H
HEX2: 93H
HEX3: 94H
HEX4: 8EH

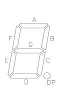
The segments of each display are mapped as : -gfedcba, ‘a’ is the least significant bit.

7-Segment Displays in DE0-CV board



Disp.	SFR
HEX0	91H
HEX1	92H
HEX2	93H
HEX3	94H
HEX4	8EH
HEX5	8FH

Segments turn on with zero.



7-segment constants

- The bit pattern for each letter is:

Letter	g	f	e	d	c	b	a	Hex
‘J’	1	1	0	0	0	0	1	61H
‘E’	0	0	0	0	1	1	0	06H
‘S’	0	0	1	0	0	1	0	12H
‘U’	1	0	0	0	0	0	1	41H
‘S’	0	0	1	0	0	1	0	12H

Exercise: Obtain the bit patterns for numbers 0 to 9

The assembly code

Address	Machine Code	Instruction
0000	75 8E 61	mov 8EH, #61H
0003	75 94 06	mov 94H, #06H
0006	75 93 12	mov 93H, #12H
0009	75 92 41	mov 92H, #41H
000C	75 91 12	mov 91H, #12H
000F	80 FE	sjmp $

Loading the Code by Hand

- First load the POF file CV_8052.POF to the Altera DE0-CV board (you need to do this only once):
  – Set SW10 to the ‘PROG’ position
  – Turn the board on. Connect to computer and load Quartus.
  – Click the ‘programmer’ icon in Quartus and send the above POF file to the DE0-CV board.
  – Set SW10 to the ‘RUN’ position.
- Second: We need to activate ‘manual load’ in the CV_8052 soft processor by pressing and releasing FPGA_RESET while KEY2 is pressed.

Manual load in the CV_8052

- HEX5 to HEX2 show the memory address while HEX1 and HEX0 show the current data at that memory address.
- SW8 selects address LOW entry.
- SW9 selects address HIGH entry.
- SW0 to SW7 value to enter.
- KEY3 increments the address by one.
- KEY2 decrements the address by one.
- KEY1 enters data.
- KEY0 and KEY1 at the same time stores changes to flash.

Example 4 (time permitting)

- Write a program that turns LEDR0 on/off every 0.5s. Compile and load the program manually. The bit address of LEDR0 is E8H.

Big delay: loops inside loops

```
main_loop:
    mov R2, #90  ; 90 is 5AH
L3: mov R1, #250 ; 250 is FAH
L2: mov R0, #250
L1: djnz R0, L1  ; 3 machine cycles-> 3*30ns*250=22.5us
    djnz R1, L2  ; 22.5us*250=5.625ms
    djnz R2, L3  ; 5.625ms*90=0.506s (approximately)
    cpl 0E8H     ; Bit address of LEDR0 is E8H
    sjmp main_loop
```

One possible solution! The same result can be achieved using many other valid programs.

Program compiled by hand

```
                 main_loop:
0000  7A 5A          mov R2, #90
0002  79 FA      L3: mov R1, #250
0004  78 FA      L2: mov R0, #250
0006  D8 FE      L1: djnz R0, L1
0008  D9 FA          djnz R1, L2
000A  DA F6          djnz R2, L3
000C  B2 E8          cpl 0E8H
000E  80 F0          sjmp main_loop
```


Exercises

- The machine code below runs in a CV-8052 processor that takes 1 clock per cycle with a 33.3333MHz clock. Find how long it takes this code to execute. (Warning: there is a loop you have to consider!)
  7E 08 0E 00 8E 80 00 00 DE FA
- Assemble by hand (both op-codes and operands) the program below.

```
MOV R7, #75H
MOV A, R7
ANL A, #0FH
ORL A, #30H
MOV R0, A
MOV A, R7
SWAP A
ANL A, #0FH
ORL A, #30H
MOV R1, A
```

- Modify the programs of examples 3 and 4 so that they turn off all unused LEDs by writing zero to them. The SFR addresses for the LEDS are: LEDR0-7=E8H, LEDR8-9=95H.

University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers CPEN312

Lecture 13b: Introduction to 8051 Assembly II

Dr. Jesús Calviño-Fraga
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 9, 2017

Objectives

- Get familiar with 8051's assembler programs.
- Write compilable assembly programs.
- Compile assembly programs using a computer.
- Load and run programs written in assembly language using a computer.

Assembling by hand…

- Assembling by hand is sometimes 'fun' but it is also:
  - Prone to error.
  - Slow.
  - Tedious.
  - Hard to add simple changes.
- We use a computer to 'assembly' our code from the assembly code mnemonics. We need:
  - A computer
  - The assembler compiler!

The Assembler

- In this course we will be using A51.
  - I wrote this assembler!
  - Syntax is compatible with original Intel ASM51. The syntax is mostly the same as the assembler described in the text book.
- We also need a text editor. Any text editor will do, for example Notepad works fine. On the other had we can use an editor with syntax highlighting as in Crosside:

http://www.ece.ubc.ca/~jesusc/crosside_setup.exe

Assembler Overview

- Symbols
- Labels
- Assembler Controls
- Assembler Directives
- ASCII Literals
- Comments
- Macros

Symbols

- The legal characters: both upper and lower case (A..Z,a..z) letters, decimal numbers (0..9) and these two special characters: question mark (?) and underscore (_).
- Symbols can not start with a number.
- The assembler converts all symbols to uppercase. For instance these two are the same symbol:
  - My_symbol
  - MY_SYMBOL
- Mnemonics, register names, assembler controls and assembler directives are reserved words and can not be used as symbols.

Labels

- Labels are symbols also: same rules apply.
- Must appear BEFORE mnemonics, the storage directives (DB and DW), or data reservation directives (DS and DBIT).
- A label is defined by a symbol name and ':'
- Example:

```
L1: DJNZ R0, L1
```


Assembler Controls

- Assembler controls are used to tell the program where to gets its input source files (include files), where it puts the object file, and how it formats the listing file.
- Example:

```
$MOD52
$TITLE(CPEN312 – Example for Lecture 14)
$LIST
$PAGEWIDTH(75)
```

Assembler Controls

$DATE(date) Places date in page header
$INCLUDE(file) Inserts file in source program
$TITLE(string) Places string in page header
$NOLIST Stops outputting the listing
$MOD52 Uses 8052 predefined symbols
$MOD51 Uses 8051 predefined symbols
$MODDE0CV Uses CV-8052 predefined symbols
$NOMOD No predefined symbols used
$NOOBJECT No object file is generated
$NOPAGING Print listing w/o page breaks
$PAGEWIDTH(n) No. of columns on a listing page
$NOPRINT Listing will not be output
$NOSYMBOLS Symbol table will not be output
$EJECT Places a form feed in listing
$LIST Allows listing to be output
$OBJECT(file) Places object output in file
$PAGING Break output listing into pages
$PAGELENGTH(n) No. of lines on a listing page
$PRINT(file) Places listing output in file
$SYMBOLS Append symbol table to listing

Actually you can use any register definition file provided with the assembler, or even write your own!

Assembler Directives

- Assembler directives are used to define symbols, reserve memory space, store values in program memory and switch between different memory spaces.
- Examples of directives:

  TEN EQU 10

  RESET CODE 0

  ORG 4096

Assembler Directives

- Symbol Definition Directives: EQU, SET, BIT, CODE, DATA, IDATA, XDATA
- Segment Selection Directives: CSEG, BSEG, DSEG, ISEG, XSEG
- Memory Reservation and Storage Directives: DS, DBIT, DB, DW
- Miscellaneous Directives: ORG, USING, END

EQU directive

- Very useful to keep your code looking clean and professional:

```
CLK   EQU 33333333
FREQ EQU 1000
BAUD equ 115200
TIMER_0_RELOAD EQU (0x10000-(CLK/(12*FREQ)))
TIMER_2_RELOAD equ (0x10000-(CLK/(32*BAUD)))
```


Segment Selection Directives

- There are five Segment Selection directives: CSEG, BSEG, DSEG, ISEG, XSEG, one for each of the five memory spaces in the 8051 architecture.
- Mostly used to allocate variables and constants in memory. Normally combined with DS, DBIT, DB, DW.
- Examples:

  DSEG at 30H ; Variables after the register banks and bits

  BSEG at 20H ;

  CSEG at 8000H

Memory Reservation and Storage Directives

- DS Directive: Used to reserve memory for variables. Works when ISEG, DSEG or XSEG are the active segments.
- Examples:

```
          DSEG at 30H ; Select data segment
          DS       32 ; Label is optional!
BUFF1:    DS       16
AVERAGE:  DS       8
COUNT:    DS       1
```


Memory Reservation and Storage Directives

- DBIT Directive: is used to reserve bits within the BIT segment.
- Examples:

```
          BSEG      ; Select bit segment
LEDON:    DBIT      1
LEDOFF:   DBIT      8
ERROR:    DBIT      1
```


Memory Reservation and Storage Directives

- DB Directive: is used to store byte constants in the Program Memory Space. It can only be used when CSEG is the active segment.
- Examples:

```
CSEG at 8000H
COPYRGHT_MSG:
   DB          '(c) Copyright, 2017 Jesus Calvino-Fraga', 0
RUNTIME_CONSTANTS:
   DB        127, 13, 54, 0, 99     ; Table of constants
   DB        17, 32, 239, 163, 49   ; Second line of const.
MIXED:
   DB        2*8, 'MPG' , 2*16, 'abc' ; Can mix literals & no.
```

Memory Reservation and Storage Directives

- DW Directive: is used to store word constants in the Program Memory Space. Similar to DB, but it stores two bytes (16-bits) at once.
- Examples:

```
CSEG at 8000H
DW       'AB', 1000H
```

Lecture 13b: Introduction to 8051 Assembly II 17

Miscellaneous Directives

- ORG Directive is used to specify a value for the currently active segment's location counter.
- **Examples:**

```
CSEG
ORG 0
  LJMP MyCode
ORG 1BH ; Timer 1 ISR vector location
  LJMP 1803H
MyCode:
  MOV SP, #7FH
```

Lecture 13b: Introduction to 8051 Assembly II 18

Miscellaneous Directives

"The USING Directive is used to specify which of the four General Purpose Register banks is used in the code that follows the directive. It allows the use of the predefined register symbols AR0 thru AR7 instead of the register's direct addresses. It should be noted that the actual register bank switching must still be done in the code. This directive simplifies the direct addressing of a specified register bank."

Example:

```
USING 1
PUSH AR2 ; Pushes R2 of bank 1 into the stack
```

Lecture 13b: Introduction to 8051 Assembly II 19

Miscellaneous Directives

- END Directive is used to signal the end of the source program to the Cross assembler.
- Your code should have an END directive!
- Example:

```
END      ;This is the End
```

Lecture 13b: Introduction to 8051 Assembly II 20

ASCII literals

- ASCII characters can be used directly as an immediate operand, or they can used to define symbols or store ASCII bytes in Program Memory (DB directive).
- **Example:**
  - **MOV A,#'m' ; Load A with 06DH (ASCII m)**
  - **DB 'Hello there!' ; Stored in Program Memory**

Lecture 13b: Introduction to 8051 Assembly II 21

Comments

- Comments can be placed anywhere, but they must be the last field in any line.
- They are preceded by ';'
- **Example:**

```
      MOV R0, #100 ; Repeat 100 times
LOOP: DJNZ R0, LOOP
```

- **Beware of useless comments:**

```
      MOV R0, #100 ; Move 100 to R0
LOOP: DJNZ R0, LOOP ;Decrement and
                    ;jump if no zero
                    ;to LOOP
```

Lecture 13b: Introduction to 8051 Assembly II 22

Macros

- A macro is a name assigned to one or more assembly statements or directives. Macros are used to include the same sequence of instructions in several places.
- A macro is a segment of instructions that is enclosed between the directives MAC and ENDMAC. The format of a macro is as follows:

Lecture 13b: Introduction to 8051 Assembly II 23

Macros

```
average MAC ; computes the average of three byte arguments
    mov A, %0
    add A, %1
    add A, %2
    mov B, #3
    div AB ; divide B into A and A gets the average
ENDMAC ; terminate the macro definition
```

To invoke the above defined macro, enter the macro name and its parameters. The statement:

```
average (R1 ,R2, R3)
```

will enable the assembler to generate the following instructions, starting from the current location counter:

```
mov A, Rl
add A, R2
add A, R3
mov B, #3
div AB
```

Lecture 13b: Introduction to 8051 Assembly II 24

Number representation

- Decimal (default): 2957 or 2957D
- Binary: 101110001101B
- Octal: 5615o or 5615Q
- Hexadecimal: 0B8DH, 0b8dh, 0x0B8D, 0x0b8d

The maximum number that can be input in any radix is 65535 which corresponds to 16 bits.

Addressing Modes

1. Register Inherent
2. Direct
3. Immediate
4. Indirect
5. Indexed
6. Relative
7. Absolute
8. Long
9. Bit Inherent
10. Bit Direct

Addressing Modes: Register Inherent

- The opcode is already associated with the register for speed and efficiency
- Examples:
  - MOV R1, #10 ; opcode: 01111001B + 00001010B
  - INC A ; opcode: 00000100B
  - INC R3 ; opcode: 00001011B

Addressing Modes: Direct

- Used to access the first 128 bytes of internal ram (address 0 to 127) or the SFRs (address 128 to 255).
- Examples:
  - MOV 20H, A ; 11110101B + 00100000B
  - MOV 50H, 25H ; 10000101B + 00100101B + 01010000B
  - MOV 50H, #25H ; 01110101B + 01010000B + 00100101B
  - MOV 80H, A ; 80H>127 therefore is a SFR!
  - MOV P0, A ; SFR P0 is at address 80H

Addressing Modes: Immediate

- Used to initialize a register or memory.
- The number to load MUST be preceded with '#'.
- Example:

```
    MOV R0, #10
L1: MOV P1, #01H
    NOP ; Does nothing just wastes time!
    MOV P1, #00H
    DJNZ R0, L1 ; Decrement an jump if no zero
```


Addressing Modes: Indirect

- Can be used to access ALL the internal RAM of the microcontroller. It is the only means of accessing the internal RAM from address 128 to 255!
- Only means of accessing XRAM memory using the MOVX instruction.
- EXAMPLES:

```
MOV R0, #80H
MOV A, @R0 ; Copy content of RAM loc. 80H into the Accumulator
MOV A, 80H ; Copy SFR 80H (P0) into the accumulator (See the difference!)
MOV DPTR, #200H
MOVX A, @DPTR ; Copy content of XRAM loc. 200H into the Accumulator
MOV R6, A
INC DPTR
MOVX A, @DPTR ; Copy content of XRAM loc. 201H into the Accumulator
MOV R7, A
```


Addressing Modes: Indexed

- Uses a base register (DPTR) and a offset register (Accumulator)
- Very common in other processors such as the Pentium, but limited in the 8051.
- Examples:

```
JMP @a+DPTR
MOVC a, @a+DPTR
MOVC a, @a+PC
```


Addressing Modes: Relative

- Used to jump (conditionally or unconditionally) to other parts of your code.
- The offset is an 8-bit SIGNED number. We can jump in a range -128 to +127 bytes:

```
                  1    $mod52
0000              2    L1:
0000 00           3            nop
0001 80FD         4            sjmp L1
0003 8001         5            sjmp L2
0005 00           6            nop
0006 00           7    L2:     nop
                  8    end
```

Addressing Modes: Relative

- To jump conditionally use any bit variable available:
  - JZ L1
  - JNZ L2
  - JC L3
  - JNC L4
  - JB P1.3, L5

Addressing Modes: Absolute

- Jump or call using a combination of the 11 bits in the destination and 5 upper bits of the program counter.
- It allow to jump or call within the same 2K page you program counter is.
- Two byte instructions!
- No conditional jumping available.
- Only two instructions available:
  - ACALL myroutine
  - AJMP DONE

Labels

Addressing Modes: Long

- Jumps or calls to any destination in the 16-bit range of the microcontroller.
- Three byte instructions (takes one more byte that the absolute instructions)
- No conditional jumping available.
- Only two instructions available:
  - LCALL myroutine
  - LJMP DONE

Relative vs. Absolute vs. Long

```
                       1     $mod52
0000 8007              2              sjmp L1
0002 0109              3              ajmp L1
0004 020009            4              ljmp L1
0007 4000              5              jc L1
0009 00                6     L1:      nop
                       7     end
```

The encoding of the AJMP/ACALL instructions is a bit weird. For example for ACALL:

A_{10}	A_9	A_8	1	0	0	0	1	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0

Addressing Modes: Bit Inherent

- Works only with the carry flag.
- Uses these three instructions:
  - SETB C
  - CLR C
  - CPL C

Addressing Modes: Bit Direct

- Very few processors have this mode.
- The first 128 bits are mapped in internal ram from bytes 20H to 2FH.
- The second 128 bits are mapped to SFR if the SFR address is divisible by 8.
- Examples:
  - CLR P0.0 ; Clear bit 0 of port 0 (Addr. 80H)
  - SETB P0.1 ; Set bit 1 of port 0 (Addr. 81H)
  - SETB 00H ; This is bit 0 of byte 20 in internal RAM

EXAMPLE: Blinky

```
; Blinky.asm: blinks LEDR0 of the CV-8052 each second.
$MODDE0CV

org 0000H
        ljmp myprogram

;For a 33.333333MHz clock, one machine cycle takes 30ns
WaitHalfSec:
        mov R2, #90
L3:     mov R1, #250
L2:     mov R0, #250
L1:     djnz R0, L1 ; 3 machine cycles-> 3*30ns*250=22.5us
        djnz R1, L2 ; 22.5us*250=5.625ms
        djnz R2, L3 ; 5.625ms*90=0.506s (approximately)
        ret
myprogram:
        mov SP, #7FH ; Set the beginning of the stack (more on this later)
        mov LEDRA, #0 ; Turn off all unused LEDs (Too bright!)
        mov LEDRB, #0
M0:
        cpl LEDRA.0
        lcall WaitHalfSec
        sjmp M0
END
```


EXAMPLE: Blinky

- To run “Blinky” in the CV-8052 we need to complete the following steps:
  1. Setup the CV-8052 processor in the Altera DE0-CV board.
  2. Edit and Compile “Blinky.asm”.
  3. Load and run “Blinky.hex” into the CV-8052 processor.

Setup the CV-8052 processor

- Download and Install Quartus from the Altera/Intel website into your computer. Also, follow Altera's instructions on how to install the USB-Blaster driver. Of course you can skip this step if you have installed Quartus already! Execute these instructions only once:
  - Download and open the project CV-8052 from the course web page.
  - Connect the USB cable to the DE0-CV board and the computer. Toggle switch SW10 from "RUN" to "PROG". Turn the DE0-CV board on.
  - Click Tools->Programmer->Start. Wait for the program to finish.
  - Toggle SW10 back to "RUN".

Compile "Blinky.asm" and flash "Blinky.hex".

- Download and install Crosside. See slide 4.
- Start Crosside and create a new assembly file. Copy and paste the code from slide 39. Save as "Blinky.asm".
- Click Build->ASM51. Click the "Browse" button beside "Complete path of assembler" and select a51.exe from the "Call51\bin" folder where you installed Crosside. Then press "Ok".

Flash and run into the CV-8052

- While pressing button "KEY0", press and release button "FPGA_RESET". When "boot" shows up on the 7-segment display, release "KEY0".
- In Crosside, click "fLash"->"Quartus SignalTap II". Make sure the file "Blinky.hex" is selected. Also make sure that the "quartus_STP.exe" and "Load_script.tcl" fields point to valid files. Click the "Flash" button and wait until it finishes.
- Press and release "FPGA_RESET". The program starts running!

Exercises

- Explain the differences between these two assembly instructions:
  ```
  mov a, #10H
  mov a, 10H
  ```
- There are two ways to access the internal memory of the 8051/8052 microcontroller: directly and indirectly. Explain why the combination of these two instructions
  ```
  mov R0, #0A0H
  mov a, @R0
  ```
  and this supposedly equivalent instruction
  ```
  mov a, 0A0H
  ```
  Result with different values in the accumulator.

Exercises

- In the CV-8052 the seven segments displays HEX0 to HEX5 are mapped to Special Function Registers with the same name. For instance, to turn on all segments of display HEX5 in assembly language we use "mov HEX5, #00H". Write an assembly program that counts from '0' to '9' and displays the number in HEX0.
- Write an assembly program that displays the least significant digits of your UBC student number (6 decimal digits) into the seven segment displays of the CV-8052.
- Write a macro that decrements the data pointer.

University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers CPEN312

Lecture 13c: Writing 8051 Assembly

Dr. Jesús Calviño-Fraga
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 9, 2017

Step 1

```
$MODDE0CV ; Special Function Registers declaration for CV-8052

org 0000H ; After reset, the processor starts at location zero

Forever:
	cpl LEDRA.0 ; Turn LEDR0 on/off
	ljmp Forever; Repeat forever
END
```

Step 2

```
$MODDE0CV ; Special Function Registers declaration for CV-8052

org 0000H ; After reset, the processor starts at location zero

        mov LEDRA, #0 ; Turn off LEDs LEDR[0..7] Bit addressable
        mov LEDRB, #0 ; Turn off LEDs LEDR[8..9] Not bit addressable
Forever:
        cpl LEDRA.0 ; Turn LEDR0 on/off
        ljmp Forever     ; Repeat forever
END
```


Step 3

```
$MODDE0CV ; Special Function Registers declaration for CV-8052

org 0000H ; After reset, the processor starts at location zero

        mov LEDRA, #0 ; Turn off LEDs LEDR[0..7] Bit addressable
        mov LEDRB, #0 ; Turn off LEDs LEDR[8..9] Not bit addressable
Forever:
        cpl LEDRA.0 ; Turn LEDR0 on/off
        lcall Delay
        ljmp Forever     ; Repeat forever

Delay:
        mov R0, #250
L0:     djnz R0, L0 ; 3 machine cycles-> 3*30ns*250=22.5us
        ret
END
```


Step 4

```
$MODDE0CV ; Special Function Registers declaration for CV-8052

org 0000H ; After reset, the processor starts at location zero

        mov LEDRA, #0 ; Turn off LEDs LEDR[0..7] Bit addressable
        mov LEDRB, #0 ; Turn off LEDs LEDR[8..9] Not bit addressable
Forever:
        cpl LEDRA.0 ; Turn LEDR0 on/off
        lcall Delay
        ljmp Forever     ; Repeat forever

Delay:
        mov R2, #90
L2:     mov R1, #250
L1:     mov R0, #250
L0:     djnz R0, L0 ; 3 machine cycles-> 3*30ns*250=22.5us
        djnz R1, L1 ; 22.5us*250=5.625ms
        djnz R2, L2 ; 5.625ms*90=0.506s (approximately)
        ret
END
```


Step 5

```
$MODDE0CV ; Special Function Registers declaration for CV-8052

org 0000H ; After reset, the processor starts at location zero
        ljmp main
Delay:
        mov R2, #90
L2:     mov R1, #250
L1:     mov R0, #250
L0:     djnz R0, L0 ; 3 machine cycles-> 3*30ns*250=22.5us
        djnz R1, L1 ; 22.5us*250=5.625ms
        djnz R2, L2 ; 5.625ms*90=0.506s (approximately)
        ret

main:   mov sp, #0x7f ; Initialize stack pointer
        mov LEDRA, #0 ; Turn off LEDs LEDR[0..7] Bit addressable
        mov LEDRB, #0 ; Turn off LEDs LEDR[8..9] Not bit addressable
Forever:
        cpl LEDRA.0 ; Turn LEDR0 on/off
        lcall Delay
        ljmp Forever     ; Repeat forever
END
```


Step 6

```
main:
        mov SP, #0x7f
        mov LEDRA, #0 ; Bit addressable
        mov LEDRB, #0 ; Not bit addressable
Forever:
        mov HEX4, #0x61 ; Letter 'J' to HEX4
        lcall Delay
        mov HEX3, #0x06 ; Letter 'E' to HEX3
        lcall Delay
        mov HEX2, #0x12 ; Letter 'S' to HEX2
        lcall Delay
        mov HEX1, #0x41 ; Letter 'U' to HEX1
        lcall Delay
        mov HEX0, #0x12 ; Letter 'S' to HEX0
        lcall Delay
        ljmp Forever ; Repeat forever
END
```


Step 7

```
        mov HEX4, #0xff ; Clear HEX4
        mov HEX3, #0xff ; Clear HEX3
        mov HEX2, #0xff ; Clear HEX2
        mov HEX1, #0xff ; Clear HEX1
        mov HEX0, #0xff ; Clear HEX0
        lcall Delay
        lcall Delay
        ljmp Forever ; Repeat forever
END
```


Step 8

```
LETTER_J EQU #0x61
LETTER_E EQU #0x06
LETTER_S EQU #0x12
LETTER_U EQU #0x41
BLANK    EQU #0xff
.
.
.
mov HEX4, LETTER_J
lcall Delay
mov HEX3, LETTER_E
lcall Delay
mov HEX2, LETTER_S
lcall Delay
mov HEX1, LETTER_U
lcall Delay
mov HEX0, LETTER_S
lcall Delay
```


Step 9

```
        mov HEX5, BLANK
        mov HEX4, LETTER_J
        mov HEX3, LETTER_E
        mov HEX2, LETTER_S
        mov HEX1, LETTER_U
        mov HEX0, LETTER_S
Forever:
        lcall Delay
        mov R4, HEX5
        mov HEX5, HEX4
        mov HEX4, HEX3
        mov HEX3, HEX2
        mov HEX2, HEX1
        mov HEX1, HEX0
        mov HEX0, R4
        ljmp Forever ; Repeat forever
```

Step 10

```
Forever:
        lcall Delay
        jb SWA.0, Scroll_Right
        mov R4, HEX5
        mov HEX5, HEX4
        mov HEX4, HEX3
        mov HEX3, HEX2
        mov HEX2, HEX1
        mov HEX1, HEX0
        mov HEX0, R4
        ljmp Forever ; Repeat forever
Scroll_Right:
        mov R4, HEX0
        mov HEX0, HEX1
        mov HEX1, HEX2
        mov HEX2, HEX3
        mov HEX3, HEX4
        mov HEX4, HEX5
        mov HEX5, R4
        ljmp Forever ; Repeat forever
```


Step 11

```
; Look-up table for 7-seg displays.
T_7seg:
        DB 40H, 79H, 24H, 30H, 19H
        DB 12H, 02H, 78H, 00H, 10H
        DB 08H, 03H, 46H, 21H, 06H
        DB 0EH

Display_Number:
        mov dptr, #T_7seg
        mov a, R7
        anl a, #0x0f ; Force bits 4 to 7 to zero
        movc a, @dptr+a ; Read from table
        mov HEX0, a ; Display low nibble
        mov a, R7
        swap a ; exchange bits 0 to 3 with bits 4 to 7
        anl a, #0x0f ; Force bits 4 to 7 to zero
        movc a, @dptr+a ; Read from table
        mov HEX1, a ; Display high nibble
        ret
```


Step 11 (cont.)

```
Forever:
        mov R7, #0x00
        lcall Display_Number
        lcall Delay
        mov R7, #0x55
        lcall Display_Number
        lcall Delay
        mov R7, #0xAA
        lcall Display_Number
        lcall Delay
        ljmp Forever ; Repeat forever
```


Step 12

```
; Look-up table for 7-seg displays.
Show mac
        mov R7, %0
        lcall Display_Number
        lcall Delay
endmac

main:
        mov SP, #0x7f
        mov LEDRA, #0 ; Bit addressable
        mov LEDRB, #0 ; Not bit addressable
Forever:
        Show(#0x00)
        Show(#0x55)
        Show(#0xAA)
        ljmp Forever ; Repeat forever
END
```


Step 13

```
; Look-up table for 7-seg displays.
Show mac
        mov R7, %0
        lcall Display_Number
        lcall Delay
endmac

main:
        mov SP, #0x7f
        mov LEDRA, #0 ; Bit addressable
        mov LEDRB, #0 ; Not bit addressable
        mov R5, #0
Forever:
        Show(AR5) ; Use R5 first and explain why it fails
        inc R5
        ljmp Forever ; Repeat forever
END
```


Step 14

```
; Look-up table for 7-seg displays.
Show mac
        mov R7, %0
        lcall Display_Number
        lcall Delay
endmac

main:
        mov SP, #0x7f
        mov LEDRA, #0 ; Bit addressable
        mov LEDRB, #0 ; Not bit addressable
        mov R5, #0
Forever:
        Show(AR5)
        mov a, R5
        add a, #1
        da a
        mov R5, a
        ljmp Forever ; Repeat forever
END
```


Step 15

```
Display_at mac
        mov dptr, #T_7seg
        mov a, %2
        anl a, #0x0f ; Force bits 4 to 7 to zero
        movc a, @dptr+a ; Read from table
        mov %0, a ; Display low nibble
        mov a, %2
        swap a ; exchange bits 0 to 3 with bits 4 to 7
        anl a, #0x0f ; Force bits 4 to 7 to zero
        movc a, @dptr+a ; Read from table
        mov %1, a ; Display high nibble
endmac

Increment_BCD mac
        mov a, %0
        add a, #1
        da a
        mov %0, a
endmac
```


Step 15 (cont.)

```
        mov R3, #0x12
        mov R4, #0x59
        mov R5, #0x48
Forever:
        Display_at(HEX4, HEX5, R3)
        Display_at(HEX2, HEX3, R4)
        Display_at(HEX0, HEX1, R5)
        lcall Delay
        Increment_BCD(R5)
        cjne a, #0x60, Forever
        mov R5, #0
        Increment_BCD(R4)
        cjne a, #0x60, Forever
        mov R4, #0
        Increment_BCD(R3)
        cjne a, #0x13, Forever
        mov R3, #1
        ljmp Forever ; Repeat forever
END
```

Step 16

```
        jb KEY.3, skip_hour
        mov R3, SWA
skip_hour:
        jb KEY.2, skip_min
        mov R4, SWA
skip_min:
        jb KEY.1, skip_sec
        mov R5, SWA
skip_sec:

        mov a, SWB ; SWB is not bit addressable, but the acc is!
        jb acc.1, Forever ; Do not increment!
```


University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Lecture 14: Microcomputer Integer Arithmetic I

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

Objectives

- Perform single-byte-unsigned arithmetic add and subtract operations.
- Perform multi-byte unsigned arithmetic add and subtract operations.
- Compare single-byte and multi-byte numbers (=,<=, <, >)
- Perform single-byte and multi-byte signed arithmetic add and subtract operations.
- Perform Binary to BCD and BCD to Binary conversion.

Unsigned-byte (8 bit) addition

Adding decimal numbers, by hand:

```
    4 4  <- LSD
  + 5 5
  -----
    9 9
```

Adding binary numbers, by hand:

```
                    LSB
    0 0 1 0 1 1 0 0   (44)
  + 0 0 1 1 0 1 1 1   (55)
  -----------------
    0 1 1 0 0 0 1 1   (99)
```

```
MOV A, #44
ADD A, #55
```


Unsigned-byte (8 bit) addition

Add the content of R0 and R1 and store it in R7:

```
MOV A, R0
ADD A, R1
MOV R7, A
```

Add the content of memory locations 50 and 51, and save the result in location 52:

```
MOV A, 50
ADD A, 51
MOV 52, A
```


Unsigned-byte (8 bit) subtraction

Subtract decimal numbers, by hand:

```
    5 5  <- LSD
  – 4 4
  -----
    1 1
```

Subtract binary numbers, by hand:

```
                    LSB
    0 0 1 0 1 1 0 0   (55)
  – 0 0 1 1 0 1 1 1   (44)
  -----------------
    0 0 0 0 1 0 1 1   (11)
```

```
CLR C
MOV A, #55
SUBB A, #44
```


Unsigned-byte (8 bit) subtraction

Subtract the content of R0 and R1 and store it in R7:

```
MOV A, R0
CLR C
SUBB A, R1
MOV R7, A
```

Subtract the content of memory locations 60 and 61, and save the result in register DPL:

```
MOV A, 60
CLR C
SUBB A, 61
MOV DPL, A
```

Unsigned multi-byte addition / subtraction and the Carry / Borrow flag

- We can handle integers of more than 8 bits by cascading addition/subtraction operation and use the carry/borrow flag.
- The MCS-51 have instructions to add both with carry and without carry, but it can only subtract with borrow!
  - ADD
  - ADDC
  - SUBB

16-bit addition

```
                 LSD
  2 0 4 7
+   3 2 2
  -------
  2 3 6 9
```

```
                  Carry
      High byte               Low byte

  0 0 0 0 0 1 1 1       1 1 1 1 1 1 1 1
+ 0 0 0 0 0 0 0 1       0 1 0 0 0 0 1 0
  ---------------       ---------------
  0 0 0 0 1 0 0 1       0 1 0 0 0 0 0 1
```


16-bit addition

In MCS-51 Assembly language:

```
mov a, #low(2047)
add a, #low(322)
mov b, a ; save low result in reg. b
mov a, #high(2047)
addc a, #high(322)
```

'high' and 'low' are assembler directives that take the upper or lower 8 bits of a 16-bit number.

16-bit addition

In MCS-51 Assembly language:

```
mov a, #0FFH
add a, #42H
mov b, a ; save low result in reg. b
mov a, #03H
addc a, #01H
```

Same as the previous slide, but you need to convert the numbers 2047 and 322 to hex (or binary!) and write them down in the program. Windows 'calc' is very handy for this!

32-bit addition

- Add the two 32 bit numbers located at RAM memory addresses 30H and 34H and store the result at XRAM address 100H.

NOTE: For regular internal memory we use the **MOV** instruction. For expanded memory we use the **MOVX** instruction.

First try: brute force!

```
mov a, 30H
add a, 34H
mov dptr, #100H
movx @dptr, a
mov a, 31H
addc a, 35H
inc dptr
movx @dptr, a
```

```
mov a, 32H
addc a, 36H
inc dptr
movx @dptr, a
mov a, 33H
addc a, 37H
inc dptr
movx @dptr, a
```


32-bit addition

- Add the two 32 bit numbers located at RAM memory addresses 30H and 34H and store the result at XRAM address 100H.
- Now, let us use assembler directives and indirect addressing (@R0, @R1, and @DPTR) in a compilable program:

32-bit addition

```
; Add32.asm: shows how to add two 32-bit numbers at RAM
; addresses 30H and 34H and place result at XRAM address
; 100H

$MODDE2

org 0000H
      ljmp myprogram

DSEG at 30H

Num1:  DS     4
Num2:  DS     4

XSEG at 100H
Result:     DS     4
```

32-bit addition

```
CSEG
myprogram:
        mov SP, #07FH
; Make Num1=55555555H for testing
        mov R0, #Num1
L1:     mov @R0, #55H
        inc R0
        cjne R0, #Num1+4, L1
; Make Num1=66666666H for testing
        mov R1, #Num2
L2:     mov @R1, #66H
        inc R1
        cjne R1, #Num2+4, L2
; Initialize pointers
        mov R0, #Num1
        mov R1, #Num2
        mov DPTR, #Result
```


32-bit addition

```
        clr c
        mov R2, #4
; Add the bytes, one by one
L3:     mov A, @R0
        addc A, @R1
        movx @dptr, A
        inc R0
        inc R1
        inc dptr
        djnz R2, L3

; Done!  Loop forever
forever:
        jmp forever
END
```


16-bit subtraction

LSD

```
  1 4 0 9 6
−   4 1 2 8
  ---------
    9 9 6 8
```

Borrow

High byte | Low byte

```
  0 0 1 1 0 1 1 1   0 0 0 1 0 0 0 0
− 0 0 0 1 0 0 0 0   0 0 1 0 0 0 0 0
  ---------------   ---------------
  0 0 1 0 0 1 1 0   1 1 1 1 0 0 0 0
```


16-bit subtraction

In MCS-51 Assembly language:

```
clr c ; the 'carry' is also the 'borrow'!
mov a, #low(14096)
subb a, #low(4128)
mov b, a ; save low result in register b
mov a, #high(3710H)
subb a, #high(1020H)
```

14096=3710H=0011011100010000B

4128=1020H=0001000000100000B

32-subtraction…

```
clr c
mov a, 30H
subb a, 34H
mov dptr, #100H
movx @dptr, a
mov a, 31H
subb a, 35H
inc dptr
movx @dptr, a
```

```
mov a, 32H
subb a, 36H
inc dptr
movx @dptr, a
mov a, 33H
subb a, 37H
inc dptr
movx @dptr, a
```


32-bit subtraction…

Simply modify the example given above for 32-bit addition:

```
        clr c
        mov R2, #4
; Subtract the bytes, one by one
L3:     mov A, @R0
        subb A, @R1
        movx @dptr, A
        inc R0
        inc R1
        inc dptr
        djnz R2, L3

; Done!  Loop forever
forever:
        jmp forever
END
```


Comparing numbers

- We use single or multi-byte subtraction to compare two numbers:
  - If the result of the subtraction is zero: minuend = subtrahend.
  - If the result of the subtraction is not zero: minuend ≠ subtrahend.
  - If 'borrow' set: minuend < subtrahend.
  - If no 'borrow' set: minuend >= subtrahend.
- To check, we use the 'zero' and 'carry' flags and the instructions: JZ, JNZ, JC, and JNC.

Are two 16-bit numbers Equal?

A 16-bit number is stored in R0 and R1, where R0 is the LSB. Write the assembly code to check if the stored number is 1000.

```
clr c
mov a, #low(1000)
subb a, R0
jnz NotEqual
mov a, #high(1000)
subb a, R1
jnz NotEqual
sjmp Equal
```

'NotEqual' and 'Equal' are labels somewhere else in the program.

Number1 > Number2?

Number1 is stored in R0 and R1, where R0 is the LSB. Number2 is stored at addresses 48H and 49H, where 48H is the LSB.

```
clr c
mov a, 48H
subb a, R0
mov a, 49H
subb a, R1
jc MyLabel ; jump if number1 > number2
```


Number1 < Number2?

Number1 is stored in R0 and R1, where R0 is the LSB. Number2 is stored at addresses 48H and 49H, where 48H is the LSB.

`clr c`
`mov a, 48H`
`subb a, R0`
`mov a, 49H`
~~subb a, R1~~
`jnc MyLabel ; jump if number1 < number2`

WRONG!

Is the carry set if number1 = number2?

On the other hand, this is a perfectly good (Number1 <= Number2)

Number1 < Number2?

Number1 is stored in R0 and R1, where R0 is the LSB. Number2 is stored at addresses 48H and 49H, where 48H is the LSB.

```
clr c
mov a, R0
subb a, 48H
mov a, R1
subb a, 49H
jc MyLabel ; jump if number1 < number2
```


Signed integer representation

True magnitude number representation:

$\boxed{0}$ 0 1 1 0 1 0 0 = +52

Sign bit

$\boxed{1}$ 0 1 1 0 1 0 0 = -52

Sign bit

Easy to understand, HARD to work with!

Signed integer representation

2's-complement number representation:

$\boxed{0}$ 0 1 1 0 1 0 0 = +52

Sign bit

$\boxed{1}$ 1 0 0 1 1 0 0 = -52

Sign bit

Very convenient to perform arithmetic!

2's Complement

To obtain the 2's complement of a number:

0 0 1 1 0 1 0 0	Original number: +52
1 1 0 0 1 0 1 1	1's complement
1	Add 1
1 1 0 0 1 1 0 0	2's complement

In 8051's assembly language:

```
MOV A, #52
CPL A
INC A
```

Or

```
CLR C
CLR A
SUBB A, #52
```


Signed Integer Addition

Case 1: two positive numbers:

	0 0 1 1 0 1 1 1	55
+	0 0 0 1 0 1 0 0	20
	0 1 0 0 1 0 1 1	75

Case 2: positive number larger than negative number:

	0 0 1 1 0 1 1 1	55
+	1 1 0 0 1 1 0 0	-52
	0 0 0 0 0 0 1 1	3

Signed Integer Addition

Case 3: negative number larger than positive number :

	0 0 0 1 0 1 0 0	20
+	1 1 0 0 1 1 0 0	-52
	1 1 1 0 0 0 0 0	-32

Case 4: Two negative numbers:

	1 1 1 1 0 0 0 1	-15
+	1 1 0 0 1 1 0 0	-52
	1 0 1 1 1 1 0 1	-67

Signed Integer Addition

Case 5: equal and opposite numbers :

```
    0 0 1 1 0 1 0 0    52
  + 1 1 0 0 1 1 0 0   -52
    0 0 0 0 0 0 0 0     0
```

This technique also works for decimal number calculations. It is particularly handy when performing subtraction by hand. For more information, check the "method of complements" in Wikipedia:

http://en.wikipedia.org/wiki/Method_of_complements

Method of Complements for Hand Calculation of Decimal Subtraction

```
   9 8 5 3 1 8 0 3 1
 – 4 5 1 7 0 2 4 6
```

'Negate' in base 10 and add 1 to get the ten's complement

```
   9 5 4 8 2 9 7 5 3
 +                 1
   9 5 4 8 2 9 7 5 4
```

Add the Ten's complement

```
     9 8 5 3 1 8 0 3 1
   + 9 5 4 8 2 9 7 5 4
   1 9 4 0 1 4 7 7 8 5
```

Correct answer!

Disregard!

Signed Integer Addition

Write the assembly code to add two 16-bit signed numbers at RAM addresses 20H-21H and 30H-31H and save the result in R0-R1.

```
mov A, 21H
add A, 31H
mov R1, A
mov A, 20H
addc A, 30H
mov R0, A
```


Signed Integer Subtraction

Exactly the same as signed addition. Just get the 2's complement of the subtrahend before the addition!

Write the assembly code to subtract the signed 16-bit number at addresses 30H-31H from the 16-bit signed number at RAM addresses 20H-21H. Save the result in R0-R1.

```
mov A, 31H
cpl A
add A, #1
mov R1, A
mov A, 30H
cpl A
addc A, #0
mov R0, A
```

```
mov A, 21H
add A, R1
mov R1, A
mov A, 20H
addc A, R0
mov R0, A
```


Binary to BCD Conversion

- Registers, counters, addresses, numbers, etc. in the microcontroller are represented in binary.
- People like to work with decimal numbers! Also, it is very easy to convert BCD to ASCII:

Code used to represent English characters and symbols in displays, terminals, and printers

```
mov R1,#91H ; BCD number
mov a, R1
swap a
anl a,#0FH
orl a,#30H ; ASCII "9" is #39H
mov 50H, a
mov a, R1
anl a,#0FH
orl a,#30H ; ASCII "1" is #31H
mov 51H, a
```


ASCII Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Binary to BCD conversion of 8-bit numbers in the 8051

```
; Eight bit number to convert passed in acc.
; Result in r1,r0
Bin2BCD_8bit:
        mov b, #100
        div ab
        mov r1, a     ; Save hundreds
        mov a, b      ; Remainder is in register b
        mov b, #10
        div ab
        swap a
        orl a, b
        mov r0, a     ; Save tens and ones
        ret
```


Binary to BCD Conversion: the double dabble algorithm

1. BIN: 01011011 BCD=000
   Multiply BDC by two and add the underlined bit:
   BCD=(000+000+0)=000
2. BIN: 01011011 BCD=000
   Multiply BDC by two and add the underlined bit:
   BCD=(000+000+1)=001
3. BIN: 01011011 BCD=001
   Multiply BDC by two and add the underlined bit:
   BCD=(001+001+0)=002
4. BIN: 01011011 BCD=002
   Multiply BDC by two and add the underlined bit:
   BCD=(002+002+1)=005

Binary to BCD Conversion : the double dabble algorithm

5. BIN: 01011011 BCD=005
   Multiply BDC by two and add the underlined bit:
   BCD=(005+005+1)=011
6. BIN: 01011011 BCD=011
   Multiply BDC by two and add the underlined bit:
   BCD=(011+011+0)=022
7. BIN: 01011011 BCD=022
   Multiply BDC by two and add the underlined bit:
   BCD=(022+022+1)=045
8. BIN: 01011011 BCD=045
   Multiply BDC by two and add the underlined bit:
   BCD=(045+045+1)=091 Final result is 91

Multiply a BCD number by 2.

- If the processor supports addition of BCD numbers then add the BCD number to itself.
- If the processor DOES NOT support addition of BCD numbers:
  - Add 3 (0011B) to each BCD digit > 4 (0100B).
  - Shift Left one bit.

BCD addition in the 8051: the "DA A" instruction.

- **Function:** Decimal Adjust Accumulator
- **Description:** DA adjusts the contents of the Accumulator to correspond to a BCD (Binary Coded Decimal) number after two BCD numbers have been added by the ADD or ADDC instruction. If the carry bit is set or if the value of bits 0-3 exceed 9, 06H is added to the accumulator. If the carry bit was set when the instruction began, or if 06H was added to the accumulator in the first step, 60H is added to the accumulator.

Page 2-39 of "MCS-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET" has more details about the works of this instruction!

How to multiply a BCD by two if BCD addition is not available.

Multiply BCD number 3925 by 2:

```
0011 1001 0010 0101   BCD number to multiply by 2
0000 0011 0000 0011   Correction before left shift
0011 1100 0010 1000   Sum
0111 1000 0101 0000   Shift left
  7    8    5    0
```

3925 * 2 = 7850

16-bit binary to BCD

```
;Converts the hex number in R0-R1 to BCD
;in R2-R3-R4
hex2bcd:
    mov R2, #0   ;Set BCD result to 00000
    mov R3, #0
    mov R4, #0
    mov R5, #16  ;Loop counter.
L0:
    mov a, R1  ;Shift R0-R1 left through carry
    rlc a
    mov R1, a
    mov a, R0
    rlc a
    mov R0, a
```

Continues…

16-bit binary to BCD

```
; Perform bcd + bcd + carry
; using BCD numbers
mov a, R4
addc a, R4
da a
mov R4, a
mov a, R3
addc a, R3
da a
mov R3, a
mov a, R2
addc a, R2
mov R2, a
djnz R5, L0
ret
```


BCD to Binary Conversion

- We may need to convert to binary any number provided by humans!
- Many chips work only with BCD numbers, for example Real Time Clocks (RTCs) often count in BCD only.
- One way to convert to BCD to binary is by multiplying by ten (10) and adding a four bit number:

BCD to Binary Conversion for 8-bit binary numbers in the 8051.

1. BCD=137 BIN: 00000000
   Multiply BIN by ten and add the underlined digit:
   BIN=((00000000*1010)+0001)=00000001
2. BCD=137 BIN: 00000001
   Multiply BIN by ten and add the underlined digit:
   BIN=((00000001*1010)+0011)=00001101
3. BCD=137 BIN: 00001101
   Multiply BIN by ten and add the underlined digit:
   BIN =((00001101*1010)+0111)=**10001001**

BCD to Binary Conversion for 8-bit binary numbers in the 8051.

- Convert a three digit BCD number (0 to 255) stored in R6-R7 to binary and save the result in R5.

```
mov a, R6
mov b, #10
mul ab
mov R5, a
mov a, R7
swap a
anl a,#0fH
add a, R5
mov b, #10
mul ab
mov R5, a
```

```
mov a, R7
anl a,#0fH
add a, R5
mov R5, a
```


BCD to Binary Conversion for n-bit binary numbers.

Convert 205 to binary:

```
205 ÷ 2 = 102  R = 1
102 ÷ 2 =  51  R = 0
 51 ÷ 2 =  25  R = 1
 25 ÷ 2 =  12  R = 1
 12 ÷ 2 =   6  R = 0
  6 ÷ 2 =   3  R = 0
  3 ÷ 2 =   1  R = 1
  1 ÷ 2 =   0  R = 1
```

To convert BCD to binary we need to divide a BCD number by two! This algorithm works in reverse from the double dabble algorithm!!!

205 = 11001101B = 0CDH

Divide BCD by two

- Shift the BCD number right by one
- If the most significant bit of a BCD digit is one, subtract 3 to that digit.
- The least significant bit of the least significant BCD digit is the remainder.

Divide BCD by two

Divide BCD number 3925 by 2:

```
0011 1001 0010 0101  BCD number to divide by 2
0001 1100 1001 0010  Right shift
0000 0011 0011 0000  Correction
0001 1001 0110 0010  Subtract
  1    9    6    2
```

3925 / 2 = 1962, remainder=1

16-bit BCD to binary

```
; bcd2hex: Converts the BCD number in R2-R3-R4
; to binary in R0-R1

rrc_and_correct:
    rrc a
    mov r6, psw ; Save carry (it is changed by the add)
    jnb acc.7, nocor1
    add a, #(100H-30H) ; subtract 3 from packed BCD MSD
nocor1:
    jnb acc.3, nocor2
    add a, #(100H-03H) ; subtract 3 from packed BCD LSD
nocor2:
    mov psw, r6 ; Restore carry
    ret
```


16-bit BCD to binary

```
bcd2hex:
    mov R5, #16 ;Loop counter.
L0:
    ; Divide BCD by two
    clr c
    mov a, R2
    lcall rrc_and_correct
    mov R2, a
    mov a, R3
    lcall rrc_and_correct
    mov R3, a
    mov a, R4
    lcall rrc_and_correct
    mov R4, a
```


16-bit BCD to binary

```
;Shift R0-R1 right through carry
mov a, R0
rrc a
mov R0, a
mov a, R1
rrc a
mov R1, a

djnz R5, L0
ret
```


Exercises

- Write an assembly subroutine for the 8051 that checks if a 32-bit number stored in registers R0 to R3 is zero.
- Write the assembly equivalent of this piece of C code (the size of int is 2 bytes):

```
unsigned int x, y;
unsigned char z;
.
[other code comes here]
.
if (x>y) z=0;
else z=1;
```

Assembly for this!

University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Lecture 15: Microcomputer Integer Arithmetic II

Dr. Jesús Calviño-Fraga
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 16, 2017

Objectives

- Perform multiplication and division.
- Understand and use the 'sjmp' and 'ljmp' instructions.
- Understand and use the 8051 stack.
- Understand and use the 'lcall' and 'ret' instructions.
- Understand and use the 'push' and 'pop' instructions.

Multiplication

- Most modern processors include a 'multiply' instruction.
- Multiplication can be implemented using the shift/rotate instructions.
- To multiply 8 bits in the 8051 we can just use:

MUL AB

Multiplication of Binary Numbers

			1	0	0	1	Multiplicand = 9
			1	0	1	1	Multiplier = 11
			1	0	0	1	Partial products
		1	0	0	1		
	0	0	0	0			
1	0	0	1				
1	1	0	0	0	1	1	Final Product=99

It takes too much memory to store the partial products, so we add them as we go:

Multiplication of Binary Numbers

			1	0	0	1	First partial product
		1	0	0	1		Second partial product
		1	1	0	1	1	
	0	0	0	0			Third partial product
	0	1	1	0	1	1	
1	0	0	1				Fourth partial product
1	1	0	0	0	1	1	Final Product=99

Example: 16-bit Multiply

Write a program that multiplies the two 16-bit numbers 'Num1' and 'Num2'. Store the 32-bit result in 'Result'. Use the rotate left and right instructions to complete the program. Do not use the assembly instruction 'mul ab'.

In mat32.asm used in lab 6, uses a more efficient way of multiplying multi-byte numbers using the 'mul ab' instruction.

16-bit Multiply

```
; Mul16.asm: Multiplies 'Num1' by 'Num2'.  Stores
; 32-bit result in 'Result'.

$MOD52

org 0000H
      ljmp myprogram

DSEG at 30H
Num1:           DS    2
Num2:           DS    2
Result:         DS    4
PartProd:       DS    4
Mult:           DS    2
```


16-bit Multiply

```
CSEG
myprogram:

      ; Load Num1 and Num2 with test values
      mov Num1, #low(300)
      mov Num1+1, #high(300)
      mov Num2, #low(20)
      mov Num2+1, #high(20)

      mov R3, #16 ; We have 16 partial products

      ;Copy the numbers to the work variables
      mov Mult,   Num1
      mov Mult+1, Num1+1
      mov PartProd,   Num2
      mov PartProd+1, Num2+1
      mov PartProd+2, #0
      mov PartProd+3, #0
```

16-bit Multiply

```
        ;Initialize result to zero
        clr a
        mov Result,   a
        mov Result+1, a
        mov Result+2, a
        mov Result+3, a

Mult16Loop:
        ;Shift the multiplicand right
        clr c
        mov a, Mult+1
        rrc a
        mov Mult+1, a
        mov a, Mult
        rrc a
        mov Mult, a
; Add the Partial product to the result only if carry is set
        jnc SkipAdd
```


16-bit Multiply

```
; Add the Partial product to the result
        mov R2, #4
        mov R0, #PartProd
        mov R1, #Result
        clr c
AL0:
        mov a, @R0
        addc a, @R1
        mov @R1, a
        inc R0
        inc R1
        djnz R2, AL0

SkipAdd:
        djnz R3, ShiftLeft
        sjmp forever
```


16-bit Multiply

```
; Shift the partial product left
ShiftLeft:
        clr c
        mov R1, #4
        mov R0, #PartProd
SL0:
        mov a, @R0
        rlc a
        mov @R0, a
        inc R0
        djnz R1, SL0
        sjmp Mult16Loop

; Done!  Loop forever
forever:
        sjmp forever
END
```


Division

- Division (efficient division) is the most complicated of the arithmetic operations.
- The 8051 includes a function to divide 8-bit by 8-bit unsigned integers:
  - ‘div ab’ divides register A by register B. A holds the quotient, B holds the remainder.
- Dividing bigger numbers requires the implementation of a multi-byte division algorithm:

n-bit division

- To perform division efficiently we can use repeated subtraction with the recursive formula:

$$P_{j+1} = P_j \times R - q_{n-(j+1)} D$$

P is the remainder
R is the radix
q_m is the digit at position m
D is the denominator

n-bit integer division algorithm

- If the denominator is zero finish with error!
- Shift the denominator left until the most significant digit (for a given radix) is different from zero. Make *n*=number_of_shifts +1.
- Repeat *n* times the recurrence formula:

$$P_{j+1} = (P_j - q_{n-(j+1)} D) \times R$$

If you repeat more than n times you'll get the fractions of the division.

Division Example 1

- Solve 3391 ÷ 17 using the algorithm from the previous slide. Used radix 10.

$$\frac{3391}{17} \rightarrow \frac{3391}{1700}, n=3, R=10$$

$$P = (3391 - (1) \times 1700) \times 10 = 16910 \rightarrow q_2 = 1$$

$$P = (16910 - (9) \times 1700) \times 10 = 16100 \rightarrow q_1 = 9$$

$$P = (16100 - (9) \times 1700) \times 10 = 8000 \rightarrow q_0 = 9$$

Answer is: 199

Division Example

- Check the library “Math32.asm” used in Lab 6. The function “div32” implements the algorithm above.
- See if you can follow what “div32” does!

sjmp (short jump)

– Requires two bytes: Opcode (80H) and 8-bit operand.
– The jump address is an “offset” to the next instruction.
– The 8-bit operand is a two-complements signed number. We can jump forward 127 bytes or back 128 bytes counting from the instruction after the sjmp.
– All conditional jumps behave just like sjmp.
– Also know as a ‘relative’ jump.

sjmp examples

```
001E      L1:
001E 00        nop
001F 00        nop
0020 00        nop
0021 00        nop
0022 8006      sjmp L2
0024 80F8      sjmp L1
0026 00        nop
0027 00        nop
0028 00        nop
0029 00        nop
002A      L2:
```

24H+06H=2AH

0F8H → 07+1 → -8:

26H-8H=1EH

Conditional Jumps

Mnemonic	Opcode	B	C	Function
JZ rel	60H	2	2/3	(PC) = (PC) + 2 IF (A) = 0 THEN (PC) = (PC) + rel
JNZ rel	70H	2	2/3	(PC) = (PC) + 2 IF (A) ≠ 0 THEN (PC) = (PC) + rel
JC rel	40H	2	2/3	(PC) = (PC) + 2 IF (C) = 1 THEN (PC) = (PC) + rel
JNC rel	50H	2	2/3	(PC) = (PC) + 2 IF (C) ≠0 THEN (PC) = (PC) + rel
JB bit, rel	20H	3	3/4	(PC) = (PC) + 3 IF (bit) = 1 THEN (PC) = (PC) + rel
JNB bit, rel	30H	3	3/4	(PC) = (PC) + 3 IF (bit) = 0 THEN (PC) = (PC) + rel
JBC bit, rel	10H	3	3/4	(PC) = (PC) + 3 IF (bit) = 1 THEN (bit) = 0 and (PC) = (PC) + rel

Conditional Jumps

WARNING: These change the carry flag!

Mnemonic	Opcode	B	C	Function
CJNE A, direct, rel	B5H	3	3/4	Compare and jump if not equal rel
CJNE A, #data, rel	B4H	3	3/4	Compare and jump if not equal rel
CJNE Rn, #data, rel	B8H-BFH	3	3/4	Compare and jump if not equal rel
CJNE @Ri, #data, rel	B6H-B7H	3	3/4	Compare and jump if not equal rel
DJNZ Rn, rel	D8H-DFH	3	2/3	Decrement and Jump if not zero
DJNZ direct,rel	D5H	3	3/4	Decrement and Jump if not zero

ljmp (long jump)

- Can jump anywhere in the 64k code memory space.
- Requires three bytes: opcode (02H) + 16-bit address:

```
0000 02001E    ljmp myprogram
001B 021803    ljmp 1803H
```


The 8051 stack

- We need the stack to use the lcall instruction.
- The stack is an area of memory where variables can be stacked. It is a LIFO memory: the last variable you put in is the first variable that comes out.
- Special Function Register SP (stack pointer) points to the beginning of the stack. SP in the 8051 is incremented **before** it is used (for push), or used and them decremented (for pop).
- After reset, SP is set to 07H. If you have variables in internal RAM, any usage of the stack is likely to corrupt them. Solution: at the beginning of your program set the SP special function register so it points to free memory:
  mov SP, #7FH ; Set the stack pointer to idata start

‘lcall’ and ‘ret’ instructions

- The ‘lcall’ instructions pushes the address of the next instruction (16-bit, LSB first) into the stack and jumps to the address passed as an operand to the ‘lcall’ instruction.
- The ret instruction pops the address stored in the stack and then jumps to that address.
- The ‘lcall’ can call any address in the 64k code memory space. Works similarly to ‘ljmp’…

Push and Pop

- When the microcontroller executes a push into the stack it:
  a) Increments the SP.
  b) Saves the value in the internal RAM location pointed by the SP.
- When the microcontroller execute a pop from the stack it:
  a) Retrieves the value from the internal RAM location pointed by the SP.
  b) Decrements the SP.
- As you may have suspected, the 8051 (as well as most other microprocessors!) have push and pop instructions.

lcall example

```
; Blinky.asm: blinks an LED connected to LEDR0
$MODDE0CV

org 0000H
    ljmp myprogram

;The clock in the CV-8052 is 33.3333MHz. (1 cycle=30ns)
WaitHalfSec:
    mov R2, #90
L3: mov R1, #250
L2: mov R0, #250
L1: djnz R0, L1 ; 3 machine cycles-> 3*30ns*250=22.5us
    djnz R1, L2 ; 22.5us*250=5.625ms
    djnz R2, L3 ; 5.625ms*90=0.506s (give or take)
    ret
```

Subroutine

lcall example

```
myprogram:
    mov SP, #7FH
    ; Turn off all LEDs...
    mov LEDRA, #0
    mov LEDRB, #0
M0:
    cpl LEDRA.0
    lcall WaitHalfSec
    sjmp M0
END
```


Saving and Restoring Registers to/from the Stack using push and pop

- Before using the stack (lcall, push, pop) make sure you set the SP special function register.
- Popular registers to push/pop: ACC, DPL, DPH, PSW, R0 to R7.
- Pop registers from the stack in the **REVERSE** order you pushed them! Remember the stack is a LIFO.

Push and Pop Example

```
WasteTime:
    push Acc
    push B
    push dpl
    mov Acc, #100
L3: mov B, #100
L2: mov dpl, #100
L1: djnz dpl, L1
    djnz B, L2
    djnz Acc, L3
    pop dpl
    pop B
    pop Acc
    ret
```


Common bug!

```
WasteTime:
    push B
    push Acc
    push dpl
    mov Acc, #100
L3: mov B, #100
L2: mov dpl, #100
L1: djnz dpl, L1 ; 3 bytes, 2 machine cycles
    djnz B, L2
    djnz Acc, L3
    pop dpl
    pop B
    pop Acc
    ret
```

Where is it?
pops are in the wrong order!

Push/Pop for R0 to R7

```
WaitHalfSec:
    push AR0
    push AR1
    push AR2
    mov R2, #20
L3: mov R1, #250
L2: mov R0, #184
L1: djnz R0, L1 ; 2 machine cycles-> 2*0.27126us*184=100us
    djnz R1, L2 ; 100us*250=0.025s
    djnz R2, L3 ; 0.025s*20=0.5s
    pop AR2
    pop AR1
    pop AR0
    ret
```

The extra 'A' is for 'direct address' of register R0. This is only required for registers R0 to R7

Stack Trace Example

For the program shown write down the stack values in the space provided when the execution reaches the indicated point in code.

Note: 'stack trace' makes good exam questions!

```
1000            myprogram:
1000 75813F         mov SP, #3FH
1003 7410           mov a, #10H
1005 C0E0           push acc
1007 75F0F0         mov b, #0F0H
100A 12100F         call a_x_b_plus1
100D            forever:
100D 80FE           jmp forever
100F            a_x_b_plus1:
100F C0E0           push acc
1011 C0F0           push b
1013 A4             mul ab
1014 04             inc a ; ← HERE!!!!
1015 D0F0           pop b
1017 D0E0           pop acc
1019 22             ret
```

Address	40H	41H	42H	43H	44H	45H	46H	47H
Value								

Stack Trace Example

```
1000            myprogram:
1000 75813F         mov SP, #3FH
1003 7410           mov a, #10H
1005 C0E0           push acc
1007 75F0F0         mov b, #0F0H
100A 12100F         call a_x_b_plus1
100D            forever:
100D 80FE           jmp forever
100F            a_x_b_plus1:
100F C0E0           push acc
1011 C0F0           push b
1013 A4             mul ab
1014 04             inc a ; ← HERE!!!!
1015 D0F0           pop b
1017 D0E0           pop acc
1019 22             ret
```

2 bytes!

Address	40H	41H	42H	43H	44H	45H	46H	47H
Value	??	??	??	??	??	??	??	??

Using a debugger (cmon51):

```
P89LPC9351> r
PC=0000 A=00 PSW=00 B=00 IE=00 DPL=00 DPH=00 SP=07 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
0000: 02 10 00  ljmp    1000
P89LPC9351> d 40 10
D:40:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ................
P89LPC9351> s
PC=1000 A=00 PSW=00 B=00 IE=08 DPL=00 DPH=00 SP=07 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1000: 75 81 3f  mov     SP,#3f
P89LPC9351> s
PC=1003 A=00 PSW=00 B=00 IE=08 DPL=00 DPH=00 SP=3f REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1003: 74 10     mov     a,#10
P89LPC9351> s
PC=1005 A=10 PSW=01 B=00 IE=08 DPL=00 DPH=00 SP=3f REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1005: c0 e0     push    ACC
P89LPC9351> s
PC=1007 A=10 PSW=01 B=00 IE=08 DPL=00 DPH=00 SP=40 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1007: 75 f0 f0  mov     B,#f0
```


Using a debugger (cmon51):

```
P89LPC9351> s
PC=100a A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=40 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
100a: 12 10 0f  lcall   100f
P89LPC9351> s
PC=100f A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=42 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
100f: c0 e0     push    ACC
P89LPC9351> s
PC=1011 A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=43 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1011: c0 f0     push    B
P89LPC9351> s
PC=1013 A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=44 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1013: a4        mul     ab
P89LPC9351> s
PC=1014 A=00 PSW=04 B=0f IE=08 DPL=00 DPH=00 SP=44 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1014: 04        inc     a
P89LPC9351> d 40 10
D:40:  10 0d 10 10 f0 14 10 04 : 00 00 00 00 00 00 00 00  ................
```

Answer!

Exercises

- Write an assembly program to multiply the 24-bit binary number stored in registers R2, R1, R0 (R0 is the least significant byte) by 10 (decimal). Save the result in R3, R2, R1, R0. Use the MUL AB instruction.
- Write an assembly program to find the approximate square root of a 16-bit number stored in registers DPH and DPL. Save the result to register R7. Tip: Use a binary search algorithm to find the square root quickly!

Exercises

- A common way of passing parameters to a function is via the stack. Modify the function **WaitHalfSec** so that it receives the number of half-seconds to wait in the stack. (Note: this problem is not as trivial as it sounds. You may need to increment and/or decrement register SP to solve this problem)
- Most C programs pass parameters to functions via the stack. Also C programs use the stack to allocate automatic variables (local variables defined within the function). This works fine most of the time, but sometimes a condition commonly known as "stack overflow" occurs. Explain what causes "stack overflow".

University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Lecture 16-17: Stack, Interrupts, Timers

Dr. Jesús Calviño-Fraga
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

April 6, 2022

Objectives

- Understand and use the 8051 stack.
- Understand and use the 'lcall' and 'ret' instructions.
- Understand and use the 'push' and 'pop' instructions.
- Setup and use Interrupts.
- Understand and use Timers/Counters.

The 8051 stack

- We need the stack to use the lcall instruction as well as interrupts.
- The stack is an area of memory where variables can be stacked. It is a LIFO memory: the last variable you put in is the first variable that comes out.
- Special Function Register SP (stack pointer) points to the beginning of the stack. SP in the 8051 is incremented **before** it is used (for push), or used and them decremented (for pop).
- After reset, SP is set to 07H. If you have variables in internal RAM, any usage of the stack is likely to corrupt them. Solution: at the beginning of your program set the SP special function register so it points to free memory:
  mov SP, #7FH ; Set the stack pointer to idata start

'lcall' and 'ret' instructions

- The 'lcall' instructions pushes the address of the next instruction (16-bit, LSB first) into the stack and jumps to the address passed as an operand to the 'lcall' instruction.
- The 'ret' instruction pops the address stored in the stack and then jumps to that address.
- The 'lcall' can call any address in the 64k code memory space. Works similarly to 'ljmp'…

Push and Pop

- When the microcontroller executes a push into the stack, it:
  a) Increments the SP.
  b) Saves the value in the internal RAM location pointed by the SP.
- When the microcontroller executes a pop from the stack, it:
  a) Retrieves the value from the internal RAM location pointed by the SP.
  b) Decrements the SP.
- The 8051 (as well as most other microprocessors!) have push and pop instructions.

lcall example

```
; Blinky.asm: blinks an LED connected to LEDR0
$MODDE0CV

org 0000H
    ljmp myprogram

;The clock in the CV-8052 is 33.3333MHz. (1 cycle=30ns)
WaitHalfSec:
    mov R2, #90
L3: mov R1, #250
L2: mov R0, #250
L1: djnz R0, L1 ; 3 machine cycles-> 3*30ns*250=22.5us
    djnz R1, L2 ; 22.5us*250=5.625ms
    djnz R2, L3 ; 5.625ms*90=0.506s (give or take)
    ret
```

Subroutine

A subroutine (or function) starts with a label and ends with a 'ret' or 'iret'.

lcall example

```
myprogram:
    mov SP, #7FH
    ; Turn off all LEDs...
    mov LEDRA, #0
    mov LEDRB, #0
M0:
    cpl LEDRA.0
    lcall WaitHalfSec
    sjmp M0
END
```

Never jump into a subroutine!

Saving and Restoring Registers to/from the Stack using push and pop

- Before using the stack (lcall, push, pop) make sure you set the SP special function register.
- Popular registers to push/pop: ACC, DPL, DPH, PSW, R0 to R7 (using their addresses: AR0 to AR7).
- Pop registers from the stack in the **REVERSE** order you pushed them! Remember the stack is a LIFO.

Push and Pop Example

```
WasteTime:
    push Acc
    push B
    push dpl
    mov Acc, #100
L3: mov B, #100
L2: mov dpl, #100
L1: djnz dpl, L1
    djnz B, L2
    djnz Acc, L3
    pop dpl
    pop B
    pop Acc
    ret
```


Common bug!

```
WasteTime:
    push B
    push Acc
    push dpl
    mov Acc, #100
L3: mov B, #100
L2: mov dpl, #100
L1: djnz dpl, L1 ; 3 bytes, 2 machine cycles
    djnz B, L2
    djnz Acc, L3
    pop dpl
    pop B
    pop Acc
    ret
```

Where is it?

pops are in the wrong order!

Push/Pop for R0 to R7

```
WaitHalfSec:
    push AR0
    push AR1
    push AR2
    mov R2, #20
L3: mov R1, #250
L2: mov R0, #184
L1: djnz R0, L1 ; 2 machine cycles-> 2*0.27126us*184=100us
    djnz R1, L2 ; 100us*250=0.025s
    djnz R2, L3 ; 0.025s*20=0.5s
    pop AR2
    pop AR1
    pop AR0
    ret
```

The extra 'A' is for 'direct address' of register R0. This is only required for registers R0 to R7

Interrupts

- 'Interrupts' are a means of executing subroutines automatically without using the 'lcall' instruction. The only difference is that the subroutine that is automatically called must end with 'reti' instead of 'ret'.
- Associated with external logic that requires CPU attention on command.
- Interrupt uses:
  - Handshake I/O thus preventing CPU from being tied up.
  - Providing a way to handle some errors: illegal opcodes, dividing by 0, power failure, etc.
  - Getting the CPU to perform periodic tasks: generate square waves, keep time of day, measure frequency, etc.
  - Waking up the processor when in low power mode.

Interrupts

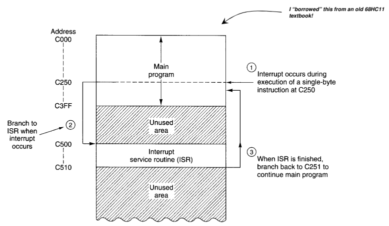


FIGURE 8.11 Example of how an interrupt causes a change in the execution of a program.

Interrupts

- Most processors provide a way of enabling / disabling all maskable interrupts. For the 8051:
  - ***clr EA*** ;Disable interrupts
  - ***setb EA*** ;Enable interrupts
- Some other interrupts are non-maskable and they MUST be serviced. For example, the X86 has the "Non-Maskable Interrupt" NMI.
- Maskable interrupts can be enabled/disabled individually. For the 8051 use register IE:

IE: INTERRUPT ENABLE REGISTER. (Address A8H)

EA	EC	ET2	ES	ET1	EX1	ET0	EX0

Bit	Name	Description
7	EA	Interrupt Enable Bit: EA = 1 interrupt(s) can be serviced, EA = 0 interrupt servicing disabled.
6	BPE	Breakpoint Enable bit. (CV-8052)
5	ET2	Timer 2 Interrupt Enable. (8052)
4	ES	Serial Port Interrupt Enable
3	ET1	Timer 1 Overflow Interrupt Enable.
2	EX1	External Interrupt 1 Enable.
1	ET0	Timer 0 Overflow Interrupt Enable.
0	EX0	External Interrupt 0 Enable.

Interrupt Service Routines (ISR) Vectors

- The 8051 will ***lcall*** to an specific memory location when an interrupt occurs. They are different for different 8051 variants. For the CV-8052 this are the interrupts supported and their vector addresses:

Interrupt source	Address
External 0	0003H
Timer 0	000BH
External 1	0013H
Timer 1	001BH
Serial port	0023H
Timer 2	002BH

Interrupt Service Routines (ISR) Vectors

- Notice that there are only 8 bytes available between vectors. Not enough for a decent ISR, but more than enough for a *ljmp* instruction!
- IF you enable a particular interrupt, there **MUST** be an ISR, or your program **WILL** crash. A fool proof code technique is to setup all the ISR vectors and place a ***reti*** (return from interrupt) instruction for those that are not used (next example).
- In assembly language you can use the "*org*" directive to set an ISR vector.
- To return from an ISR use the ***reti*** instruction. To return from a normal routine use the ***ret*** instruction.

Example 1

```
; Basic interrupt setup

; We need the register definitions for the 8052:
$MOD52

org 0h
ljmp myprogram

; Interrupt Service Routines (see page 2-12 of MCS-51 bible)
; Notice that there is not much space to put code between
; service routines, but enough to put a ljmp!

; External interrupt 0
org 3h
reti

; Timer 0 interrupt
org 0bh
reti
```

Dummy ISRs

WARNING: org directives must be sequential!

Example 1 (cont.)

```
; External interrupt 1
org 13h
reti

; Timer 1 interrupt
org 1bh
reti

; Serial port interrupt
org 23h
reti

; Timer 2 interrupt
org 2bh
reti

; Dummy program, just to compile and see...
myprogram:
    mov R1, #00H ; do something
    sjmp myprogram
END
```

Dummy ISRs

Example 2: Enable timer 0 interrupt and setup an ISR

```
myprogram:
        ; Enable timer 0
        mov a, TMOD
        anl a, #0f0H
        orl a, #00000010B ; 8-bit auto reload timer (this is in binary)
        mov TMOD, a
        mov TH0, #080H ; Set the interrupt rate
        setb TR0 ; Enable timer 0
        setb ET0 ; Enable timer 0 interrupt

        setb EA ; Enable all interrupts!

forever:
        [...other code here...]
        jmp forever
```

Example 2: (cont.) the ISR.

```
; Timer 0 interrupt
org 0bh
   cpl P1.1 ; Check this pin with the scope!
   reti
```


Example 2: use a *ljmp* to go to the ISR

```
; Timer 0 interrupt
org 0bh
   ljmp timer0_ISR

; Other ISR vectors come here! (Not shown to save space)

; Actual ISR for timer 0.
timer0_ISR:
   cpl P1.1
   reti
```


Saving and Restoring Registers in the Stack

- If your ISR routine uses a register, you must make sure that it will remain **unmodified** before returning to the interrupted program.
- As mentioned before you use the instructions ***push***/***pop*** to save/restore registers to/from the stack.
- Additionally, you could use one of four available register banks in your ISR.

8051's Timers/Counters

- The original 8051 has only two timers/counters: 0 and 1.
- Newer 8051 microcontrollers usually have:
  1. The 8051 timers/counters: timers 0 and 1
  2. The 8052 timer/counter: timer 2
  3. Additional timers (3, 4, 5, etc.) Not available in the CV-8052.
  4. The Programmable Counter Array (PCA). Not available in the CV-8052, but very common in many other processors.
- Let us begin with timers 0 and 1:

Timer 0 and Timer 1 Operation Modes (Section 3-10 of MCS-51 manual)

- Timer 0 and 1 have four modes of operation:
  - Mode 0: 13-bit timer/counter (compatible with the 8048 microcontroller, the predecessor of the 8051). Do not use this mode; use mode 1 instead!
  - Mode 1: 16-bit timer/counter.
  - Mode 2: 8-bit auto reload timer counter.
  - Mode 3: Special mode 8-bit timer/counter (timer 0 only). (I have never used it!)
- Timer 1 can be used as baud rate generator for the serial port. Some 8051/8052 microcontrollers have a dedicated baud rate generator.

TMOD timer/counter mode control register (Address 89H)

Is this SFR bit addressable?

Timer 1				Timer 0			
GATE	C/T*	M1	M0	GATE	C/T*	M1	M0

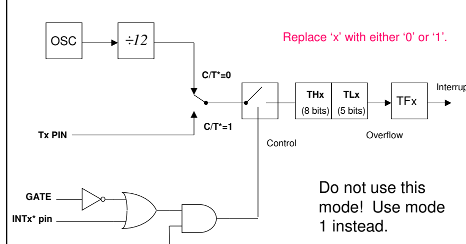
Bit	Name		Description
7 & 3	GATE		1: uses either INT0 or INT1 pins to enable/disable the timer/counter
6 & 2	C/T*		0: timer; 1: counter (pins T0 and T1)
All the other pins!	M1	M0	
	0	0	13-bit timer/counter
	0	1	16-bit timer/counter
	1	0	8-bit auto-reload timer/counter
	1	1	Special mode

TCON: timer/counter control register. (Address 88H)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

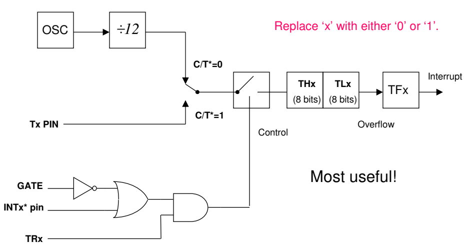
Bit	Name	Description
7	TF1	Timer 1 overflow flag.
6	TR1	Timer 1 run control.
5	TF0	Timer 0 overflow flag.
4	TR0	Timer 0 run control.
3	IE1	Interrupt 1 flag.
2	IT1	Interrupt 1 type control bit.
1	IE0	Interrupt 0 flag.
0	IT0	Interrupt 0 type control bit.

Timer/Counter 0 or 1 in Mode 0



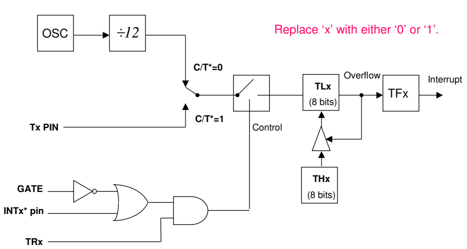
Do not use this mode! Use mode 1 instead.

Timer/Counter 0 or 1 in Mode 1



Most useful!

Timer/Counter 0 or 1 in Mode 2



Timer/Counter 0 in Mode 2

```
myprogram:
    ; After reset, the stack pointer register is set to 07h
    ; We may need space for variables, so move the SP
    mov SP, #7fH
    ; Enable timer 0
    mov a, TMOD
    anl a, #0f0H
    orl a, #00000010B ; GATE=0, C/T*=0, M1=1, M0=0: 8-bit auto reload timer
    mov TMOD, a
    mov TH0, #080H ; Set the interrupt rate
    setb TR0 ; Enable timer 0
    setb ET0 ; Enable timer 0 interrupt
    setb EA
forever:
    .
    .
    .
    .
    .
    .
    jmp forever
```

$$Rate = \frac{12}{OSC} \times (100H - TH0)$$

$$Rate = \frac{12}{33.3333MHz} \times (100H - 80H) = 46.08\mu s$$

Timer/Counter 2 (Page 3-12 of MCS51 manual)

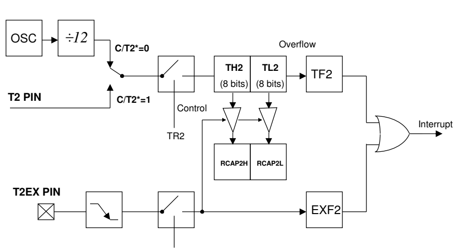
- It is a 16-bit timer/counter.
- It has four modes of operation:
  - Capture
  - Auto-reload
  - Baud rate generation
  - Programmable clock out (not implemented in CV-8052)

T2CON: timer/counter 2 control register. (Address C8H)

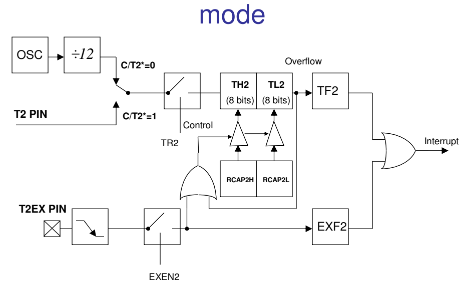
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2*	CP/RL2*

Bit	Name	Description
7	TF2	Timer/counter 2 overflow flag.
6	EXF2	Timer/counter 2 external flag.
5	RCLK	Receive clock flag.
4	TCLK	Transmit clock flag.
3	EXEN2	Timer/Counter 2 external enable.
2	TR2	Start/stop for timer/counter 2.
1	C/T2*	Timer or Counter select.
0	CP/RL2*	Capture/Reload Flag.

Timer/Counter 2 in capture mode



Timer/Counter 2 in auto-reload mode



Example: Time Delay Using a Timer

- To use a timer to implement a delay we need to:
  - Initialize the timer: use TMOD SFR.
  - Load the timer: use THx and TLx.
  - Clear the timer overflow flag: TFx=0;
  - Start the timer: Use TRx.
  - Check the timer overflow flag: Use TFx.

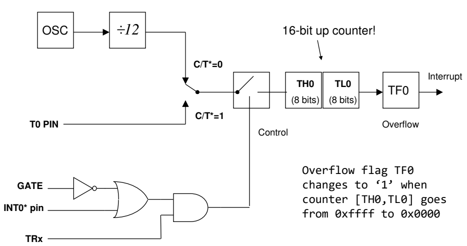
For the registers above ‘x’ is either ‘0’ for timer 0, or ‘1’ for timer 1.

Time Delay Using a Timer

- Implement a 10 ms delay subroutine using timer 0. Assume the routine will be running in a CV-8052 soft processor.

First, we have to find the divider (TH0, TL0) needed for a 10 ms delay…

Timer 0 in Mode 1



Overflow flag TF0 changes to '1' when counter [TH0,TL0] goes from 0xffff to 0x0000

Calculating TH0 and TL0

$$\text{Rate}=\frac{CLK/12}{2^{16}-[\text{THn,TLn}]}=\frac{33.3333\text{MHz}/12}{65536-[\text{THn,TLn}]}$$

$$[\text{THn,TLn}]=65536-\frac{2.77777\text{MHz}}{\text{Rate}}=65536-\frac{2.77777\text{MHz}}{(1/10\text{ms})}=37758$$

Maximum delay achievable?

$$\text{Rate}=\frac{CLK/12}{2^{16}-[\text{THn,TLn}]}=\frac{2.777777\text{MHz}}{65536-[\text{THn,TLn}]}$$

$$[\text{THn,TLn}]=0$$

$$\text{Rate}=\frac{2.777777\text{MHz}}{65536}=42.39Hz\rightarrow 23.59ms$$

Time Delay Using Timer 0

```
Wait10ms:
    ; Initialize the timer
    mov a, TMOD
    anl a, #11110000B ; Clear bits for timer 0, keep bits for timer 1
    orl a, #00000001B ; GATE=0, C/T*=0, M1=0, M0=1: 16-bit timer
    mov TMOD, a
    clr TR0 ; Disable timer 0
    ; Load the timer [TH0, TL0]=65536-(2777777/(1/10E-3))
    mov TH0, #high(37758)
    mov TL0, #low(37758)
    clr TF0 ;Clear the timer flag
    setb TR0 ; Enable timer 0
Wait10ms_L0:
    jnb TF0, Wait10ms_L0 ; Wait for overflow
    ret
```


Time Delay Using Timer 0

```
; Let the Assembler do the calculation for us!
XTAL equ 33333333
FREQ equ 100 ; 1/100Hz=10ms
RELOAD_TIMER0_10ms equ 65536-(XTAL/(12*FREQ))

Wait10ms:
    ; Initialize the timer
    mov a, TMOD
    anl a, #11110000B ; Clear bits for timer 0, keep bits for timer 1
    orl a, #00000001B ; GATE=0, C/T*=0, M1=0, M0=1: 16-bit timer
    mov TMOD, a
    clr TR0 ; Disable timer 0
    mov TH0, #high(RELOAD_TIMER0_10ms )
    mov TL0, #low(RELOAD_TIMER0_10ms )
    clr TF0 ;Clear the timer flag
    setb TR0 ; Enable timer 0
Wait10ms_L0:
    jnb TF0, Wait10ms_L0 ; Wait for overflow
    ret
```


CV-8052 Pin Assignments

	JP1				JP2		
LCD_DATA[0]	1	2	LCD_DATA[1]	P0.0	1	2	P0.1
LCD_DATA[2]	3	4	LCD_DATA[3]	P0.2	3	4	P0.3
LCD_DATA[4]	5	6	LCD_DATA[5]	P0.4	5	6	P0.5
LCD_DATA[6]	7	8	LCD_DATA[7]	P0.6	7	8	P0.7
LCD_EN	9	10	LCD_RS	P1.0	9	10	P1.1
5V	11	12	GND	5V	11	12	GND
LCD_RW	13	14	TXD	P1.2	13	14	P1.3
LCD_ON	15	16	RXD	P1.4	15	16	P1.5
FL_DQ[0]	17	18	FL_DQ[1]	P1.6	17	18	P1.7
FL_DQ[2]	19	20	FL_DQ[3]	P2.0	19	20	P2.1
FL_DQ[4]	21	22	FL_DQ[5]	P2.2	21	22	P2.3
FL_DQ[6]	23	24	FL_DQ[7]	P2.4	23	24	P2.5
FL_RST_N	25	26	FL_WE_N	P2.6	25	26	P2.7
FL_OE_N	27	28	FL_CE_N	P3.0	27	28	P3.1
3.3V	29	30	GND	3.3V	29	30	GND
TDO	31	32	TDI	P3.2	31	32	P3.3
TCS	33	34	TCK	P3.4	33	34	P3.5
Not used	35	36	Not used	P3.6	35	36	P3.7
T0	37	38	**T1**	INT0	37	38	INT1
T2	39	40	T2EX	Not Used	39	40	Not used

I/O ports in the 8051/CV-8052

- The Input/Output (I/O) pins are accessed using SFRs P0, P1, P2, P3. They are all bit addressable.
- To use the I/O as output pins configure them with the PxMOD register (not bit addressable). '1' makes the pin an output. For example to set P0.1 as output:

```
orl P0MOD, #00000010B
```

- Check the manual of the processor you are using to configure the pins:

Always check the manual!

For the STM32Fxx:

Table 24. GPIO register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOA_MODER	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

For the STM32Lxx:

Table 42. GPIO register map and reset values

Offset	Register name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOA_MODER	MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]		MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
	Reset value	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1

Always check the manual!

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

For the STM32Fxx:

These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

For the STM32Lxx:

Initialize pin PA0 as output in the STM32Fxx:

```
GPIOA->MODER |= 0x00000001;
```

Initialize pin PA0 as output in the STM32Lxx:

```
GPIOA->MODER = (GPIOA->MODER & 0xfffffffc) | 0x00000001;
```


Exercises

- A common way of passing parameters to a function is via the stack. Modify the function **WaitHalfSec** so that it receives the number of half-seconds to wait in the stack. (Note: this problem is not as trivial as it sounds. You may need to increment and/or decrement register SP to solve this problem)
- Most C programs pass parameters to functions via the stack. Also C programs use the stack to allocate automatic variables (local variables defined within the function). This works fine most of the time, but sometimes a condition commonly known as "stack overflow" occurs. Explain what causes "stack overflow".

Exercises

- Write an Interrupt service routine for timer 0 that generates a 1 kHz square wave in pin P0.0 of the CV-8052 processor.
- Write an interrupt service routine for timer 2 that increments a two digit BCD counter displayed in the 7-segment displays HEX1 and HEX0 of the CV-8052 every second. Make sure that the ISR for this question and the ISR from the previous question can run concurrently in the same processor.

Exercises

- Write a one second delay function using timer 1. This function will run in a CV-8052 with a 33.33MHz clock.
- Program profiling is used to find the usage of resources by a piece of code (a subroutine, for example). A profile value often needed is execution time. Show how to use timer 0 to find out the execution time of a subroutine.

Exercises

- From the examples given in this lecture, explain how to use the timer overflow flag to measure frequencies higher than 65535 Hz while using a 1-second time interval.

University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Lecture 18: Memory in the 8051 Microcontroller

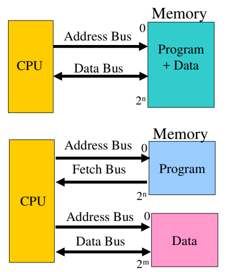
Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 30, 2022

Objectives

- Understand the differences and uses of bit, data, idata, xdata, and code memory in the 8051 microcontroller.
- Use the appropriate assembly instruction for each type of memory.
- Define variables using assembly code in the bit, data, idata, xdata, and code memory spaces.
- Use code memory to create look-up tables.

Microcontroller Architectures



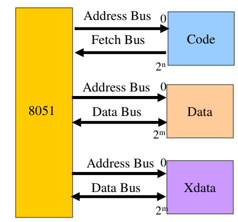
Von Neumann Architecture

Same assembly instruction can access code and data.

Harvard Architecture

Different assembly instructions to access code and data.

"Modern" 8051 Microcontroller Memory

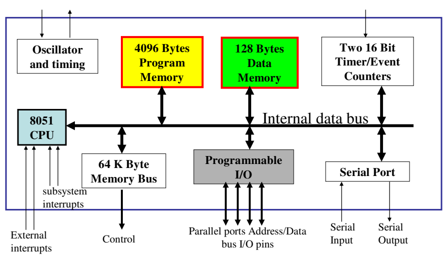


"ROM" memory: anywhere from 2k to 64k bytes

"RAM" memory: 128 or 256 bytes

"RAM" memory: 256 to 64k bytes

"Original" 8051 Microcontroller



ROM

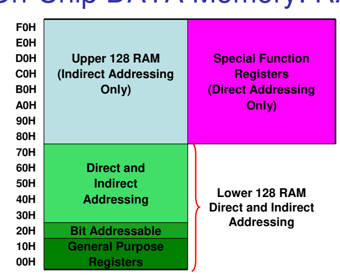
- ROM (Read Only Memory): can only be read by the microcontroller.
- Usually 'Flash'. Cheap, big, fast, and reliable! DO NOT BUY MICROCONTROLLERS WITHOUT BUILT-IN FLASH MEMORY.

RAM Types

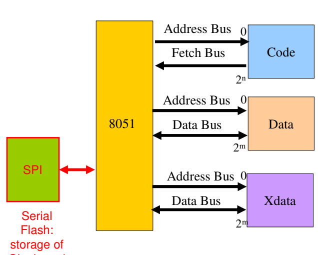
RAM (Random Access Memory):

- Dynamic: Ultra dense, but need to be refreshed periodically or the content will fade away. One memory cell is basically a capacitor! Not used very often with small/inexpensive microcontrollers.
- Static: Not as dense, but simpler to use. One memory cell is basically a flip-flop.

On-Chip DATA Memory: RAM



"Modern" 8051 Microcontroller Memory

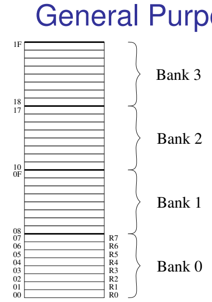


Flash

Fast RAM

RAM/EEPROM

General Purpose Registers



- There are efficient MOV instructions that target the registers directly:

  MOV R7, #55H ; takes 2 bytes

  MOV 07H, #55H ; takes 3 bytes

- To change the register bank in use set/reset the RS0 and RS1 bits in PSW:

RS1	RS0	Selects
0	0	Bank 0
0	1	Bank 1
1	0	Bank 2
1	1	Bank 3

Bit Addressable Memory

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

20h – 2Fh (16 locations X 8-bits = 128 bits)

Bit addressing:

setb 1Ah ; Two bytes

Byte addressing:

orl 23h, #04H ; Three bytes

Direct addressable memory

- If address < 80H: the target is general purpose RAM.
- If address ≥ 80H: the target is Special Function Registers

```
mov 30H, #0AAH ; Move number 0AAH to RAM 30H
mov 80H, #04H  ; Move number 04H to SFR 80H (P0)
mov P0, #04H   ; Move number 04H to SFR 80H (P0)
```


Bit Addressable SFRs

- If the address of a SFR ends in “0” or “8”, the bits of the SFR are bit addressable.

```
orl 80H, #04H ; Set bit 2 of SFR 80H (P0)
orl P0, #04H  ; Set bit 2 of SFR 80H (P0)
setb P0.2     ; Set bit 2 of SFR 80H (P0)
setb 82H      ; Set bit 2 of SFR 80H (P0)
orl DPH, #0xf ; DPH is at address 83H, not bit addressable
```


Indirectly addressable memory

- All the data memory in the 8051 is indirectly addressable.
- <u>The SFRs are not indirectly addressable.</u>
- To indirectly access memory we use the indexing registers R0 and R1.

Indirectly addressable memory

Write a program that clears (set to zero) 32 consecutive bytes from internal RAM starting at address 70H.

```
    mov R0, #70H ; Starting address we want to clear
    mov R7, #32  ; How many bytes we want to clear
L1:
    mov @R0, #0  ; Set memory pointed by R0 to zero
    inc R0       ; Point to the next byte
    djnz R7, L1  ; Repeat as many times as needed
```


Expanded RAM

- In the original 8051 expanded (or external) RAM was only available as additional memory chips. Up to 64k!
- Newer 8051’s may include some built in expanded RAM.
- Expanded RAM can only be accessed indirectly (via a pointer) using the MOVX instruction.
- The MOVX instruction uses either the DPTR (DPL, DPH), R0, or R1 as indexes.

Expanded RAM Access

Write a program that sets 50H bytes from expanded memory at address 80H to 255

```
    mov A, #0FFH      ; Value we want to set the memory to
    mov DPTR, #128    ; Starting address we want to set
    mov R0, #50H      ; How many bytes we want to set
X1:
    movx @DPTR, A     ; Set memory pointed by R0 to zero
    inc DPTR          ; Point to the next memory location
    djnz R0, X1       ; Repeat as many times as needed
```

Some 8051’s derivatives have two DPTR registers to simplify memory access. Check the data sheet for ‘dual data pointers’.

Remember: 80H=128, 0FFH=255

Code Memory

- Code memory can only be accessed indirectly using the MOVC instruction.
- The MOVC instruction uses either the DPTR (DPL, DPH), or the accumulator as indexes.
- Very useful when working with look-up tables!
- We can only READ from code memory.

Code Memory Access

Write a program to copy 8 bytes from code memory at address 1000H to data memory at address A0H

```
; Copy from code memory to data memory
    mov DPTR, #1000H    ; Source address in code memory
    mov R0, #A0H        ; Destination address in data memory
    mov R2, #08H        ; # of bytes we want to copy
C1:
    clr A
    movc A, @A+DPTR     ; Read from code memory
    mov @R0, A          ; Copy to data memory!
    inc DPTR            ; Point to next byte in code memory
    inc R0              ; Point to next byte in data memory
    djnz R2, C1         ; Repeat as many times as needed
```


Defining Variables in Assembly Source Code

- Use the segment directives (BSEG, DSEG, ISEG, XSEG, CSEG) to select bit, data, idata, xdata, or code memory.
- Use the origin directive override (AT) to set the beginning of the segment.
- USE the space allocation directives (DS, DBIT, DB, DW) to define the variables or their values.

Defining Variables in Assembly Source Code

```
; Defining bit variables
BSEG

ALARM: dbit 1
ONOFF: dbit 1

; Defining variables in data memory
DSEG at 30H

COUNT_HIGH: ds 1
COUNT_LOW:  ds 1
COUNT:      ds 2
BUFFER:     ds 16
```


Defining Variables in Assembly Source Code

```
; Defining xdata variables
XSEG at 80H

Sum:    ds 2
Adjust: ds 20

; Defining constants in code memory
CSEG at 1000H

Message: db 'Hello there…'
        db 0ah, 0dh ; ASCII code for nl/cr
        db 0
```


Lookup tables: some references

- Art of Assembly: http://webster.cs.ucr.edu/AoA/DOS/ch09/CH09-7.html
- Google: "Wikipedia lookup table"
- Google: "8051 lookup table"

Example: 7-segment conversion

```
; Look-up table for 7-seg displays.
T_7seg:
    DB 40H, 79H, 24H, 30H, 19H
    DB 12H, 02H, 78H, 00H, 10H
    DB 08H, 03H, 46H, 21H, 06H
    DB 0EH

Display_at mac
    mov dptr, #T_7seg
    ; Display Seconds
    mov a, %2
    anl a, #0x0f ; Force bits 4 to 7 to zero
    movc a, @dptr+a ; Read from table
    mov %0, a ; Display low nibble
    mov a, %2
    swap a ; exchange bits 0 to 3 with bits 4 to 7
    anl a, #0x0f ; Force bits 4 to 7 to zero
    movc a, @dptr+a ; ; Read from table
    mov %1, a ; Display high nibble
endmac
```


University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

I/O Ports in the 8051

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

April 8, 2020

Objectives

- Perform digital I/O (input/output) using the 8051.
- Use 7-segment displays to show numbers (again!).
- Read push buttons.
- Understand and solve the problem of contact bounce.
- Using the oscilloscope to test assembly code.
- I/O examples

Internal 8051 I/O ports

- The standard 8051 has **FOUR** 8-bit bidirectional I/O ports.
- To use a pin on the original 8051 as an input you **MUST** first write a ‘1’ to it.
- To use a pin on the CV-8052 as an output you MUST configure that pin as an output. (This is what most modern microcontrollers do!)
- All standard I/O pins are bit and byte addressable. For example:

```
setb P1.0
clr P2.7
mov P0, #0C0H
```

- All I/O pins may serve multiple functions. For example P3.4 can also be used as the input for Timer 0 (original 8051).

Configuring the Ports for Input/Output

- Newer 8051 variants have ports that can handle more current, but they need to be configured first.
- For example, to configure the ports in the CV-8052 soft processor we use SFRs P0MOD, P1MOD, P2MOD, and P3MOD:

```
mov P0MOD, #00000010b ; Make P0.1 output
mov P1MOD, #0ffH ; All pins of P1 are now outputs
mov P2MOD, #00001111b ; P2.0 to P2.3 outputs, P2.4 to P2.7 inputs
```

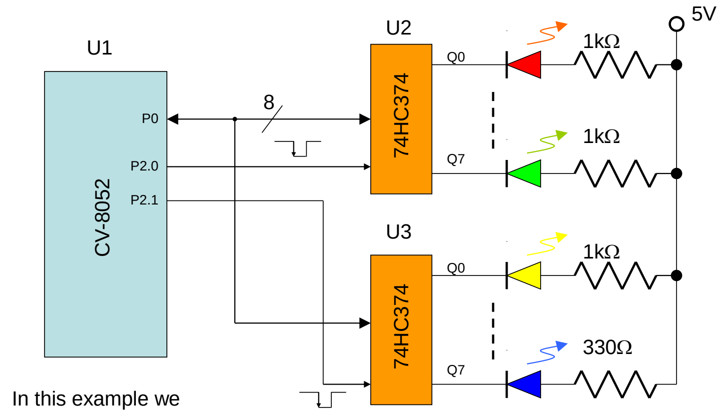

Port Connector in the CV-8052 using Terasic’s DE0-CV

	JP1				JP2		
LCD_DATA[0]	1	2	LCD_DATA[1]	P0.0	1	2	P0.1
LCD_DATA[2]	3	4	LCD_DATA[3]	P0.2	3	4	P0.3
LCD_DATA[4]	5	6	LCD_DATA[5]	P0.4	5	6	P0.5
LCD_DATA[6]	7	8	LCD_DATA[7]	P0.6	7	8	P0.7
LCD_EN	9	10	LCD_RS	P1.0	9	10	P1.1
5V	11	12	GND	5V	11	12	GND
LCD_RW	13	14	TXD	P1.2	13	14	P1.3
LCD_ON	15	16	RXD	P1.4	15	16	P1.5
FL_DQ[0]	17	18	FL_DQ[1]	P1.6	17	18	P1.7
FL_DQ[2]	19	20	FL_DQ[3]	P2.0	19	20	P2.1
FL_DQ[4]	21	22	FL_DQ[5]	P2.2	21	22	P2.3
FL_DQ[6]	23	24	FL_DQ[7]	P2.4	23	24	P2.5
FL_RST_N	25	26	FL_WE_N	P2.6	25	26	P2.7
FL_OE_N	27	28	FL_CE_N	P3.0	27	28	P3.1
3.3V	29	30	GND	3.3V	29	30	GND
TDO	31	32	TDI	P3.2	31	32	P3.3
TCS	33	34	TCK	P3.4	33	34	P3.5
Not used	35	36	Not used	P3.6	35	36	P3.7
T0	37	38	T1	INT0	37	38	INT1
T2	39	40	T2EX	Not Used	39	40	Not used

Expanding the Internal Ports

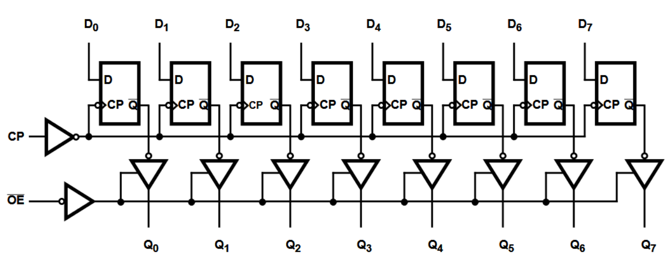
- If you need more digital I/O there are basically two options with modern microcontrollers:
  - Use SPI or I^{2}C peripherals. Works great and nowadays you can get interesting SPI or I^{2}C peripherals at very low cost!
  - Use digital logic to expand the available ports. Latches and shift registers seem to be the favorites.

Example1: Expanding the Internal Ports



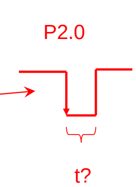
In this example we expand 10 outputs into 16 outputs!

74HC374



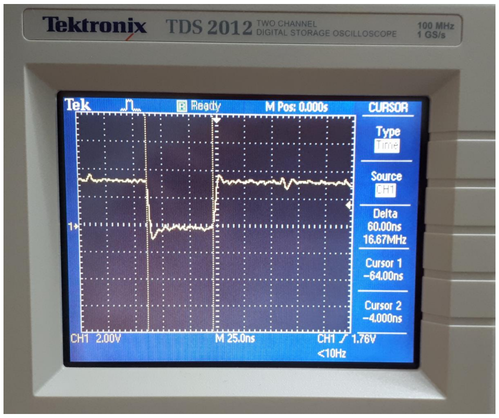
Expanding the Internal Ports

```
Write_to_Latches:
    ; Configure P0 and P2
    mov P0MOD, #0FFH ; All outs
    mov P2MOD, #03H ; P2.0, P2.1 out

    ; Write R2 to U2
    mov P0, R2
    ; Strobe the value into U2
    clr P2.0
    setb P2.0
    ; Write R3 to U3
    mov P0, R3
    ; Strobe the value into U3
    clr P2.1
    setb P2.1
    ret
```



f=33.33MHz, 1 clock per cycle, two cycle instruction: 60ns

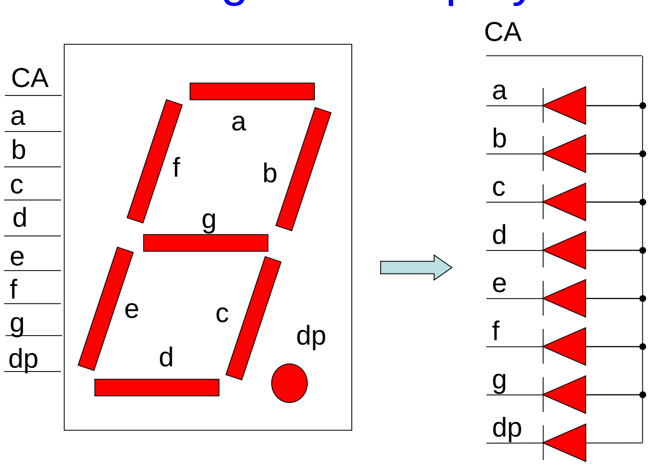
P2.0



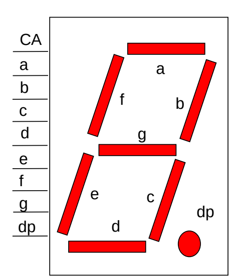
Example 2: Writing to Seven Segment LED displays

- Add two 7-segment LED displays to ports 0 and 3 of the CV-8052 microcontroller. The clock of the microcontroller has a frequency of 33.33MHz and it takes one clock period per machine cycle. Write a program to increment a counter approximately every second from 00 to 59 (in BCD). When the count reaches 60 reset it to zero and keep going.

7-Segment Display



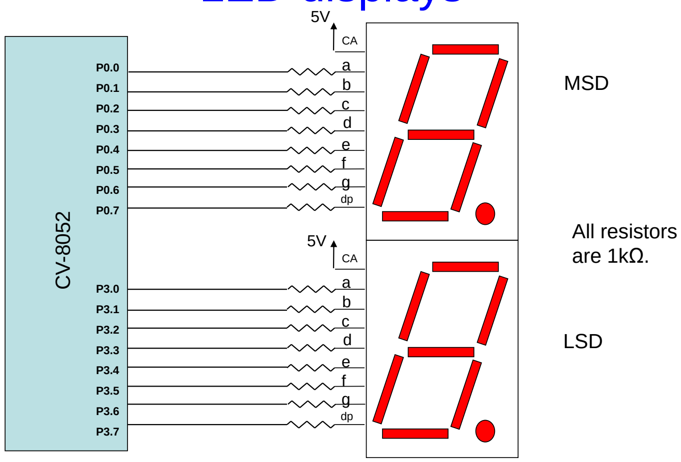
7-segment display look-up table



Zero is on!

#	dp	g	f	e	d	c	b	a	HEX
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									

Example 2: Writing to 7-Segment LED displays



Example 2: code

```
$MODDE0CV
org 0
   ljmp mycode

; Look-up table...
mytable:
    DB 0C0H, 0F9H, 0A4H, 0B0H, 099H
    DB 092H, 082H, 0F8H, 080H, 090H
Wait1Sec:
;33.33MHz, 1 clock per cycle: 0.030us
    mov R2, #180
L3: mov R1, #250
L2: mov R0, #250
L1: djnz R0, L1 ; 3 machine cycles-> 3*30ns*250=22.5us
    djnz R1, L2 ; 22.5us*250=5.625ms
    djnz R2, L3 ; 5.625ms*180=1s (approximately)
    ret
```


Example 2: code

```
mycode:
    mov SP, #7FH
    mov P0MOD, #0FFH
    mov P3MOD, #0FFH
    mov B, #0 ; counter
    mov dptr, #mytable
forever:
; Display MSB
    mov A, B
    swap A
    anl A, #0FH
    movc A, @A+dptr
    mov P0, A
; Display LSB
    mov A, B
    anl A, #0FH
    movc A, @A+dptr
    mov P3, A
```

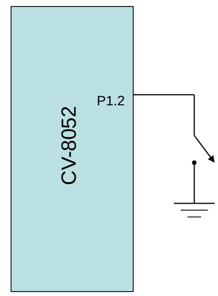

Example 2: code

```
; Do the one second delay
    lcall Wait1Sec
; Increment counter (register B) and convert to BCD
    mov A, B
    add A, #1
    da A
    mov B, A
; Check if count is 60
    cjne A, #060H, forever
    mov B, #0
    sjmp forever
END
```

Example 3: Reading Push Buttons

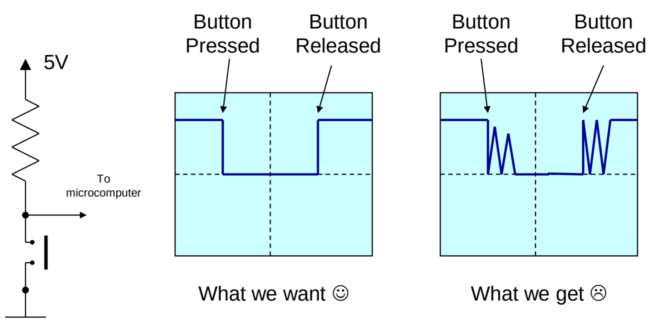
- Before using a pin for input we need to configure it:
  - Original 8051: Write ‘1’ to the pin to be used as input.
  - Newer 8051s: configure the pin as input using designated SFRs.
- In the original 8051 and almost all modern derivatives any pin can be used as output or input.
- In the 8051, pins in the same port can be independently used as inputs or outputs. For example pin P0.0 can be used as input, while P0.1 can be used as output!

Example 3: Reading Push Buttons




```
mov A, P1MOD
anl a, #11111011b ; P1.2 is input
mov P1MOD, a
.
.
.
.
jnb P1.2, ButtonPressed
jb P1.2, ButtonNotPressed
```

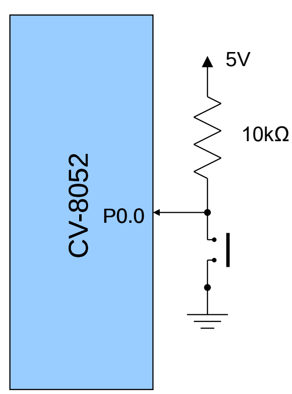

Problem: Contact Bounce



What we want ☺ What we get ☹

The time the contact bounces can be as long as 50ms!

Software De-bouncing

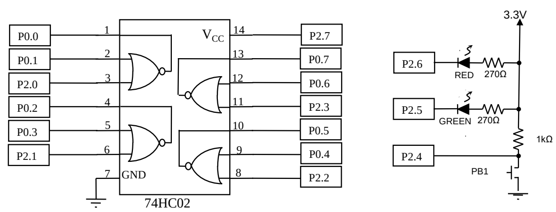



```
; After reset all ports are inputs
jb P0.0, not_pressed
lcall Wait50ms ; Wait and check again
jb P0.0, not_pressed
; Wait for the button to be released
L0: jnb P0.0, L0
sjmp pressed
```

This technique is called *wait-and-see* in many textbooks

Example 4: Logic IC testing.

- The circuit in the figure below can be used to verify that a 74HC02 integrated circuit (IC) operates correctly. The 74HC02 IC consists of four 2-input NOR gates. Write a subroutine for the CV-8052 processor that tests the IC after push button PB1 is pressed and then released (no de-bouncing needed). Additionally, the subroutine should: configure the input and output pins, apply power to the IC, test all possible input/output combinations and either turn the green LED on if the IC passes all the tests or the red LED on if the IC fails any test. The subroutine should set the power pin as well as all of the IC inputs to zero before returning so that the IC can be removed safely from the circuit after the tests are completed.



Other ICs that can be easily tested

- 74HC86
- 74HC32
- 74HC02
- 74HC00
- 74HC08
- 74HC04
- 74HC74
- 74HC73

14-pin ICs

- 74HC138
- 74HC139
- 74HC257
- 74HC259
- 74HC4051
- 74HC4052
- 74HC4053

16-pin ICs

Example 4: Logic IC testing.

- Pins that are outputs from the CV-8052, inputs to the IC:

  P0.1, P2.0, P0.3, P2.1, P2.2, P0.4, P2.3, P0.6, P2.5, P2.6, P2.7
- Pins that inputs to the CV-8052, outputs from the IC:

  P0.0, P0.2, P0.5, P0.7, P2.4

Example 4: Logic IC testing.

```
Test_NOR_IC:
    mov P0MOD, #01011010b
    mov P2MOD, #11101111b

    jb P2.4, $ ; Wait for PB1 press
    mov P2, #11100000B ; Power on, red LED off, green LED off
    jnb P2.4, $ ; Wait for PB1 release

    ; 0 nor 0 = 1
    clr P0.1
    clr P2.0
    jnb P0.0, error_NOR
    clr P0.3
    clr P2.1
    jnb P0.2, error_NOR
    clr P0.4
    clr P2.2
    jnb P0.5, error_NOR
    clr P2.3
    clr P0.6
    jnb P0.7, error_NOR
```

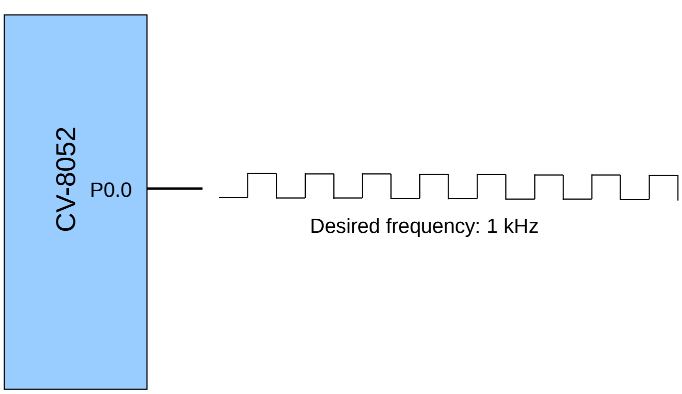
Example 4: Logic IC testing.

```
; 1 nor 0 = 0
setb P0.1
clr P2.0
jb P0.0, error_NOR
setb P0.3
clr P2.1
jb P0.2, error_NOR
setb P0.4
clr P2.2
jb P0.5, error_NOR
setb P2.3
clr P0.6
jb P0.7, error_NOR

; 0 nor 1 = 0
clr P0.1
setb P2.0
jb P0.0, error_NOR
clr P0.3
setb P2.1
jb P0.2, error_NOR
clr P0.4
setb P2.2
jb P0.5, error_NOR
clr P2.3
setb P0.6
jb P0.7, error_NOR
```


Example 4: Logic IC testing.

```
    ; 1 nor 1 = 0
    setb P0.1
    setb P2.0
    jb P0.0, error_NOR
    setb P0.3
    setb P2.1
    jb P0.2, error_NOR
    setb P0.4
    setb P2.2
    jb P0.5, error_NOR
    setb P2.3
    setb P0.6
    jb P0.7, error_NOR

    clr P2.5 ; Success, all tests pass
    sjmp done
error_NOR:
    clr P2.6 ; At least one test fails
done:
    anl P0, #10100101B ; Set all the IC inputs to zero
    anl P2, #11110000B ; Set all the IC inputs to zero
    clr P2.7 ; Turn power off
    ret
```


Example 5: Generating a square wave with Timer 0 ISR



Example 5: Generating a square wave with Timer 0 ISR

```
$MODDE0CV
org 0
    ljmp mycode

org 0bh
    ljmp Timer0_ISR

Init_Timer_0:
    anl TMOD, #0F0H  ; Mask the bits of timer 0
    orl TMOD, #01H   ; Mode 1
    clr TF0          ; Overflow flag=0
    mov TH0, #0xff   ; Interrupt immediately
    mov TL0, #0xff   ; Interrupt immediately
    setb ET0         ; Enable overflow interrupt
    setb EA          ; Enable global interrupts
    setb TR0         ; Start timer/counter 0
    ret
```

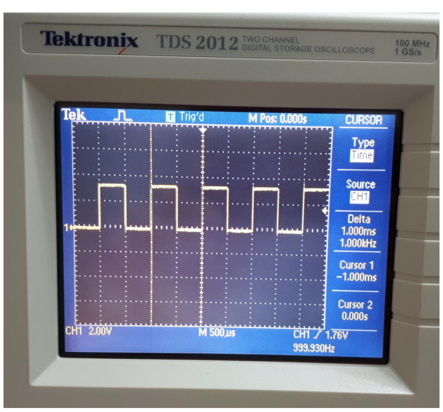
Example 5: Generating a square wave with Timer 0 ISR

```
CLK EQU 33333333
FREQU_OUT EQU 1000
TIMER_0_RELOAD EQU (65536-(CLK/(12*(FREQU_OUT*2))))

Timer0_ISR:
    clr TF0 ; Not needed for timer 0
    mov TL0, #low(TIMER_0_RELOAD)
    mov TH0, #high(TIMER_0_RELOAD)
    cpl P0.0
    reti
```


Example 5: Generating a square wave with Timer 0 ISR

```
mycode:
    mov SP, #7FH
    clr a
    mov LEDRA, a
    mov LEDRB, a

    mov P0MOD, #00000001B ; Make P0.0 output
    lcall Init_Timer_0

forever:
    sjmp forever

END
```


P0.0 output



Exercises

- The CV-8052 soft processor has six 7-segment displays accessible through SFRs HEX0 to HEX5. Write an assembly clock program that displays the time of day as hours, minutes, and seconds.
- Modify the program from the previous exercise so that the hour, minute, and second displays can be set using push buttons KEY1, KEY2, and KEY3. De-bounce the push buttons!
- Add two extra 7-segment displays to the CV-8052 connected to ports P0 and P1. Write a program that displays the 8 digits of your student number using P1, P0, HEX5, HEX4, HEX3, HEX2, HEX1, and HEX0.