CPE – ETI – Electronique
09/05/2022 09:39

3ETI

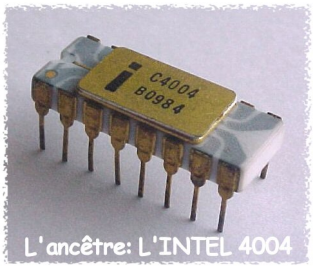
Architecture des systèmes à microprocesseur (S6)

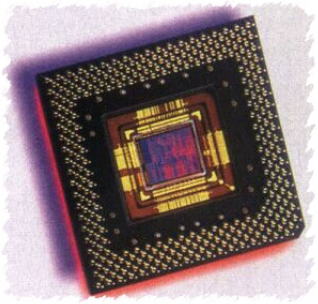
TRAVAUX PRATIQUES
ARCHITECTURE DES SYSTEMES A MICROPROCESSEUR

SILICON LABS

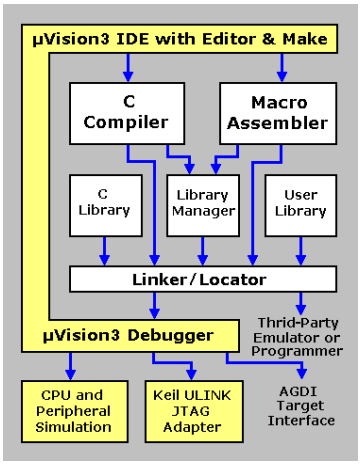
3ETI -- Semestre 6

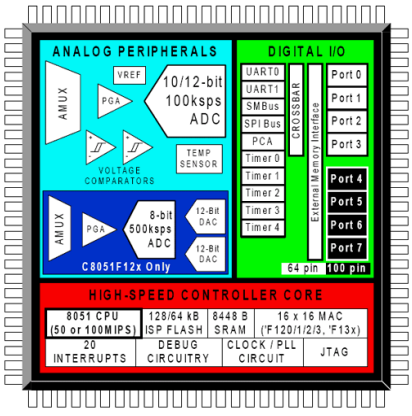
KEIL Tools by ARM
µVision®5
Integrated Development Environment
KEIL is a trademark and µVision is a registered trademark of ARM Ltd. All rights reserved.
This product is protected by US and international laws.





Documentation
Architecture 8051
et Microcontrôleur 8051F020
2022





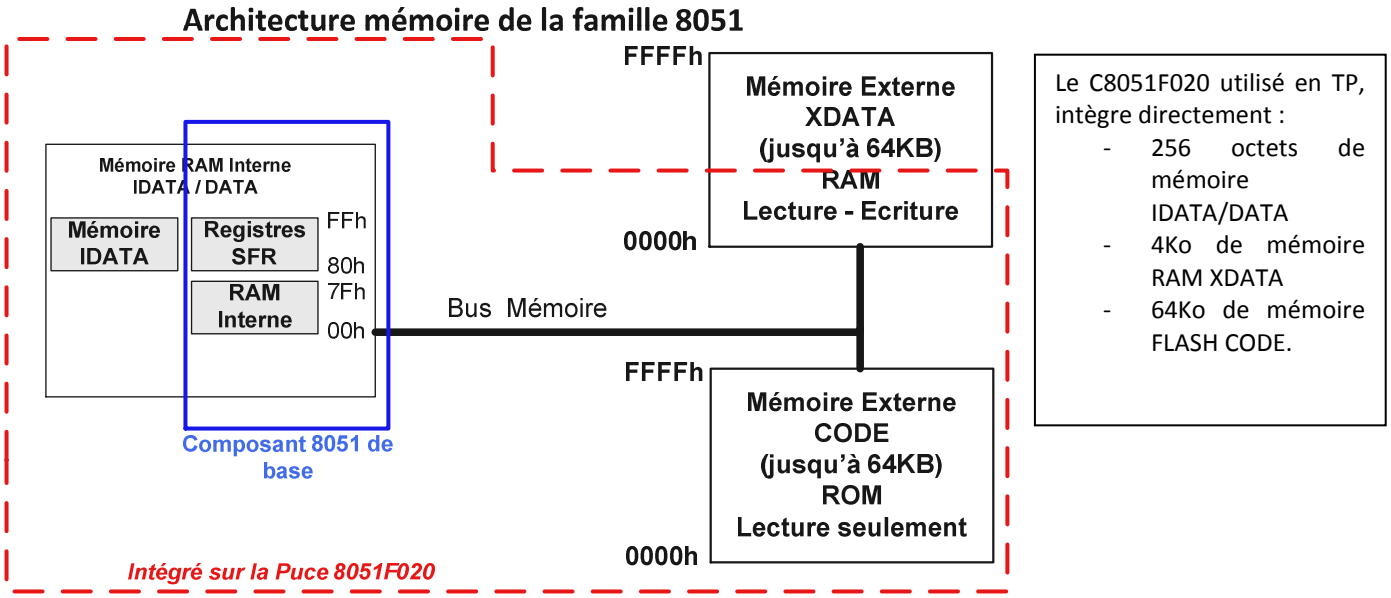
Nacer ABOUCHI - François JOLY

Département **ETI - IRC**
23/04/2018 14:30

PRESENTATION ARCHITECTURE 8051

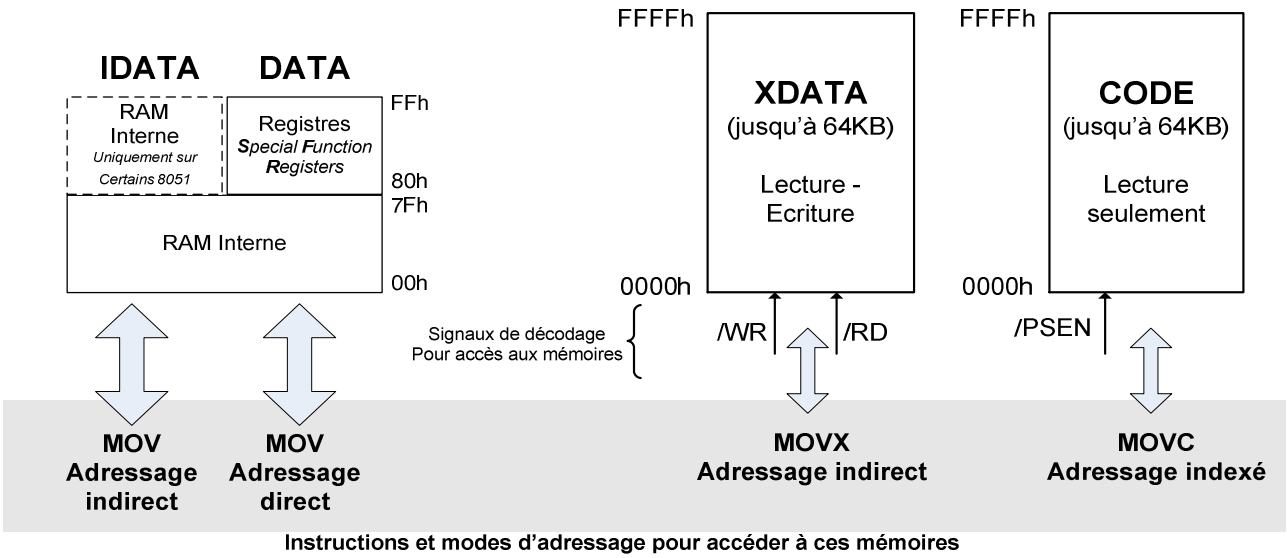
1 ARCHITECTURE 8051 – LES ESPACES MEMOIRE.

Un 8051 peut adresser 3 espaces mémoire distincts. Deux de ces espaces sont réservés au stockage des données, tandis que le troisième espace est réservé au code. Cette architecture est donc du type Harvard puisque les données et le code sont dans des espaces mémoire séparés.



Historiquement, les toutes premières versions de 8051, il y a une vingtaine d'années, n'intégraient sur leur puce que la zone mémoire DATA, d'une taille maximale de 256 octets, tandis que les espaces XDATA et CODE étaient gérés à l'extérieur de la puce 8051. De nos jours, du fait de l'intégration plus poussée sur le silicium, on peut trouver des composants de la famille 8051 qui intègrent la quasi-totalité des espaces mémoires sur la puce du processeur.
Par exemple, le microcontrôleur C8051F020 utilisé en TP, intègre directement 256 octets de mémoire DATA, 4Ko de mémoire RAM XDATA et 64Ko de mémoire FLASH CODE.

Espaces mémoire gérés par un 8051



Instructions et modes d'adressage pour accéder à ces mémoires

1.1 L'espace mémoire CODE.

D'une taille maximale de 64Ko, c'est l'espace affecté aux mémoires non volatiles telles que ROM, EPROM, FLASH EPROM.
Cet espace sert à stocker le code exécutable et les constantes.
Il ne peut qu'être lu, avec l'instruction MOVC en adressage indexé.
Quand il est placé à l'extérieur du 8051, il est décodé avec le signal /PSEN. Aujourd'hui, la quasi-totalité des composants de la famille 8051 intègrent de la mémoire CODE en interne.

1.2 L'espace mémoire XDATA.

D'une taille maximale de 64Ko, c'est l'espace affecté aux mémoires volatiles de type SRAM
Cet espace sert à stocker les données.
Il peut être lu et écrit, avec l'instruction MOVX en adressage indirect. A cause de ce type d'adressage, le temps d'accès est bien supérieur au temps d'accès à la mémoire DATA
Quand il est placé à l'extérieur du 8051, il est décodé avec les signaux /RD et /WR.

1.3 L'espace mémoire DATA

C'est un espace mémoire qui est toujours interne au 8051. Il est limité à 256 octets, organisés selon :
Les 128 premiers octets (de 00H à 07FH):

- 00h à 1Fh : 4 Bancs de registres.
- 20h à 2Fh : RAM adressable bit à bit.
- 30h à 7Fh : RAM d'usage général.

Les 128 octets suivants (de 80H à 0FFH):

- 80H à FFh en adressage direct : registres spéciaux, et registres pour périphériques (Special Function Registers)
- 80h à FFh en adressage indirect : RAM d'usage général (cette zone n'est pas disponible sur tous les composants 8051)

1.3.1 Les bancs de registre R0…R7

Le 8051 peut manipuler 8 registres d'usage général, nommés R0 à R7.
Ces registres peuvent être utilisés dans de nombreuses instructions du type :
MOV A,R0 ; Contenu de R0 copié dans l'accumulateur A – Adressage par registre

Or R0…R7 sont en fait localisés dans l'espace mémoire DATA, et peuvent occuper différents emplacements mémoire, selon le contenu des bits RS0 et RS1 du PSW selon

RS0	RS1	Numéro de Banc	Adresse physique de R0 dans l'espace DATA	Adresse physique de R7 dans l'espace DATA
0	0	Banc0	00h	07h
1	0	Banc1	08h	0Fh
0	1	Banc2	10h	17h
1	1	Banc3	18h	1Fh

Ainsi, pour reprendre l'exemple précédent, si RS0 et RS1 sont à 1, le banc 3 est sélectionné, alors l'instruction **MOV A,R0** est équivalente fonctionnellement à **MOV A,18h**, mais la première instruction sera codée sur un octet, contre 2 octets pour la seconde.
Cette commutation de bancs sera très pratique pour des changements rapides de contexte lors des appels d'interruption.

1.3.2 La zone mémoire adressable Bit à Bit

Une des particularités du 8051 est de permettre de manipuler des bits (contenu 0 ou 1), directement avec des instructions spécifiques. Ainsi chaque bit des cases mémoire 20h à 2Fh de la zone DATA est accessible individuellement. Une adresse « bit » comprise entre 00h et 7Fh lui est affectée. (voir tableau page suivante). Dans cette zone, il sera possible de faire des accès octets, mais aussi des accès bit.
Par exemple, le bit d'adresse 11h occupe en fait la position du bit b1 de la case d'adresse 22h de la ram interne DATA (si on numérote les bits de chaque octet de b0 à b7).
Ajoutons que les adresses bit de 80h à FFh permettent d'accéder à la configuration de certains bits dans les registres de périphériques (SFR registers).

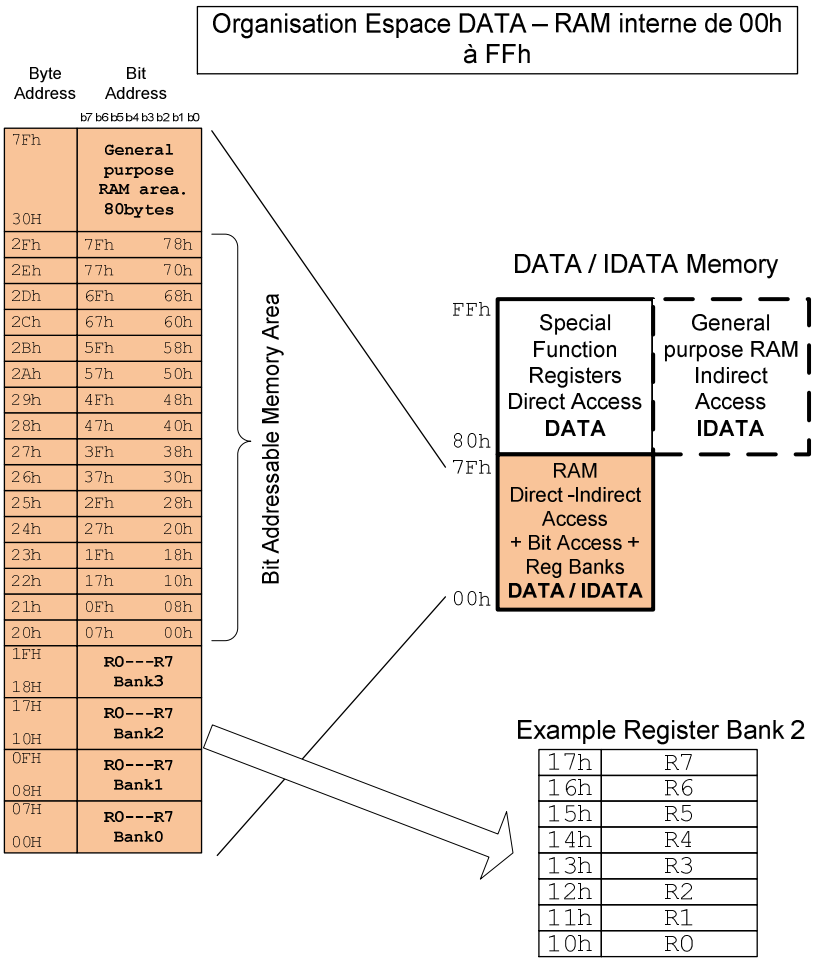
1.3.3 La RAM d'usage général

Elle couvre en fait l'ensemble de la zone 00h à 7Fh de l'espace DATA. Mais la zone 00h à 2Fh est aussi utilisée pour les bancs de registre et la zone adressable bit à bit. Cette zone est adressable en direct et en indirect. C'est ici que l'on y trouvera la zone mémoire dédiée à la pile.

A noter que cette zone mémoire interne au processeur est beaucoup rapide d'accès que la zone mémoire externe XDATA, aussi on privilégiera le stockage des données couramment utilisées dans cette zone.
Dans certaines versions de 8051, et c'est notamment le cas pour le 8051F020, la RAM interne s'étend de 80h à FFh. On y accède uniquement par adressage indirect (en adressage direct on adresse en fait les registres SFR).

1.3.4 Les registres SFR (Special fonction Registers)

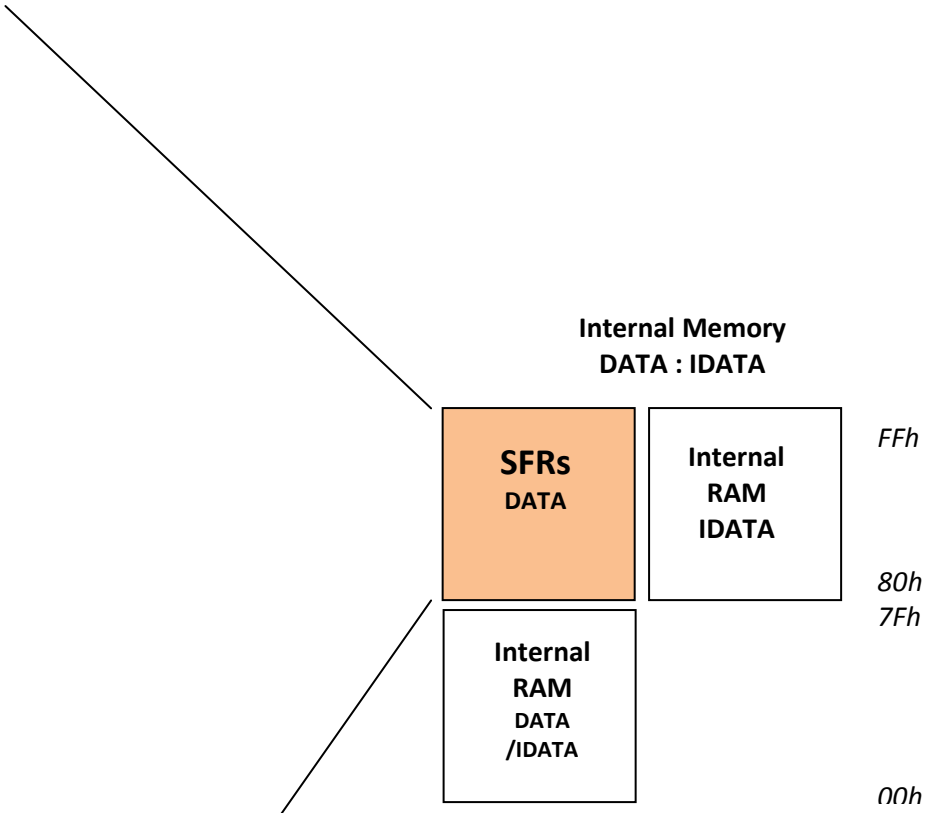
Ce sont des registres d'usage général, tels que l'accumulateur, le registre d'état PSW, le pointeur de pile SP, etc…. ou des registres utilisés pour la configuration des périphériques.
Ils sont accessibles en adressage direct dans la zone mémoire DATA 80h à FFh.
Attention les cases mémoire de la zone DATA 80h à FFh, en accès direct, non utilisées par des registres, ne sont pas accessibles.



Département **ETI - IRC**
23/04/2018 14:30

Organisation Espace DATA – RAM Interne 80 à FFh
Registres SFR

FFh		
F0h	B	*
E0h	A(accumulator)	*
D0h	PSW	*
B8h	IP	*
B0h	Port3(P3)	*
A8h	IE	*
A0h	Port2(P2)	*
99h	SBUF	
98h	SCON	*
90h	Port1(P1)	*
8Dh	TH1	
8Ch	TH0	
8Bh	TL1	
8Ah	TL0	
89h	TMOD	
88h	TCON	*
87h	PCON	
83h	DPH	
82h	DPL	
81h	SP	
80h	Port0(P0)	*



* indicates the SFR registers which are bit addressable

Attention : ce tableau correspond aux registres pour un 8051 basique. Le 8051F020 utilisé en TP, contient ces registres, mais aussi de nombreux autres registres requis pour assurer la configuration de ses périphériques spécifiques.

COEUR 8051F020

Ce chapitre constitué d'une partie de la fiche technique du 8051F020, permet d'apporter les informations suivantes :

- Description globale – Schéma bloc
- Jeu d'instructions – Temps d'exécution de ces instructions
- Organisation mémoire – Inventaire des différents espaces disponibles
- Descriptions des registres spécifiques, ACC, B, DPL, DPH, SP et PSW

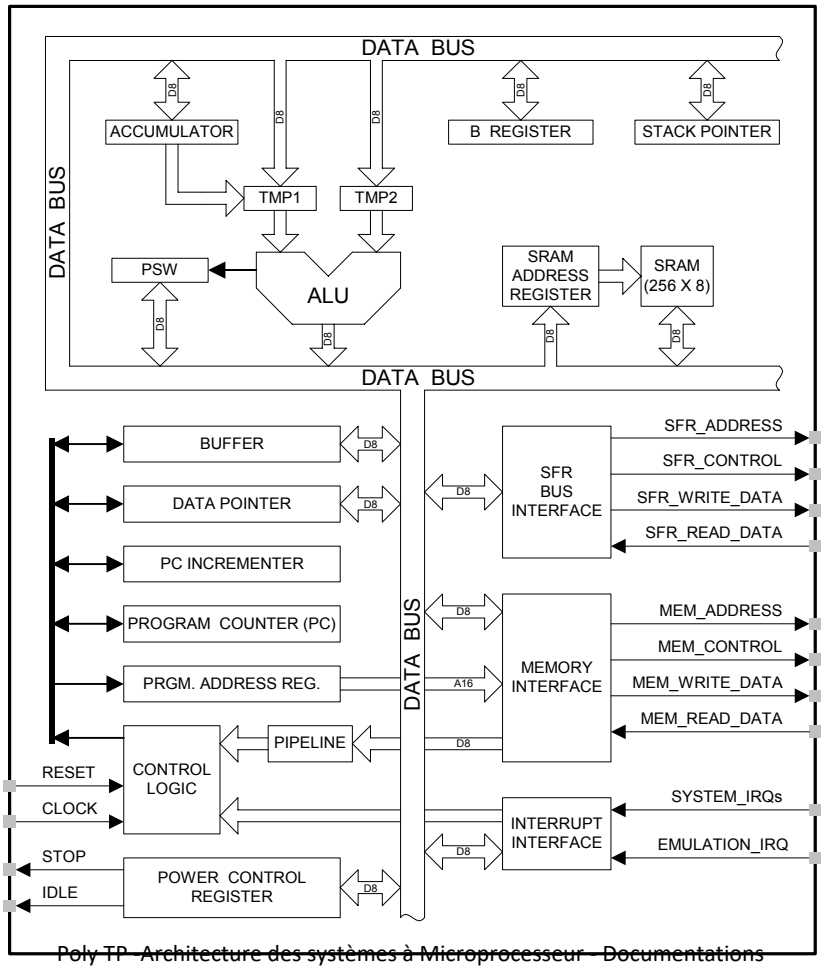
12. CIP-51 MICROCONTROLLER

The MCU system controller core is the CIP-51 microcontroller. The CIP-51 is fully compatible with the MCS-51™ instruction set; standard 803x/805x assemblers and compilers can be used to develop software. The MCU family has a superset of all the peripherals included with a standard 8051. Included are five 16-bit counter/timers (see description in **Section 22**), two full-duplex UARTs (see description in **Section 20** and **Section 21**), 256 bytes of internal RAM, 128 byte Special Function Register (SFR) address space (see **Section 12.2.6**), and 8/4 byte-wide I/O Ports (see description in **Section 17**). The CIP-51 also includes on-chip debug hardware (see description in **Section 24**), and interfaces directly with the MCUs' analog and digital subsystems providing a complete data acquisition or control-system solution in a single integrated circuit.

The CIP-51 Microcontroller core implements the standard 8051 organization and peripherals as well as additional custom peripherals and functions to extend its capability (see Figure 12.1 for a block diagram). The CIP-51 includes the following features:

- Fully Compatible with MCS-51 Instruction Set
- 25 MIPS Peak Throughput with 25 MHz Clock
- 0 to 25 MHz Clock Frequency
- 256 Bytes of Internal RAM
- 8/4 Byte-Wide I/O Ports
- Extended Interrupt Handler
- Reset Input
- Power Management Modes
- On-chip Debug Logic
- Program and Data Memory Security

Figure 12.1. CIP-51 Block Diagram



Performance
The CIP-51 employs a pipelined architecture that greatly increases its instruction throughput over the standard 8051 architecture. In a standard 8051, all instructions except for MUL and DIV take 12 or 24 system clock cycles to execute, and usually have a maximum system clock of 12 MHz. By contrast, the CIP-51 core executes 70% of its instructions in one or two system clock cycles, with no instructions taking more than eight system clock cycles.

With the CIP-51's maximum system clock at 25 MHz, it has a peak throughput of 25 MIPS. The CIP-51 has a total of 109 instructions. The table below shows the total number of instructions that require each execution time.

Clocks to Execute	1	2	2/3	3	3/4	4	4/5	5	8
Number of Instructions	26	50	5	14	7	3	1	2	1

Programming and Debugging Support
A JTAG-based serial interface is provided for in-system programming of the FLASH program memory and communication with on-chip debug support logic. The re-programmable FLASH can also be read and changed a single byte at a time by the application software using the MOVC and MOVX instructions. This feature allows program memory to be used for non-volatile data storage as well as updating program code under software control.

The on-chip debug support logic facilitates full speed in-circuit debugging, allowing the setting of hardware breakpoints and watch points, starting, stopping and single stepping through program execution (including interrupt service routines), examination of the program's call stack, and reading/writing the contents of registers and memory. This method of on-chip debug is completely non-intrusive and non-invasive, requiring no RAM, Stack, timers, or other on-chip resources.

The CIP-51 is supported by development tools from Silicon Labs and third party vendors. Silicon Labs provides an integrated development environment (IDE) including editor, macro assembler, debugger and programmer. The IDE's debugger and programmer interface to the CIP-51 via its JTAG interface to provide fast and efficient in-system device programming and debugging. Third party macro assemblers and C compilers are also available.

12.1. Instruction Set

The instruction set of the CIP-51 System Controller is fully compatible with the standard MCS-51™ instruction set; standard 8051 development tools can be used to develop software for the CIP-51. All CIP-51 instructions are the binary and functional equivalent of their MCS-51™ counterparts, including opcodes, addressing modes and effect on PSW flags. However, instruction timing is different than that of the standard 8051.

12.1.1. Instruction and CPU Timing

In many 8051 implementations, a distinction is made between machine cycles and clock cycles, with machine cycles varying from 2 to 12 clock cycles in length. However, the CIP-51 implementation is based solely on clock cycle timing. All instruction timings are specified in terms of clock cycles.

Due to the pipelined architecture of the CIP-51, most instructions execute in the same number of clock cycles as there are program bytes in the instruction. Conditional branch instructions take one less clock cycle to complete when the branch is not taken as opposed to when the branch is taken. Table 12.1 is the CIP-51 Instruction Set Summary, which includes the mnemonic, number of bytes, and number of clock cycles for each instruction.

12.1.2. MOVX Instruction and Program Memory

In the CIP-51, the MOVX instruction serves three purposes: accessing on-chip XRAM, accessing off-chip XRAM, and accessing on-chip program FLASH memory. The FLASH access feature provides a mechanism for user software to update program code and use the program memory space for non-volatile data storage (see **Section “15. FLASH**

MEMORY" on page 139). The External Memory Interface provides a fast access to off-chip XRAM (or memory-mapped peripherals) via the MOVX instruction. Refer to **Section "16. EXTERNAL DATA MEMORY INTERFACE AND ON-CHIP XRAM" on page 145** for details.

Table 12.1. CIP-51 Instruction Set Summary

Mnemonic	Description	Bytes	Clock Cycles
ARITHMETIC OPERATIONS			
ADD A, Rn	Add register to A	1	1
ADD A, direct	Add direct byte to A	2	2
ADD A, @Ri	Add indirect RAM to A	1	2
ADD A, #data	Add immediate to A	2	2
ADDC A, Rn	Add register to A with carry	1	1
ADDC A, direct	Add direct byte to A with carry	2	2
ADDC A, @Ri	Add indirect RAM to A with carry	1	2
ADDC A, #data	Add immediate to A with carry	2	2
SUBB A, Rn	Subtract register from A with borrow	1	1
SUBB A, direct	Subtract direct byte from A with borrow	2	2
SUBB A, @Ri	Subtract indirect RAM from A with borrow	1	2
SUBB A, #data	Subtract immediate from A with borrow	2	2
INC A	Increment A	1	1
INC Rn	Increment register	1	1
INC direct	Increment direct byte	2	2
INC @Ri	Increment indirect RAM	1	2
DEC A	Decrement A	1	1
DEC Rn	Decrement register	1	1
DEC direct	Decrement direct byte	2	2
DEC @Ri	Decrement indirect RAM	1	2
INC DPTR	Increment Data Pointer	1	1
MUL AB	Multiply A and B	1	4
DIV AB	Divide A by B	1	8
DA A	Decimal adjust A	1	1
LOGICAL OPERATIONS			
ANL A, Rn	AND Register to A	1	1
ANL A, direct	AND direct byte to A	2	2
ANL A, @Ri	AND indirect RAM to A	1	2
ANL A, #data	AND immediate to A	2	2
ANL direct, A	AND A to direct byte	2	2
ANL direct, #data	AND immediate to direct byte	3	3
ORL A, Rn	OR Register to A	1	1
ORL A, direct	OR direct byte to A	2	2
ORL A, @Ri	OR indirect RAM to A	1	2
ORL A, #data	OR immediate to A	2	2
ORL direct, A	OR A to direct byte	2	2
ORL direct, #data	OR immediate to direct byte	3	3
XRL A, Rn	Exclusive-OR Register to A	1	1
XRL A, direct	Exclusive-OR direct byte to A	2	2
XRL A, @Ri	Exclusive-OR indirect RAM to A	1	2

Table 12.1. CIP-51 Instruction Set Summary

Mnemonic	Description	Bytes	Clock Cycles
XRL A, #data	Exclusive-OR immediate to A	2	2
XRL direct, A	Exclusive-OR A to direct byte	2	2
XRL direct, #data	Exclusive-OR immediate to direct byte	3	3
CLR A	Clear A	1	1
CPL A	Complement A	1	1
RL A	Rotate A left	1	1
RLC A	Rotate A left through Carry	1	1
RR A	Rotate A right	1	1
RRC A	Rotate A right through Carry	1	1
SWAP A	Swap nibbles of A	1	1
DATA TRANSFER			
MOV A, Rn	Move Register to A	1	1
MOV A, direct	Move direct byte to A	2	2
MOV A, @Ri	Move indirect RAM to A	1	2
MOV A, #data	Move immediate to A	2	2
MOV Rn, A	Move A to Register	1	1
MOV Rn, direct	Move direct byte to Register	2	2
MOV Rn, #data	Move immediate to Register	2	2
MOV direct, A	Move A to direct byte	2	2
MOV direct, Rn	Move Register to direct byte	2	2
MOV direct, direct	Move direct byte to direct byte	3	3
MOV direct, @Ri	Move indirect RAM to direct byte	2	2
MOV direct, #data	Move immediate to direct byte	3	3
MOV @Ri, A	Move A to indirect RAM	1	2
MOV @Ri, direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate to indirect RAM	2	2
MOV DPTR, #data16	Load DPTR with 16-bit constant	3	3
MOVC A, @A+DPTR	Move code byte relative DPTR to A	1	3
MOVC A, @A+PC	Move code byte relative PC to A	1	3
MOVX A, @Ri	Move external data (8-bit address) to A	1	3
MOVX @Ri, A	Move A to external data (8-bit address)	1	3
MOVX A, @DPTR	Move external data (16-bit address) to A	1	3
MOVX @DPTR, A	Move A to external data (16-bit address)	1	3
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A, Rn	Exchange Register with A	1	1
XCH A, direct	Exchange direct byte with A	2	2
XCH A, @Ri	Exchange indirect RAM with A	1	2
XCHD A, @Ri	Exchange low nibble of indirect RAM with A	1	2
BOOLEAN MANIPULATION			
CLR C	Clear Carry	1	1
CLR bit	Clear direct bit	2	2
SETB C	Set Carry	1	1
SETB bit	Set direct bit	2	2
CPL C	Complement Carry	1	1

Table 12.1. CIP-51 Instruction Set Summary

Mnemonic	Description	Bytes	Clock Cycles
CPL bit	Complement direct bit	2	2
ANL C, bit	AND direct bit to Carry	2	2
ANL C, /bit	AND complement of direct bit to Carry	2	2
ORL C, bit	OR direct bit to carry	2	2
ORL C, /bit	OR complement of direct bit to Carry	2	2
MOV C, bit	Move direct bit to Carry	2	2
MOV bit, C	Move Carry to direct bit	2	2
JC rel	Jump if Carry is set	2	2/3
JNC rel	Jump if Carry is not set	2	2/3
JB bit, rel	Jump if direct bit is set	3	3/4
JNB bit, rel	Jump if direct bit is not set	3	3/4
JBC bit, rel	Jump if direct bit is set and clear bit	3	3/4
PROGRAM BRANCHING			
ACALL addr11	Absolute subroutine call	2	3
LCALL addr16	Long subroutine call	3	4
RET	Return from subroutine	1	5
RETI	Return from interrupt	1	5
AJMP addr11	Absolute jump	2	3
LJMP addr16	Long jump	3	4
SJMP rel	Short jump (relative address)	2	3
JMP @A+DPTR	Jump indirect relative to DPTR	1	3
JZ rel	Jump if A equals zero	2	2/3
JNZ rel	Jump if A does not equal zero	2	2/3
CJNE A, direct, rel	Compare direct byte to A and jump if not equal	3	3/4
CJNE A, #data, rel	Compare immediate to A and jump if not equal	3	3/4
CJNE Rn, #data, rel	Compare immediate to Register and jump if not equal	3	3/4
CJNE @Ri, #data, rel	Compare immediate to indirect and jump if not equal	3	4/5
DJNZ Rn, rel	Decrement Register and jump if not zero	2	2/3
DJNZ direct, rel	Decrement direct byte and jump if not zero	3	3/4
NOP	No operation	1	1

Notes on Registers, Operands and Addressing Modes:

Rn - Register R0-R7 of the currently selected register bank.

@Ri - Data RAM location addressed indirectly through R0 or R1.

rel - 8-bit, signed (two's complement) offset relative to the first byte of the following instruction. Used by SJMP and all conditional jumps.

direct - 8-bit internal data location's address. This could be a direct-access Data RAM location (0x00-0x7F) or an SFR (0x80-0xFF).

#data - 8-bit constant

#data16 - 16-bit constant

bit - Direct-accessed bit in Data RAM or SFR

addr11 - 11-bit destination address used by ACALL and AJMP. The destination must be within the same 2K-byte page of program memory as the first byte of the following instruction.

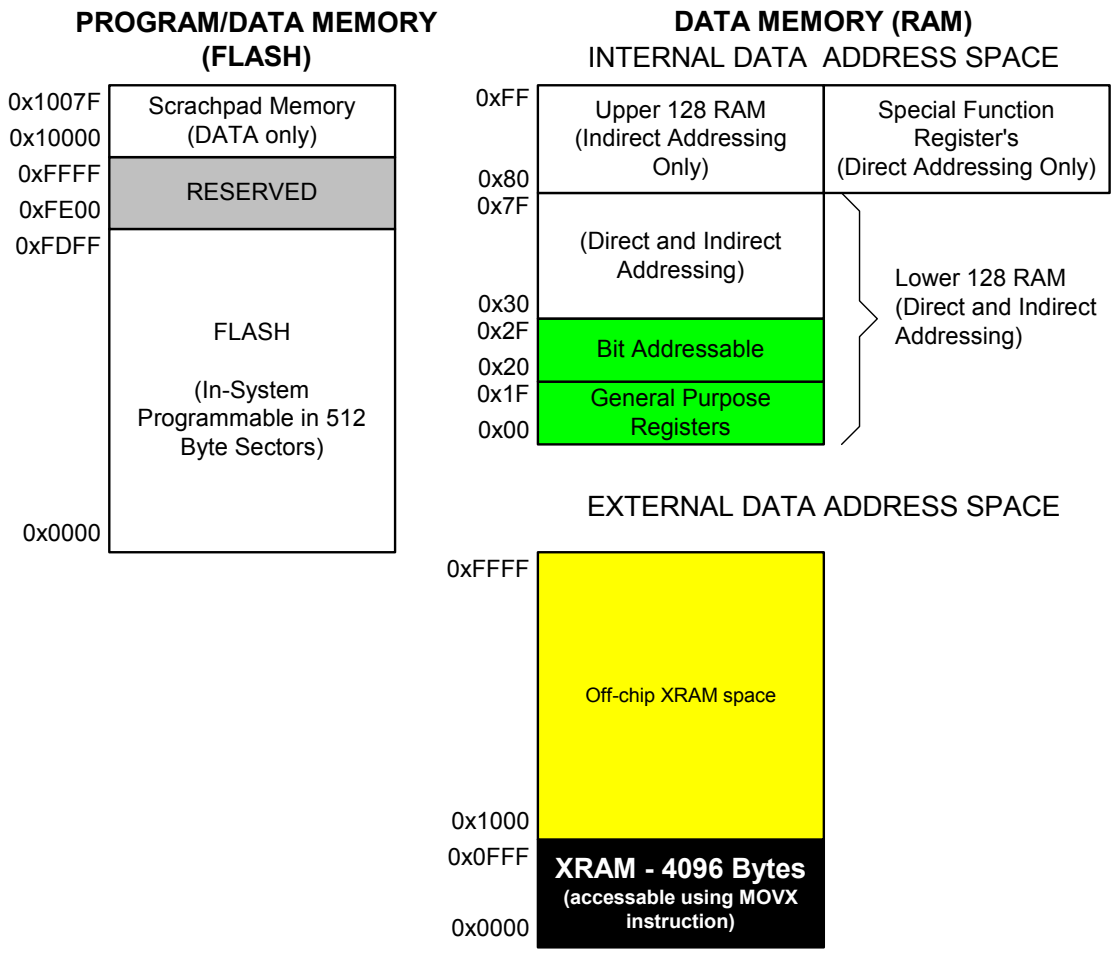
addr16 - 16-bit destination address used by LCALL and LJMP. The destination may be anywhere within the 64K-byte program memory space.

There is one unused opcode (0xA5) that performs the same function as NOP.

12.2. Memory Organization

The memory organization of the CIP-51 System Controller is similar to that of a standard 8051. There are two separate memory spaces: program memory and data memory. Program and data memory share the same address space but are accessed via different instruction types. There are 256 bytes of internal data memory and 64k bytes of internal program memory address space implemented within the CIP-51. The CIP-51 memory organization is shown in Figure 12.2.

Figure 12.2. Memory Map



12.2.1. Program Memory

The CIP-51 has a 64k byte program memory space. The MCU implements 65536 bytes of this program memory space as in-system re-programmed FLASH memory, organized in a contiguous block from addresses 0x0000 to 0xFFFF. Note: 512 bytes (0xEE00 to 0xFFFF) of this memory are reserved for factory use and are not available for user program storage.

Program memory is normally assumed to be read-only. However, the CIP-51 can write to program memory by setting the Program Store Write Enable bit (PSCTL.0) and using the MOVX instruction. This feature provides a mechanism for the CIP-51 to update program code and use the program memory space for non-volatile data storage. Refer to **Section “15. FLASH MEMORY” on page 139** for further details.

12.2.2. Data Memory

The CIP-51 implements 256 bytes of internal RAM mapped into the data memory space from 0x00 through 0xFF. The lower 128 bytes of data memory are used for general purpose registers and scratch pad memory. Either direct or indirect addressing may be used to access the lower 128 bytes of data memory. Locations 0x00 through 0x1F are addressable as four banks of general purpose registers, each bank consisting of eight byte-wide registers. The next 16 bytes, locations 0x20 through 0x2F, may either be addressed as bytes or as 128 bit locations accessible with the direct addressing mode.

The upper 128 bytes of data memory are accessible only by indirect addressing. This region occupies the same address space as the Special Function Registers (SFR) but is physically separate from the SFR space. The addressing mode used by an instruction when accessing locations above 0x7F determines whether the CPU accesses the upper 128 bytes of data memory space or the SFRs. Instructions that use direct addressing will access the SFR space. Instructions using indirect addressing above 0x7F access the upper 128 bytes of data memory. Figure 12.2 illustrates the data memory organization of the CIP-51.

12.2.3. General Purpose Registers

The lower 32 bytes of data memory, locations 0x00 through 0x1F, may be addressed as four banks of general-purpose registers. Each bank consists of eight byte-wide registers designated R0 through R7. Only one of these banks may be enabled at a time. Two bits in the program status word, RS0 (PSW.3) and RS1 (PSW.4), select the active register bank (see description of the PSW in Figure 12.6). This allows fast context switching when entering subroutines and interrupt service routines. Indirect addressing modes use registers R0 and R1 as index registers.

12.2.4. Bit Addressable Locations

In addition to direct access to data memory organized as bytes, the sixteen data memory locations at 0x20 through 0x2F are also accessible as 128 individually addressable bits. Each bit has a bit address from 0x00 to 0x7F. Bit 0 of the byte at 0x20 has bit address 0x00 while bit 7 of the byte at 0x20 has bit address 0x07. Bit 7 of the byte at 0x2F has bit address 0x7F. A bit access is distinguished from a full byte access by the type of instruction used (bit source or destination operands as opposed to a byte source or destination).

The MCS-51™ assembly language allows an alternate notation for bit addressing of the form XX.B where XX is the byte address and B is the bit position within the byte. For example, the instruction:

```
MOV    C, 22.3h
```

moves the Boolean value at 0x13 (bit 3 of the byte at location 0x22) into the Carry flag.

12.2.5. Stack

A programmer's stack can be located anywhere in the 256 byte data memory. The stack area is designated using the Stack Pointer (SP, address 0x81) SFR. The SP will point to the last location used. The next value pushed on the stack is placed at SP+1 and then SP is incremented. A reset initializes the stack pointer to location 0x07; therefore, the first value pushed on the stack is placed at location 0x08, which is also the first register (R0) of register bank 1. Thus, if more than one register bank is to be used, the SP should be initialized to a location in the data memory not being used for data storage. The stack depth can extend up to 256 bytes.

The MCUs also have built-in hardware for a stack record. The stack record is a 32-bit shift register, where each PUSH or increment SP pushes one record bit onto the register, and each CALL pushes two record bits onto the register. (A POP or decrement SP pops one record bit, and a RET pops two record bits, also.) The stack record circuitry can also detect an overflow or underflow on the 32-bit shift register, and can notify the debug software even with the MCU running at speed.

12.2.6. Special Function Registers

The direct-access data memory locations from 0x80 to 0xFF constitute the special function registers (SFRs). The SFRs provide control and data exchange with the CIP-51's resources and peripherals. The CIP-51 duplicates the SFRs found in a typical 8051 implementation as well as implementing additional SFRs used to configure and access the sub-systems unique to the MCU. This allows the addition of new functionality while retaining compatibility with the MCS-51™ instruction set. Table 12.2 lists the SFRs implemented in the CIP-51 System Controller.

The SFR registers are accessed anytime the direct addressing mode is used to access memory locations from 0x80 to 0xFF. SFRs with addresses ending in 0x0 or 0x8 (e.g. P0, TCON, P1, SCON, IE, etc.) are bit-addressable as well as byte-addressable. All other SFRs are byte-addressable only. Unoccupied addresses in the SFR space are reserved for future use. Accessing these areas will have an indeterminate effect and should be avoided. Refer to the corresponding pages of the datasheet, as indicated in Table 12.3, for a detailed description of each register.

Table 12.2. Special Function Register (SFR) Memory Map

F8	SPI0CN	PCA0H	PCA0CPH0	PCA0CPH1	PCA0CPH2	PCA0CPH3	PCA0CPH4	WDTCN
F0	B	SCON1	SBUF1	SADDR1	TL4	TH4	EIP1	EIP2
E8	ADC0CN	PCA0L	PCA0CPL0	PCA0CPL1	PCA0CPL2	PCA0CPL3	PCA0CPL4	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	RCAP4L	RCAP4H	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	
D0	PSW	REF0CN	DAC0L	DAC0H	DAC0CN	DAC1L	DAC1H	DAC1CN
C8	T2CON	T4CON	RCAP2L	RCAP2H	TL2	TH2		SMB0CR
C0	SMB0CN	SMB0STA	SMB0DAT	SMB0ADR	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH
B8	IP	SADEN0	AMX0CF	AMX0SL	ADC0CF	P1MDIN	ADC0L	ADC0H
B0	P3	OSCXCN	OSCICN			P74OUT†	FLSCL	FLACL
A8	IE	SADDR0	ADC1CN	ADC1CF	AMX1SL	P3IF	SADEN1	EMI0CN
A0	P2	EMI0TC		EMI0CF	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	SPI0CFG	SPI0DAT	ADC1	SPI0CKR	CPT0CN	CPT1CN
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	P7†	
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	P4†	P5†	P6†	PCON
	0(8) (bit addressable)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)

12.2.7. Register Descriptions

Following are descriptions of SFRs related to the operation of the CIP-51 System Controller. Reserved bits should not be set to logic l. Future product versions may use these bits to implement new features in which case the reset value of the bit will be logic 0, selecting the feature's default state. Detailed descriptions of the remaining SFRs are included in the sections of the datasheet associated with their corresponding system function.

Figure 12.3. SP: Stack Pointer

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000111
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0x81

Bits7-0: SP: Stack Pointer.
The Stack Pointer holds the location of the top of the stack. The stack pointer is incremented before every PUSH operation. The SP register defaults to 0x07 after reset.

Figure 12.4. DPL: Data Pointer Low Byte

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0x82

Bits7-0: DPL: Data Pointer Low.
The DPL register is the low byte of the 16-bit DPTR. DPTR is used to access indirectly addressed XRAM and FLASH memory.

Figure 12.5. DPH: Data Pointer High Byte

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0x83

Bits7-0: DPH: Data Pointer High.
The DPH register is the high byte of the 16-bit DPTR. DPTR is used to access indirectly addressed XRAM and FLASH memory.

Figure 12.6. PSW: Program Status Word

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	Reset Value
CY	AC	F0	RS1	RS0	OV	F1	PARITY	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0xD0

Bit7: CY: Carry Flag.
This bit is set when the last arithmetic operation resulted in a carry (addition) or a borrow (subtraction). It is cleared to 0 by all other arithmetic operations.

Bit6: AC: Auxiliary Carry Flag
This bit is set when the last arithmetic operation resulted in a carry into (addition) or a borrow from (subtraction) the high order nibble. It is cleared to 0 by all other arithmetic operations.

Bit5: F0: User Flag 0.
This is a bit-addressable, general purpose flag for use under software control.

Bits4-3: RS1-RS0: Register Bank Select.
These bits select which register bank is used during register accesses.

RS1	RS0	Register Bank	Address
0	0	0	0x00 - 0x07
0	1	1	0x08 - 0x0F
1	0	2	0x10 - 0x17
1	1	3	0x18 - 0x1F

Bit2: OV: Overflow Flag.
This bit is set to 1 if the last arithmetic operation resulted in a carry (addition), borrow (subtraction), or overflow (multiply or divide). It is cleared to 0 by all other arithmetic operations.

Bit1: F1: User Flag 1.
This is a bit-addressable, general purpose flag for use under software control.

Bit0: PARITY: Parity Flag.
This bit is set to 1 if the sum of the eight bits in the accumulator is odd and cleared if the sum is even.

Figure 12.7. ACC: Accumulator

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0xE0

Bits7-0: ACC: Accumulator.
This register is the accumulator for arithmetic operations.

Figure 12.8. B: B Register

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0xF0

Bits7-0: B: B Register.
This register serves as a second accumulator for certain arithmetic operations.

JEU D'INSTRUCTIONS 8051

Ce chapitre décrit en détail toutes les instructions assembleur utilisables sur les microcontrôleurs de la famille 8051.
Il y est décrit notamment :

- Les différents modes d'adressage
- Les différents types d'instruction
- Le descriptif détaillé des instructions
- Les codes hexadécimaux de chaque instruction

8051 Instruction Set

8051 Instruction Set

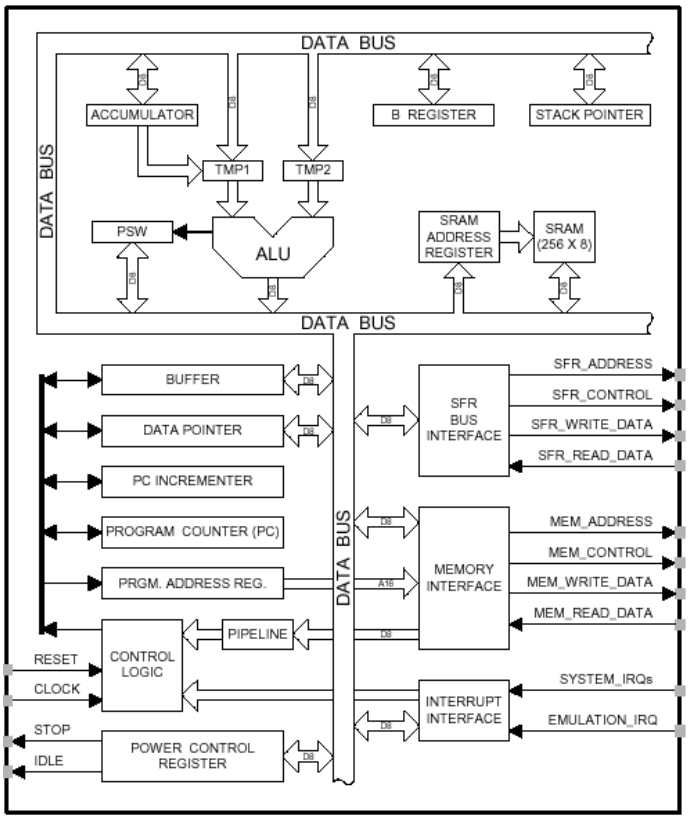
- Introduction
- CIP-51 architecture and memory organization review
- Addressing Modes
  - Register addressing
  - Direct addressing
  - Indirect addressing
  - Immediate constant addressing
  - Relative addressing
  - Absolute addressing
  - Long addressing
  - Indexed addressing
- Instruction Types
  - Arithmetic operations
  - Logical operations
  - Data transfer instructions
  - Boolean variable instructions
  - Program branching instructions

Introduction

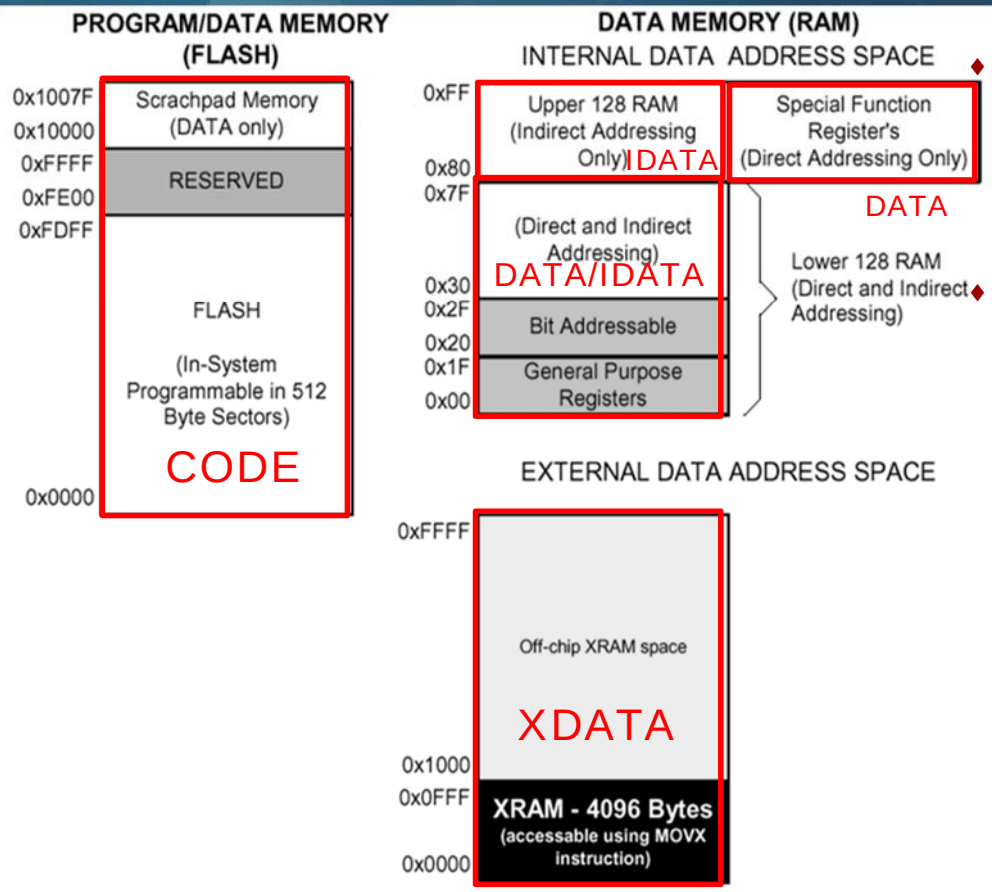
- A computer instruction is made up of an operation code (op-code) followed by either zero, one or two bytes of operands
- The op-code identifies the type of operation to be performed while the operands identify the source and destination of the data
- The operand can be:
  - The data value itself
  - A CPU register
  - A memory location
  - An I/O port
- If the instruction is associated with more than one operand, the format is always:

  Instruction Destination, Source

CIP-51 Architecture Review



Memory Organization

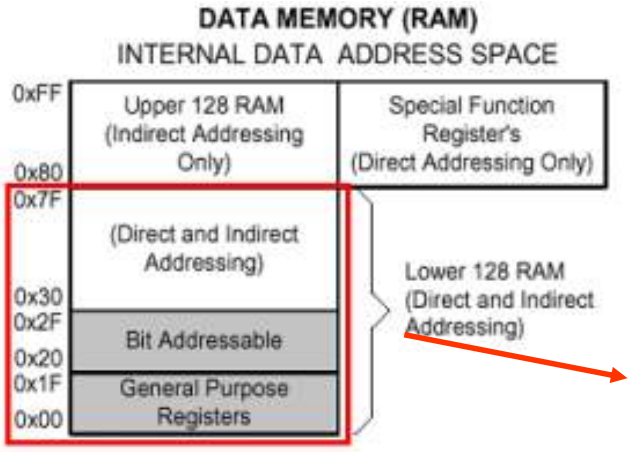


The memory organization of C8051F020 is similar to that of a standard 8051

Program and data memory share the same address space but are accessed via different instruction types

SILICON LABS

Internal Data Memory

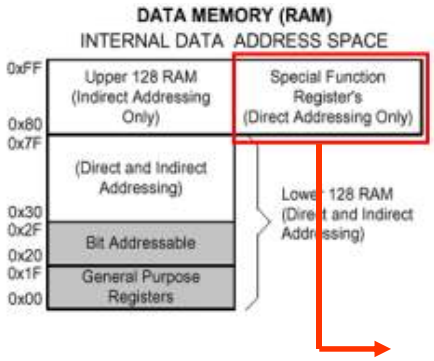


Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

(Byte Address / Bit Address)

SILICON LABS

Special Function Registers



F8	SPI0CN	PCA0H	PCA0CPH0	PCA0CPH1	PCA0CPH2	PCA0CPH3	PCA0CPH4	WDTCN
F0	B	SCON1	SBUF1	SADDR1	TL4	TH4	EIP1	EIP2
E8	ADC0CN	PCA0L	PCA0CPL0	PCA0CPL1	PCA0CPL2	PCA0CPL3	PCA0CPL4	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	RCAP4L	RCAP4H	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0M0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	
D0	PSW	REF0CN	DAC0L	DAC0H	DAC0CN	DAC1L	DAC1H	DAC1CN
C8	T2CON	T4CON	RCAP2L	RCAP2H	TL2	TH2		SMB0CR
C0	SMB0CN	SMB0STA	SMB0DAT	SMB0ADR	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH
B8	IP	SADEN0	AMX0CF	AMX0SL	ADC0CF	P1MDIN	ADC0L	ADC0H
B0	P3	OSCXCN	OSCICN			P74OUT	FLSCL	FLACL
A8	IE	SADDR0	ADC1CN	ADC1CF	AMX1SL	P3IF	SADEN1	EMI0CN
A0	P2	EMI0TC		EMI0CF	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	SPI0CFG	SPIODAT	ADC1	SPI0CKR	CPT0CN	CPT1CN
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	P7	
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	P4	P5	P6	PCON
	0(8) Bit addressable	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)

7

Addressing Modes

- Eight modes of addressing are available with the C8051F020
- The different addressing modes determine how the operand byte is selected

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

* Related to program branching instructions

8

Register Addressing

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

```
MOV     R0, A
```

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

Direct Addressing

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

Note: Abbreviated SFR names are defined in the “C8051F020.inc” header file

- *Example*:

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

Indirect Addressing

- This mode uses a pointer to hold the effective address of the operand
- Only registers R0, R1 and DPTR can be used as the pointer registers
- The R0 and R1 registers can hold an 8-bit address, whereas DPTR can hold a 16-bit address

- ***Examples*:**

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

MOVX   A,@DPTR    ;Transfer the contents from
                  ;the memory location
                  ;pointed to by DPTR into the
                  ;accumulator. DPTR could have a
                  ;16-bit address, such as 1234H.
```

11

Immediate Constant Addressing

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

- *Examples*:

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

MOV DPTR,#0FE00H  ;Move 16-bit data constant
                  ;FE00H into the 16-bit Data
                  ;Pointer Register.
```

12

Relative Addressing

- This mode of addressing is used with some type of jump instructions, like SJMP (short jump) and conditional jumps like JNZ
- These instructions transfer control from one part of a program to another
- The destination address must be within -128 and +127 bytes from the current instruction address because an 8-bit offset is used ($2^8 = 256$)
- *Example*:

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

Absolute Addressing

- Two instructions associated with this mode of addressing are ACALL and AJMP instructions
- These are 2-byte instructions where the 11-bit absolute address is specified as the operand
- The upper 5 bits of the 16-bit PC address are not modified. The lower 11 bits are loaded from this instruction. So, the branch address must be within the current 2K byte page of program memory ($2^{11} = 2048$)
- *Example*:

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

PORT_INIT: MOV    P0, #0FH     ;PORT_INIT subroutine
```

Long Addressing

- This mode of addressing is used with the LCALL and LJMP instructions
- It is a 3-byte instruction and the last 2 bytes specify a 16-bit destination location where the program branches
- It allows use of the full 64 K code space
- The program will always branch to the same location no matter where the program was previously
- *Example*:

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

15

Indexed Addressing

- The Indexed addressing is useful when there is a need to retrieve data from a look-up table
- A 16-bit register (data pointer) holds the base address and the accumulator holds an 8-bit displacement or index value
- The sum of these two registers forms the effective address for a JMP or MOVC instruction
- *Example*:

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

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

16

Instruction Types

- The C8051F020 instructions are divided into five functional groups:
  - Arithmetic operations
  - Logical operations
  - Data transfer operations
  - Boolean variable operations
  - Program branching operations

Arithmetic Operations

- With arithmetic instructions, the C8051F020 CPU has no special knowledge of the data format (e.g. signed binary, unsigned binary, binary coded decimal, ASCII, etc.)
- The appropriate status bits in the PSW are set when specific conditions are met, which allows the user software to manage the different data formats
  - [@Ri] implies contents of memory location pointed to by R0 or R1
  - Rn refers to registers R0-R7 of the currently selected register bank

Mnemonic	Description
ADD A, Rn	A = A + [Rn]
ADD A, direct	A = A + [direct memory]
ADD A,@Ri	A = A + [memory pointed to by Ri]
ADD A,#data	A = A + immediate data
ADDC A,Rn	A = A + [Rn] + CY
ADDC A, direct	A = A + [direct memory] + CY
ADDC A,@Ri	A = A + [memory pointed to by Ri] + CY
ADDC A,#data	A = A + immediate data + CY
SUBB A,Rn	A = A - [Rn] - CY
SUBB A, direct	A = A - [direct memory] - CY
SUBB A,@Ri	A = A - [@Ri] - CY
SUBB A,#data	A = A - immediate data - CY
INC A	A = A + 1
INC Rn	[Rn] = [Rn] + 1
INC direct	[direct] = [direct] + 1
INC @Ri	[@Ri] = [@Ri] + 1
DEC A	A = A - 1
DEC Rn	[Rn] = [Rn] - 1
DEC direct	[direct] = [direct] - 1
DEC @Ri	[@Ri] = [@Ri] - 1
MUL AB	Multiply A & B
DIV AB	Divide A by B
DA A	Decimal adjust A

Logical Operations

- Logical instructions perform Boolean operations (AND, OR, XOR, and NOT) on data bytes on a *bit-by-bit* basis
- Examples:

```
ANLA, #02H  ;Mask bit 1
ORLTCON, A  ;TCON=TCON-OR-A
```

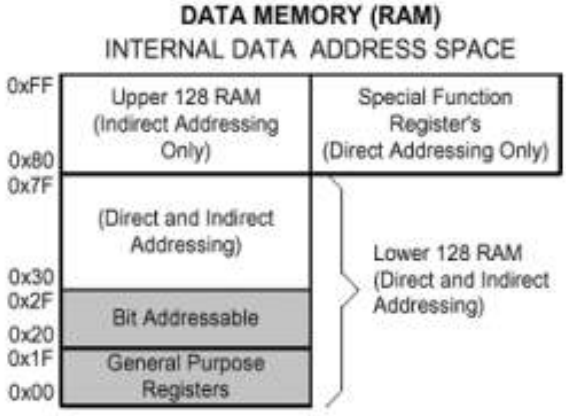
Mnemonic	Description
ANL A, Rn	A = A & [Rn]
ANL A, direct	A = A & [direct memory]
ANL A,@Ri	A = A & [memory pointed to by Ri]
ANL A,#data	A= A & immediate data
ANL direct,A	[direct] = [direct] & A
ANL direct,#data	[direct] = [direct] & immediate data
ORL A, Rn	A = A OR [Rn]
ORL A, direct	A = A OR [direct]
ORL A,@Ri	A = A OR [@RI]
ORL A,#data	A = A OR immediate data
ORL direct,A	[direct] = [direct] OR A
ORL direct,#data	[direct] = [direct] OR immediate data
XRL A, Rn	A = A XOR [Rn]
XRL A, direct	A = A XOR [direct memory]
XRL A,@Ri	A = A XOR [@Ri]
XRL A,#data	A = A XOR immediate data
XRL direct,A	[direct] = [direct] XOR A
XRL direct,#data	[direct] = [direct] XOR immediate data
CLR A	Clear A
CPL A	Complement A
RL A	Rotate A left
RLC A	Rotate A left (through C)
RR A	Rotate A right
RRC A	Rotate A right (through C)
SWAP A	Swap nibbles

SILICON LABS

19

Data Transfer Instructions

- Data transfer instructions can be used to transfer data between an internal RAM location and an SFR location without going through the accumulator
- It is also possible to transfer data between the internal and external RAM by using indirect addressing
- The upper 128 bytes of data RAM are accessed only by indirect addressing and the SFRs are accessed only by direct addressing



Mnemonic	Description
MOV @Ri, direct	[@Ri] = [direct]
MOV @Ri, #data	[@Ri] = immediate data
MOV DPTR, #data 16	[DPTR] = immediate data
MOVC A,@A+DPTR	A = Code byte from [@A+DPTR]
MOVC A,@A+PC	A = Code byte from [@A+PC]
MOVX A,@Ri	A = Data byte from external ram [@Ri]
MOVX A,@DPTR	A = Data byte from external ram [@DPTR]
MOVX @Ri, A	External[@Ri] = A
MOVX @DPTR,A	External[@DPTR] = A
PUSH direct	Push into stack
POP direct	Pop from stack
XCH A,Rn	A = [Rn], [Rn] = A
XCH A, direct	A = [direct], [direct] = A
XCH A, @Ri	A = [@Rn], [@Rn] = A
XCHD A,@Ri	Exchange low order digits

SILICON LABS

20

Boolean Variable Instructions

- The C8051F020 processor can perform single bit operations
- The operations include *set, clear, and, or* and *complement* instructions
- Also included are bit–level moves or conditional jump instructions
- All bit accesses use direct addressing

- **Examples:**

```
SETB  TR0    ;Start Timer0.
POLL: JNB    TR0, POLL  ;Wait
   till timer overflows.
```

Mnemonic	Description
CLR C	Clear C
CLR bit	Clear direct bit
SETB C	Set C
SETB bit	Set direct bit
CPL C	Complement c
CPL bit	Complement direct bit
ANL C,bit	AND bit with C
ANL C,/bit	AND NOT bit with C
ORL C,bit	OR bit with C
ORL C,/bit	OR NOT bit with C
MOV C,bit	MOV bit to C
MOV bit,C	MOV C to bit
JC rel	Jump if C set
JNC rel	Jump if C not set
JB bit,rel	Jump if specified bit set
JNB bit,rel	Jump if specified bit not set
JBC bit,rel	if specified bit set then clear it and jump

21

Program Branching Instructions

- Program branching instructions are used to control the flow of program execution

- Some instructions provide decision making capabilities before transferring control to other parts of the program (conditional branches).

Mnemonic	Description
ACALL addr11	Absolute subroutine call
LCALL addr16	Long subroutine call
RET	Return from subroutine
RETI	Return from interrupt
AJMP addr11	Absolute jump
LJMP addr16	Long jump
SJMP rel	Short jump
JMP @A+DPTR	Jump indirect
JZ rel	Jump if A=0
JNZ rel	Jump if A NOT=0
CJNE A,direct,rel	Compare and Jump if Not Equal
CJNE A,#data,rel	
CJNE Rn,#data,rel	
CJNE @Ri,#data,rel	
DJNZ Rn,rel	Decrement and Jump if Not Zero
DJNZ direct,rel	
NOP	No Operation

22

Appendix

8051 Instruction

Arithmetic Operations

- [@Ri] implies contents of memory location pointed to by R0 or R1
- Rn refers to registers R0-R7 of the currently selected register bank

Mnemonic	Description
ADD A, Rn	A = A + [Rn]
ADD A, direct	A = A + [direct memory]
ADD A,@Ri	A = A + [memory pointed to by Ri]
ADD A,#data	A = A + immediate data
ADDC A,Rn	A = A + [Rn] + CY
ADDC A, direct	A = A + [direct memory] + CY
ADDC A,@Ri	A = A + [memory pointed to by Ri] + CY
ADDC A,#data	A = A + immediate data + CY
SUBB A,Rn	A = A - [Rn] - CY
SUBB A, direct	A = A - [direct memory] - CY
SUBB A,@Ri	A = A - [@Ri] - CY
SUBB A,#data	A = A - immediate data - CY
INC A	A = A + 1
INC Rn	[Rn] = [Rn] + 1
INC direct	[direct] = [direct] + 1
INC @Ri	[@Ri] = [@Ri] + 1
DEC A	A = A - 1
DEC Rn	[Rn] = [Rn] - 1
DEC direct	[direct] = [direct] - 1
DEC @Ri	[@Ri] = [@Ri] - 1
MUL AB	Multiply A & B
DIV AB	Divide A by B
DA A	Decimal adjust A

SILICON LABS

24

ADD A,<source-byte> ADDC A,<source-byte>

- ADD adds the data byte specified by the source operand to the accumulator, leaving the result in the accumulator
- ADDC adds the data byte specified by the source operand, the carry flag and the accumulator contents, leaving the result in the accumulator
- Operation of both the instructions, ADD and ADDC, can affect the carry flag (CY), auxiliary carry flag (AC) and the overflow flag (OV)
  - CY=1 If there is a carryout from bit 7; cleared otherwise
  - AC =1 If there is a carryout from the lower 4-bit of A i.e. from bit 3; cleared otherwise
  - OV=1 If the signed result cannot be expressed within the number of bits in the destination operand; cleared otherwise

SUBB A,<source-byte>

- SUBB subtracts the specified data byte and the carry flag together from the accumulator, leaving the result in the accumulator

  CY=1 If a borrow is needed for bit 7; cleared otherwise

  AC =1 If a borrow is needed for bit 3, cleared otherwise

  OV=1 If a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not into bit 6.

- *Example*:

  The accumulator holds 0C1H (11000001B), Register1 holds 40H (01000000B) and the CY=1.The instruction,

```
SUBB   A, R1
```

  gives the value 70H (01110000B) in the accumulator, with the CY=0 and AC=0 but OV=1

INC <byte>

- Increments the data variable by 1. The instruction is used in register, direct or register direct addressing modes
- *Example*:

```
INC    6FH
```

If the internal RAM location 6FH contains 30H, then the instruction increments this value, leaving 31H in location 6FH

- *Example*:

```
MOV    R1, #5E
INC    R1
INC    @R1
```

- If R1=5E (01011110) and internal RAM location 5FH contains 20H, the instructions will result in R1=5FH and internal RAM location 5FH to increment by one to 21H

DEC <byte>

- The data variable is decremented by 1
- The instruction is used in accumulator, register, direct or register direct addressing modes
- A data of value 00H underflows to FFH after the operation
- No flags are affected

INC DPTR

- Increments the 16-bit data pointer by 1
- DPTR is the only 16-bit register that can be incremented
- The instruction adds one to the contents of DPTR directly

MUL AB

- Multiplies A & B and the 16-bit result stored in [B15-8], [A7-0]
- Multiplies the unsigned 8-bit integers in the accumulator and the B register
- The **Low** order byte of the 16-bit product will go to the accumulator and the **High** order byte will go to the B register
- If the product is greater than 255 (FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
- If ACC=85 (55H) and B=23 (17H), the instruction gives the product 1955 (07A3H), so B is now 07H and the accumulator is A3H. The overflow flag is set and the carry flag is cleared.

DIV AB

- Divides A by B
- The integer part of the quotient is stored in A and the remainder goes to the B register
- If ACC=90 (5AH) and B=05(05H), the instruction leaves 18 (12H) in ACC and the value 00 (00H) in B, since 90/5 = 18 (quotient) and 00 (remainder)
- Carry and OV are both cleared
- *If B contains 00H before the division operation, then the values stored in ACC and B are undefined and an overflow flag is set. The carry flag is cleared.*

31

DA A

- This is a decimal adjust instruction
- It adjusts the 8-bit value in ACC resulting from operations like ADD or ADDC and produces two 4-bit digits (in packed Binary Coded Decimal (BCD) format)
- Effectively, this instruction performs the decimal conversion by adding 00H, 06H, 60H or 66H to the accumulator, depending on the initial value of ACC and PSW
- If ACC bits A3-0 are greater than 9 (xxxx1010-xxxx1111), or if AC=1, then a value 6 is added to the accumulator to produce a correct BCD digit in the lower order nibble
- If CY=1, because the high order bits A7-4 is now exceeding 9 (1010xxxx-1111xxxx), then these high order bits will be increased by 6 to produce a correct proper BCD in the high order nibble but not clear the carry

32

Logical Operations

- Logical instructions perform Boolean operations (AND, OR, XOR, and NOT) on data bytes on a ***bit-by-bit*** basis

Mnemonic	Description
ANL A, Rn	A = A & [Rn]
ANL A, direct	A = A & [direct memory]
ANL A,@Ri	A = A & [memory pointed to by Ri]
ANL A,#data	A= A & immediate data
ANL direct,A	[direct] = [direct] & A
ANL direct,#data	[direct] = [direct] & immediate data
ORL A, Rn	A = A OR [Rn]
ORL A, direct	A = A OR [direct]
ORL A,@Ri	A = A OR [@RI]
ORL A,#data	A = A OR immediate data
ORL direct,A	[direct] = [direct] OR A
ORL direct,#data	[direct] = [direct] OR immediate data
XRL A, Rn	A = A XOR [Rn]
XRL A, direct	A = A XOR [direct memory]
XRL A,@Ri	A = A XOR [@Ri]
XRL A,#data	A = A XOR immediate data
XRL direct,A	[direct] = [direct] XOR A
XRL direct,#data	[direct] = [direct] XOR immediate data
CLR A	Clear A
CPL A	Complement A
RL A	Rotate A left
RLC A	Rotate A left (through C)
RR A	Rotate A right
RRC A	Rotate A right (through C)
SWAP A	Swap nibbles

33

ANL <dest-byte>,<source-byte>

- This instruction performs the logical AND operation on the source and destination operands and stores the result in the destination variable
- No flags are affected
- *Example*:

```
ANL   A,R2
```

If ACC=D3H (11010011) and R2=75H (01110101), the result of the instruction is ACC=51H (01010001)

- The following instruction is also useful when there is a need to mask a byte
- *Example*:

```
ANL   P1,#10111001B
```

34

ORL <dest-byte>,<source-byte>

- This instruction performs the logical OR operation on the source and destination operands and stores the result in the destination variable
- No flags are affected
- *Example*:

```
ORL   A,R2
```

If ACC=D3H (11010011) and R2=75H (01110101), the result of the instruction is ACC=F7H (11110111)

- *Example*:

```
ORL   P1,#11000010B
```

This instruction sets bits 7, 6, and 1 of output Port 1

35

XRL <dest-byte>,<source-byte>

- This instruction performs the logical XOR (Exclusive OR) operation on the source and destination operands and stores the result in the destination variable
- No flags are affected
- *Example*:

```
XRL   A,R0
```

If ACC=C3H (11000011) and R0=AAH (10101010), then the instruction results in ACC=69H (01101001)

- *Example*:

```
XRL   P1,#00110001
```

This instruction complements bits 5, 4, and 0 of output Port 1

36

CLR A and CPL A

CLR A

- This instruction clears the accumulator (all bits set to 0)
- No flags are affected
- If ACC=C3H, then the instruction results in ACC=00H

CPL A

- This instruction logically complements each bit of the accumulator (one's complement)
- No flags are affected
- If ACC=C3H (11000011), then the instruction results in ACC=3CH (00111100)

SILICON LABS

37

RL A

- The 8 bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position.
- No flags are affected
- If ACC=C3H (11000011), then the instruction results in ACC=87H (10000111) with the carry unaffected

SILICON LABS

38

RLC A

- The instruction rotates the accumulator contents one bit to the left through the carry flag
- Bit 7 of the accumulator will move into carry flag and the original value of the carry flag will move into the Bit 0 position
- No other flags are affected
- If ACC=C3H (11000011), and the carry flag is 1, the instruction results in ACC=87H (10000111) with the carry flag set

39

RR A

- The 8 bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position.
- No flags are affected
- If ACC=C3H (11000011), then the instruction results in ACC=E1H (11100001) with the carry unaffected

40

RRC A

- The instruction rotates the accumulator contents one bit to the right through the carry flag
- The original value of carry flag will move into Bit 7 of the accumulator and Bit 0 rotated into carry flag
- No other flags are affected
- If ACC=C3H (11000011), and the carry flag is 0, the instruction results in ACC=61H (01100001) with the carry flag set

41

SWAP A

- This instruction interchanges the low order 4-bit nibbles (A3-0) with the high order 4-bit nibbles (A7-4) of the ACC
- The operation can also be thought of as a 4-bit rotate instruction
- No flags are affected
- If ACC=C3H (11000011), then the instruction leaves ACC=3CH (00111100)

42

Data Transfer Instructions

- Data transfer instructions can be used to transfer data between an internal RAM location and SFR location without going through the accumulator
- It is possible to transfer data between the internal and external RAM by using indirect addressing
- The upper 128 bytes of data RAM are accessed only by indirect addressing and the SFRs are accessed only by direct addressing

43

Data Transfer Instructions

Mnemonic	Description
MOV @Ri, direct	[@Ri] = [direct]
MOV @Ri, #data	[@Ri] = immediate data
MOV DPTR, #data 16	[DPTR] = immediate data
MOVC A,@A+DPTR	A = Code byte from [@A+DPTR]
MOVC A,@A+PC	A = Code byte from [@A+PC]
MOVX A,@Ri	A = Data byte from external ram [@Ri]
MOVX A,@DPTR	A = Data byte from external ram [@DPTR]
MOVX @Ri, A	External[@Ri] = A
MOVX @DPTR,A	External[@DPTR] = A
PUSH direct	Push into stack
POP direct	Pop from stack
XCH A,Rn	A = [Rn], [Rn] = A
XCH A, direct	A = [direct], [direct] = A
XCH A, @Ri	A = [@Rn], [@Rn] = A
XCHD A,@Ri	Exchange low order digits

44

MOV <dest-byte>,<source-byte>

- This instruction moves the source byte into the destination location
- The source byte is not affected, neither are any other registers or flags
- *Example*:

```
MOV   R1,#60      ;R1=60H
MOV   A,@R1       ;A=[60H]
MOV   R2,#61      ;R2=61H
ADD   A,@R2       ;A=A+[61H]
MOV   R7,A        ;R7=A
```

- If internal RAM locations 60H=10H, and 61H=20H, then after the operations of the above instructions R7=A=30H. The data contents of memory locations 60H and 61H remain intact.

45

MOV DPTR, #data 16

- This instruction loads the data pointer with the 16-bit constant and no flags are affected

- *Example*:

```
MOV   DPTR,#1032
```

- This instruction loads the value 1032H into the data pointer, i.e. DPH=10H and DPL=32H.

46

MOVC A,@A + <base-reg>

- This instruction moves a code byte from program memory into ACC
- The effective address of the byte fetched is formed by adding the original 8-bit accumulator contents and the contents of the base register, which is either the data pointer (DPTR) or program counter (PC)
- 16-bit addition is performed and no flags are affected
- The instruction is useful in reading the look-up tables in the program memory
- If the PC is used, it is incremented to the address of the following instruction before being added to the ACC
- *Example*:

```
                CLR   A
LOC1:           INC   A
                MOVC  A,@A + PC
                RET
Look_up         DB    10H
                DB    20H
                DB    30H
                DB    40H
```

- The subroutine takes the value in the accumulator to 1 of 4 values defined by the DB (define byte) directive
- After the operation of the subroutine it returns ACC=20H

47

MOVX <dest-byte>,<source-byte>

- This instruction transfers data between ACC and a byte of external data memory
- There are two forms of this instruction, the only difference between them is whether to use an 8-bit or 16-bit indirect addressing mode to access the external data RAM
- The 8-bit form of the MOVX instruction uses the EMI0CN SFR to determine the upper 8 bits of the effective address to be accessed and the contents of R0 or R1 to determine the lower 8 bits of the effective address to be accessed
- *Example*:

```
MOV     EMI0CN,#10H     ;Load high byte of
                        ;address into EMI0CN.
MOV     R0,#34H         ;Load low byte of
                        ;address into R0(or R1).
MOVX    A,@R0           ;Load contents of 1034H
                        ;into ACC.
```

48

MOVX <dest-byte>,<source-byte>

- The 16-bit form of the MOVX instruction accesses the memory location pointed to by the contents of the DPTR register

- *Example*:

```
MOV   DPTR,#1034H ;Load DPTR with 16 bit
                  ;address to read (1034H).
MOVX  A,@DPTR     ;Load contents of 1034H
                  ;into ACC.
```

- The above example uses the 16-bit immediate MOV DPTR instruction to set the contents of DPTR

- Alternately, the DPTR can be accessed through the SFR registers DPH, which contains the upper 8 bits of DPTR, and DPL, which contains the lower 8 bits of DPTR

PUSH Direct

- This instruction increments the stack pointer (SP) by 1

- The contents of *Direct*, which is an internal memory location or a SFR, are copied into the internal RAM location addressed by the stack pointer

- No flags are affected

- *Example*:

```
PUSH  22H
PUSH  23H
```

- Initially the SP points to memory location 4FH and the contents of memory locations 22H and 23H are 11H and 12H respectively. After the above instructions, SP=51H, and the internal RAM locations 50H and 51H will store 11H and 12H respectively.

POP Direct

- This instruction reads the contents of the internal RAM location addressed by the stack pointer (SP) and decrements the stack pointer by 1. The data read is then transferred to the *Direct* address which is an internal memory or a SFR. No flags are affected.

- *Example*:

```
POP   DPH
POP   DPL
```

- If SP=51H originally and internal RAM locations 4FH, 50H and 51H contain the values 30H, 11H and 12H respectively, the instructions above leave SP=4FH and DPTR=1211H

```
POP   SP
```

- If the above line of instruction follows, then SP=30H. In this case, SP is decremented to 4EH before being loaded with the value popped (30H)

XCH A,<byte>

- This instruction swaps the contents of ACC with the contents of the indicated data byte

- *Example*:

```
XCH   A,@R0
```

- Suppose R0=2EH, ACC=F3H (11110011) and internal RAM location 2EH=76H (01110110). The result of the above instruction leaves RAM location 2EH=F3H and ACC=76H.

XCHD A,@Ri

- This instruction exchanges the low order nibble of ACC (bits 0-3), with that of the internal RAM location pointed to by Ri register
- The high order nibbles (bits 7-4) of both the registers remain the same
- No flags are affected
- Example:

```
XCHD    A,@R0
```

If R0=2EH, ACC=76H (01110110) and internal RAM location 2EH=F3H (11110011), the result of the instruction leaves RAM location 2EH=F6H (11110110) and ACC=73H (01110011)

53

Boolean Variable Instructions

- The C8051F020 processor can perform single bit operations
- The operations include *set, clear*, as well as *and, or* and *complement* instructions
- Also included are bit–level moves or conditional jump instructions
- All bit accesses use direct addressing

Mnemonic	Description
CLR C	Clear C
CLR bit	Clear direct bit
SETB C	Set C
SETB bit	Set direct bit
CPL C	Complement c
CPL bit	Complement direct bit
ANL C,bit	AND bit with C
ANL C,/bit	AND NOT bit with C
ORL C,bit	OR bit with C
ORL C,/bit	OR NOT bit with C
MOV C,bit	MOV bit to C
MOV bit,C	MOV C to bit
JC rel	Jump if C set
JNC rel	Jump if C not set
JB bit,rel	Jump if specified bit set
JNB bit,rel	Jump if specified bit not set
JBC bit,rel	if specified bit set then clear it and jump

54

CLR <bit>

- This operation clears (reset to 0) the specified bit indicated in the instruction
- No other flags are affected
- CLR instruction can operate on the carry flag or any directly-addressable bit
- *Example*:

```
CLR   P2.7
```

If Port 2 has been previously written with DCH (11011100), then the operation leaves the port set to 5CH (01011100)

55

SETB <bit>

- This operation sets the specified bit to 1
- SETB instruction can operate on the carry flag or any directly-addressable bit
- No other flags are affected
- *Example*:

```
SETB  C
SETB  P2.0
```

- If the carry flag is cleared and the output Port 2 has the value of 24H (00100100), then the result of the instructions sets the carry flag to 1 and changes the Port 2 value to 25H (00100101)

56

CPL <bit>

- This operation complements the bit indicated by the operand
- No other flags are affected
- CPL instruction can operate on the carry flag or any directly-addressable bit
- *Example*:

```
CPL   P2.1
CPL   P2.2
```

- If Port 2 has the value of 53H (01010011) before the start of the instructions, then after the execution of the instructions it leaves the port set to 55H (01010101)

57

ANL C, <source-bit>

- This instruction ANDs the bit addressed with the Carry bit and stores the result in the Carry bit itself
- If the source bit is a logical 0, then the instruction clears the carry flag; else the carry flag is left in its original value
- If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, ***but the source bit itself is not affected***
- No other flags are affected
- *Example*:

```
MOV   C,P2.0      ;Load C with input pin
                  ;state of P2.0.
ANL   C,P2.7      ;AND carry flag with
                  ;bit 7 of P2.
MOV   P2.1,C      ;Move C to bit 1 of Port 2.
ANL   C,/OV       ;AND with inverse of OV flag.
```

- If P2.0=1, P2.7=0 and OV=0 initially, then after the above instructions, P2.1=0, CY=0 and the OV remains unchanged, i.e. OV=0

58

ORL C, <source-bit>

- This instruction ORs the bit addressed with the Carry bit and stores the result in the Carry bit itself
- It sets the carry flag if the source bit is a logical 1; else the carry is left in its original value
- If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, ***but the source bit itself is not affected***
- No other flags are affected
- *Example*:

```
MOV     C,P2.0          ;Load C with input pin
                        ;state of P2.0.
ORL     C,P2.7          ;OR carry flag with
                        ;bit 7 of P2.
MOV     P2.1,C          ;Move C to bit 1 of
                        ;port 2.
ORL     C,/OV           ;OR with inverse of OV
                        ;flag.
```

59

MOV <dest-bit>,<source-bit>

- The instruction loads the value of source operand bit into the destination operand bit
- One of the operands **must** be the carry flag; the other may be any directly-addressable bit
- No other register or flag is affected
- *Example*:

```
MOV     P2.3,C
MOV     C,P3.3
MOV     P2.0,C
```

- If P2=C5H (11000101), P3.3=0 and CY=1 initially, then after the above instructions, P2=CCH (11001100) and CY=0.

60

JC rel

- This instruction branches to the address, indicated by the label, if the carry flag is set, otherwise the program continues to the next instruction
- No flags are affected
- *Example*:

```
CLR   C
SUBB  A,R0
JC    ARRAY1
MOV   A,#20H
```

- The carry flag is cleared initially. After the SUBB instruction, if the value of A is smaller than R0, then the instruction sets the carry flag and causes program execution to branch to ARRAY1 address, otherwise it continues to the MOV instruction.

SILICON LABS

61

JNC rel

- This instruction branches to the address, indicated by the label, if the carry flag is **not** set, otherwise the program continues to the next instruction
- No flags are affected. **The carry flag is not modified.**
- *Example*:

```
CLR   C
SUBB  A,R0
JNC   ARRAY2
MOV   A,#20H
```

- The above sequence of instructions will cause the jump to be taken if the value of A is greater than or equal to R0. Otherwise the program will continue to the MOV instruction.

SILICON LABS

62

JB <bit>,rel

- This instruction jumps to the address indicated if the destination bit is 1, otherwise the program continues to the next instruction
- No flags are affected. **The bit tested is not modified**.
- *Example*:

```
JB    ACC.7,ARRAY1
JB    P1.2,ARRAY2
```

- If the accumulator value is 01001010 and Port 1=57H (01010111), then the above instruction sequence will cause the program to branch to the instruction at ARRAY2

63

JNB <bit>,rel

- This instruction jumps to the address indicated if the destination bit is 0, otherwise the program continues to the next instruction
- No flags are affected. **The bit tested is not modified**.
- *Example*:

```
JNB   ACC.6,ARRAY1
JNB   P1.3,ARRAY2
```

- If the accumulator value is 01001010 and Port 1=57H (01010111), then the above instruction sequence will cause the program to branch to the instruction at ARRAY2

64

JBC <bit>,rel

- If the source bit is 1, this instruction clears it and branches to the address indicated; else it proceeds with the next instruction
- **The bit is not cleared if it is already a 0**. No flags are affected.
- *Example*:

```
JBC    P1.3,ARRAY1
JBC    P1.2,ARRAY2
```

- If P1=56H (01010110), the above instruction sequence will cause the program to branch to the instruction at ARRAY2, modifying P1 to 52H (01010010)

65

Program Branching Instructions

- Program branching instructions are used to control the flow of actions in a program
- Some instructions provide decision making capabilities and transfer control to other parts of the program, e.g. conditional and unconditional branches

Mnemonic	Description
ACALL addr11	Absolute subroutine call
LCALL addr16	Long subroutine call
RET	Return from subroutine
RETI	Return from interrupt
AJMP addr11	Absolute jump
LJMP addr16	Long jump
SJMP rel	Short jump
JMP @A+DPTR	Jump indirect
JZ rel	Jump if A=0
JNZ rel	Jump if A NOT=0
CJNE A,direct,rel	Compare and Jump if Not Equal
CJNE A,#data,rel	
CJNE Rn,#data,rel	
CJNE @Ri,#data,rel	
DJNZ Rn,rel	Decrement and Jump if Not Zero
DJNZ direct,rel	
NOP	No Operation

66

ACALL addr11

- This instruction **unconditionally** calls a subroutine indicated by the address
- The operation will cause the PC to increase by 2, then it pushes the 16-bit PC value onto the stack (low order byte first) and increments the stack pointer twice
- The PC is now loaded with the value *addr11* and the program execution continues from this new location
- The subroutine called must therefore start within the same 2 kB block of the program memory
- No flags are affected
- *Example*:

```
ACALL LOC_SUB
```

- If SP=07H initially and the label “LOC_SUB” is at program memory location 0567H, then executing the instruction at location 0230H, SP=09H, internal RAM locations 08H and 09H will contain 32H and 02H respectively and PC=0567H

67

LCALL addr16

- This instruction calls a subroutine located at the indicated address
- The operation will cause the PC to increase by 3, then it pushes the 16-bit PC value onto the stack (low order byte first) and increments the stack pointer twice
- The PC is then loaded with the value *addr16* and the program execution continues from this new location
- Since it is a Long call, the subroutine may therefore begin anywhere in the full 64 kB program memory address space
- No flags are affected
- *Example*:

```
LCALL LOC_SUB
```

- Initially, SP=07H and the label “LOC_SUB” is at program memory location 2034H. Executing the instruction at location 0230H, SP=09H, internal RAM locations 08H and 09H contain 33H and 02H respectively and PC=2034H

68

RET

- This instruction returns the program from a subroutine
- RET pops the high byte and low byte address of PC from the stack and decrements the SP by 2
- The execution of the instruction will result in the program to resume from the location just after the “call” instruction
- No flags are affected
- Suppose SP=0BH originally and internal RAM locations 0AH and 0BH contain the values 30H and 02H respectively. The instruction leaves SP=09H and program execution will continue at location 0230H

69

RETI

- This instruction returns the program from an interrupt subroutine
- RETI pops the high byte and low byte address of PC from the stack and restores the interrupt logic to accept additional interrupts
- SP decrements by 2 and no other registers are affected. However the PSW is not automatically restored to its pre-interrupt status
- After the RETI, program execution will resume immediately after the point at which the interrupt is detected
- Suppose SP=0BH originally and an interrupt is detected during the instruction ending at location 0213H
  - Internal RAM locations 0AH and 0BH contain the values 14H and 02H respectively
  - The RETI instruction leaves SP=09H and returns program execution to location 0234H

70

AJMP addr11

- The AJMP instruction transfers program execution to the destination address which is located at the absolute short range distance (short range means 11-bit address)
- The destination must therefore be within the same 2kB block of program memory
- *Example*:

```
AJMP   NEAR
```

- If the label NEAR is at program memory location 0120H, the AJMP instruction at location 0234H loads the PC with 0120H

71

LJMP addr16

- The LJMP instruction transfers program execution to the destination address which is located at the absolute long range distance (long range means 16-bit address)
- The destination may therefore be anywhere in the full 64 kB program memory address space
- No flags are affected
- *Example*:

```
LJMP   FAR_ADR
```

- If the label FAR_ADR is at program memory location 3456H, the LJMP instruction at location 0120H loads the PC with 3456H

72

SJMP rel

- This is a short jump instruction, which increments the PC by 2 and then adds the relative value '*rel*' (signed 8-bit) to the PC
- This will be the new address where the program would branch to unconditionally
- Therefore, the range of destination allowed is from -128 to +127 bytes from the instruction
- *Example*:

```
SJMP   RELSRT
```

- If the label RELSRT is at program memory location 0120H and the SJMP instruction is located at address 0100H, after executing the instruction, PC=0120H.

73

JMP @A + DPTR

- This instruction adds the 8-bit unsigned value of the ACC to the 16-bit data pointer and the resulting sum is returned to the PC
- Neither ACC nor DPTR is altered
- No flags are affected
- *Example*:

```
                MOV   DPTR, #LOOK_TBL
                JMP   @A + DPTR
LOOK_TBL:       AJMP  LOC0
                AJMP  LOC1
                AJMP  LOC2
```

If the ACC=02H, execution jumps to LOC1

- AJMP is a two byte instruction

74

JZ rel

- This instruction branches to the destination address if ACC=0; else the program continues to the next instruction
- The ACC is not modified and no flags are affected
- *Example*:

```
SUBB A,#20H
JZ   LABEL1
DEC  A
```

- If ACC originally holds 20H and CY=0, then the SUBB instruction changes ACC to 00H and causes the program execution to continue at the instruction identified by LABEL1; otherwise the program continues to the DEC instruction

75

JNZ rel

- This instruction branches to the destination address if any bit of ACC is a 1; else the program continues to the next instruction
- The ACC is not modified and no flags are affected
- *Example*:

```
DEC  A
JNZ  LABEL2
MOV  RO, A
```

- If ACC originally holds 00H, then the instructions change ACC to FFH and cause the program execution to continue at the instruction identified by LABEL2; otherwise the program continues to MOV instruction

76

CJNE <dest-byte>,<source-byte>,rel

- This instruction compares the magnitude of the *dest-byte* and the *source-byte* and branches if their values are not equal
- The carry flag is set if the unsigned *dest-byte* is less than the unsigned integer *source-byte*; otherwise, the carry flag is cleared
- Neither operand is affected
- *Example*:

```
CJNE   R3,#50H,NEQU
       … …                  ;R3 = 50H
NEQU:  JC     LOC1          ;If R3 < 50H
       … …                  ;R7 > 50H
LOC1:  … …                  ;R3 < 50H
```


DJNZ <byte>,<rel-addr>

- This instruction is ”decrement jump not zero”
- It decrements the contents of the destination location and if the resulting value is not 0, branches to the address indicated by the source operand
- An original value of 00H underflows to FFH
- No flags are affected
- *Example*:

```
DJNZ 20H,LOC1
DJNZ 30H,LOC2
DJNZ 40H,LOC3
```

- If internal RAM locations 20H, 30H and 40H contain the values 01H, 5FH and 16H respectively, the above instruction sequence will cause a jump to the instruction at LOC2, with the values 00H, 5EH, and 15H in the 3 RAM locations.
  - Note, the first instruction will not branch to LOC1 because the [20H] = 00H, hence the program continues to the second instruction
  - Only after the execution of the second instruction (where the location [30H] = 5FH), then the branching takes place

NOP

- This is the no operation instruction
- The instruction takes one machine cycle operation time
- Hence it is useful to time the ON/OFF bit of an output port
- *Example*:

```
CLR   P1.2
NOP
NOP
NOP
NOP
SETB  P1.2
```

- The above sequence of instructions outputs a low-going output pulse on bit 2 of Port 1 lasting exactly 5 cycles.
  - Note a simple SETB/CLR generates a 1 cycle pulse, so four additional cycles must be inserted in order to have a 5-clock pulse width

79

SILICON LABS

SILICON LABS

www.silabs.com/MCU

Architecture des systèmes à microprocesseur(s) (S6) * 2-6-EL104-C @2012

TABLE DES CODES ASSEMBLEUR 8051

Copier/Coller du site: http://www.keil.com/support/man/docs/is51/is51_opcodes.htm

The following table lists the 8051 instructions by HEX code.

Hex Code	Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	addr11
02	3	LJMP	addr16
03	1	RR	A
04	1	INC	A
05	2	IN C	direct
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit, offset
11	2	ACALL	addr11
12	3	LCALL	addr16
13	1	RRC	A
14	1	DEC	A
15	2	DEC	direct
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	bit, offset
21	2	AJMP	addr11
22	1	RET	
23	1	RL	A
24	2	ADD	A, #immed
25	2	ADD	A, direct
26	1	ADD	A, @R0
27	1	ADD	A, @R1
28	1	ADD	A, R0
29	1	ADD	A, R1
2A	1	ADD	A, R2
2B	1	ADD	A, R3
2C	1	ADD	A, R4
2D	1	ADD	A, R5
2E	1	ADD	A, R6
2F	1	ADD	A, R7
30	3	JNB	bit, offset
31	2	ACALL	addr11
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A, #immed
35	2	ADDC	A, direct
36	1	ADDC	A, @R0
37	1	ADDC	A, @R1
38	1	ADDC	A, R0
39	1	ADDC	A, R1
3A	1	ADDC	A, R2
3B	1	ADDC	A, R3
3C	1	ADDC	A, R4
3D	1	ADDC	A, R5
80	2	SJMP	offset
81	2	AJMP	addr11
82	2	ANL	C, bit
83	1	MOVC	A, @A+PC
84	1	DIV	AB
85	3	MOV	direct, direct
86	2	MOV	direct, @R0
87	2	MOV	direct, @R1
88	2	MOV	direct, R0
89	2	MOV	direct, R1
8A	2	MOV	direct, R2
8B	2	MOV	direct, R3
8C	2	MOV	direct, R4
8D	2	MOV	direct, R5
8E	2	MOV	direct, R6
8F	2	MOV	direct, R7
90	3	MOV	DPTR, #immed
91	2	ACALL	addr11
92	2	MOV	bit, C
93	1	MOVC	A, @A+DPTR
94	2	SUBB	A, #immed
95	2	SUBB	A, direct
96	1	SUBB	A, @R0
97	1	SUBB	A, @R1
98	1	SUBB	A, R0
99	1	SUBB	A, R1
9A	1	SUBB	A, R2
9B	1	SUBB	A, R3
9C	1	SUBB	A, R4
9D	1	SUBB	A, R5
9E	1	SUBB	A, R6
9F	1	SUBB	A, R7
A0	2	ORL	C, /bit
A1	2	AJMP	addr11
A2	2	MOV	C, bit
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@R0, direct
A7	2	MOV	@R1, direct
A8	2	MOV	R0, direct
A9	2	MOV	R1, direct
AA	2	MOV	R2, direct
AB	2	MOV	R3, direct
AC	2	MOV	R4, direct
AD	2	MOV	R5, direct
AE	2	MOV	R6, direct
AF	2	MOV	R7, direct
B0	2	ANL	C, /bit
B1	2	ACALL	addr11
B2	2	CPL	bit
B3	1	CPL	C
B4	3	CJNE	A, #immed, offset
B5	3	CJNE	A, direct, offset
B6	3	CJNE	@R0, #immed, offset
B7	3	CJNE	@R1, #immed, offset
B8	3	CJNE	R0, #immed, offset
B9	3	CJNE	R1, #immed, offset
BA	3	CJNE	R2, #immed, offset
BB	3	CJNE	R3, #immed, offset
BC	3	CJNE	R4, #immed, offset
BD	3	CJNE	R5, #immed, offset

3E	1	ADDC	A, R6	BE	3	CJNE	R6, #immed, offset
3F	1	ADDC	A, R7	BF	3	CJNE	R7, #immed, offset
40	2	JC	offset	C0	2	PUSH	direct
41	2	AJMP	addr11	C1	2	AJMP	addr11
42	2	ORL	direct, A	C2	2	CLR	bit
43	3	ORL	direct, #immed	C3	1	CLR	C
44	2	ORL	A, #immed	C4	1	SWAP	A
45	2	ORL	A, direct	C5	2	XCH	A, direct
46	1	ORL	A, @R0	C6	1	XCH	A, @R0
47	1	ORL	A, @R1	C7	1	XCH	A, @R1
48	1	ORL	A, R0	C8	1	XCH	A, R0
49	1	ORL	A, R1	C9	1	XCH	A, R1
4A	1	ORL	A, R2	CA	1	XCH	A, R2
4B	1	ORL	A, R3	CB	1	XCH	A, R3
4C	1	ORL	A, R4	CC	1	XCH	A, R4
4D	1	ORL	A, R5	CD	1	XCH	A, R5
4E	1	ORL	A, R6	CE	1	XCH	A, R6
4F	1	ORL	A, R7	CF	1	XCH	A, R7
50	2	JNC	offset	D0	2	POP	direct
51	2	ACALL	addr11	D1	2	ACALL	addr11
52	2	ANL	direct, A	D2	2	SETB	bit
53	3	ANL	direct, #immed	D3	1	SETB	C
54	2	ANL	A, #immed	D4	1	DA	A
55	2	ANL	A, direct	D5	3	DJNZ	direct, offset
56	1	ANL	A, @R0	D6	1	XCHD	A, @R0
57	1	ANL	A, @R1	D7	1	XCHD	A, @R1
58	1	ANL	A, R0	D8	2	DJNZ	R0, offset
59	1	ANL	A, R1	D9	2	DJNZ	R1, offset
5A	1	ANL	A, R2	DA	2	DJNZ	R2, offset
5B	1	ANL	A, R3	DB	2	DJNZ	R3, offset
5C	1	ANL	A, R4	DC	2	DJNZ	R4, offset
5D	1	ANL	A, R5	DD	2	DJNZ	R5, offset
5E	1	ANL	A, R6	DE	2	DJNZ	R6, offset
5F	1	ANL	A, R7	DF	2	DJNZ	R7, offset
60	2	JZ	offset	E0	1	MOVX	A, @DPTR
61	2	AJMP	addr11	E1	2	AJMP	addr11
62	2	XRL	direct, A	E2	1	MOVX	A, @R0
63	3	XRL	direct, #immed	E3	1	MOVX	A, @R1
64	2	XRL	A, #immed	E4	1	CLR	A
65	2	XRL	A, direct	E5	2	MOV	A, direct
66	1	XRL	A, @R0	E6	1	MOV	A, @R0
67	1	XRL	A, @R1	E7	1	MOV	A, @R1
68	1	XRL	A, R0	E8	1	MOV	A, R0
69	1	XRL	A, R1	E9	1	MOV	A, R1
6A	1	XRL	A, R2	EA	1	MOV	A, R2
6B	1	XRL	A, R3	EB	1	MOV	A, R3
6C	1	XRL	A, R4	EC	1	MOV	A, R4
6D	1	XRL	A, R5	ED	1	MOV	A, R5
6E	1	XRL	A, R6	EE	1	MOV	A, R6
6F	1	XRL	A, R7	EF	1	MOV	A, R7
70	2	JNZ	offset	F0	1	MOVX	@DPTR, A
71	2	ACALL	addr11	F1	2	ACALL	addr11
72	2	ORL	C, bit	F2	1	MOVX	@R0, A
73	1	JMP	@A+DPTR	F3	1	MOVX	@R1, A
74	2	MOV	A, #immed	F4	1	CPL	A
75	3	MOV	direct, #immed	F5	2	MOV	direct, A
76	2	MOV	@R0, #immed	F6	1	MOV	@R0, A
77	2	MOV	@R1, #immed	F7	1	MOV	@R1, A
78	2	MOV	R0, #immed	F8	1	MOV	R0, A
79	2	MOV	R1, #immed	F9	1	MOV	R1, A
7A	2	MOV	R2, #immed	FA	1	MOV	R2, A
7B	2	MOV	R3, #immed	FB	1	MOV	R3, A
7C	2	MOV	R4, #immed	FC	1	MOV	R4, A
7D	2	MOV	R5, #immed	FD	1	MOV	R5, A
7E	2	MOV	R6, #immed	FE	1	MOV	R6, A
7F	2	MOV	R7, #immed	FF	1	MOV	R7, A

Table des codes assembleur 8051 - 2

MACRO ASSEMBLEUR 8051

1 INTRODUCTION

Ce document se veut un résumé du manuel « *Macro Assembler and Utilities* » de KEIL Software. Il s'attache à dégager les points principaux, essentiels à la rédaction de programmes assembleurs modulaires et portables. En aucune manière cependant, il ne peut se substituer complètement à la documentation originale.
Rappelons que le terme «assembleur », peut avoir deux significations possibles. Il peut désigner le langage de programmation d'un processeur, exprimé sous forme de mnémoniques (MOV, ADD, JMP….) et d'opérandes. Mais il fait aussi référence au programme informatique qui va permettre de transformer ces mêmes mnémoniques en code binaires, compréhensibles par le processeur. C'est à ce deuxième sens, que nous allons nous intéresser. Pour éviter les ambiguïtés, nous le nommerons « Macro assembleur »

2 DEFINITION

On appelle « assembleur » ou « macro-assembleur », tout outil destiné à faciliter l'écriture de programmes en langage machine. Il permet la transcription de codes symboliques en codes exécutables par le processeur.

Exemple :

Code symbolique	→	**Assemblage**	→	Code exécutable
`MOV DPTR,#01234h`				`90 12 34`

```
Mask_1  EQU $70h
Ctrl_reg EQU $82h
ORL  Ctrl_reg,#Mask_1                    43 82 70
```

En règle générale, l'opération d'assemblage est toujours complétée par l'édition de liens. Cette tâche, accomplie par l'éditeur de liens (Linker) va rassembler plusieurs programmes assemblés séparément, en un seul fichier, en suivant des règles précises d'association.
Pour écrire efficacement en assembleur, il est indispensable de connaître le jeu d'instruction du processeur, mais aussi, d'avoir à l'esprit l'architecture interne du processeur utilisé.

3 ELEMENTS D'UN MACRO-ASSEMBLEUR.

Les principaux éléments, manipulés par un macro-assembleur, sont les instructions machines, exprimées sous la forme de mnémoniques et d'opérandes.

- Les mnémoniques correspondent aux instructions à exécuter (MOV,ADD ,ORL…).
- Les opérandes sont les valeurs, les registres, les adresses mémoire manipulées par les instructions. Ces opérandes peuvent aussi être exprimés sous forme symbolique pour faciliter la tâche du programmeur.

Cependant, pour contrôler le fonctionnement du macro-assembleur, il est impératif de spécifier des règles particulières. On introduit ainsi les directives d'assemblage et les paramètres d'assemblage.
Ces éléments ne produisent pas de codes exécutables par le processeur, mais ils permettent de donner des informations au macro-assembleur pour produire un code assembleur optimisé.

- Les directives d'assemblage précisent la structure d'un programme et son stockage dans la mémoire du processeur, et assurent la gestion des symboles.
- Les paramètres d'assemblage interviennent essentiellement sur la mise en forme des fichiers produits par le macro-assembleur.

4 LA PROGRAMMATION MODULAIRE.

Le macro-assembleur associé à l'éditeur de liens permet d'aborder, sans aucune difficulté la programmation modulaire. Ainsi, un programme complexe et volumineux, pourra être découpé en plusieurs petits éléments indépendants, plus petits, plus simples à mettre au point, et surtout réutilisables.

4.1 Les Segments

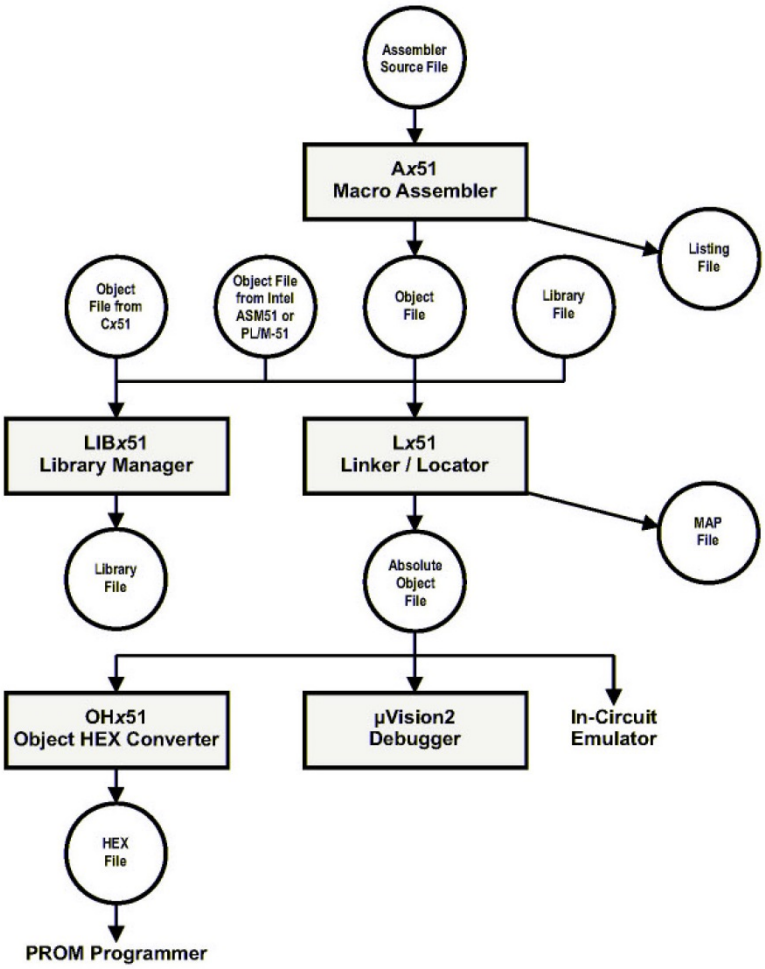
Un segment est un bloc de code ou de données en mémoire. Il peut être relogeable ou absolu.

Un segment est dit « relogeable » quand on ne lui attribue pas d'adresse. On laisse ainsi à l'éditeur de liens le soin d'affecter lui-même une adresse à ce segment.
Par opposition, un segment est dit « absolu » quant au moment de sa définition, une adresse lui est attribuée.
Un segment relogeable est défini par son nom, son type et par des attributs optionnels. Des segments de même nom, dans des modules différents, seront considérés comme faisant partie du même segment, et sont nommés « segments partiels ». C'est l'éditeur de liens qui sera chargé de regrouper les segments partiel en un seul segment.
Dans la mesure du possible, et pour respecter les contraintes de programmation modulaire, on s'attachera à manipuler des segments relogeables. On réservera l'usage des segments absolus, pour traiter des cas particuliers, comme la programmation des vecteurs d'adresse et d'interruption, ou lorsque que l'on veut manipuler des portions bien précises de la mémoire.

4.2 Les Modules

Un Module contient un ou plusieurs segments partiels ou pas. C'est un code source qui peut être assemblé de manière indépendante ; aussi, il contient toutes les définitions des symboles utilisés. Il peut être contenu dans un seul fichier source, ou dans plusieurs fichiers en utilisant le paramètre d'assemblage « Include ». Le macro assembleur génèrera un fichier « objet » par modules. Un fichier « objet » contient du code binaire machine relogeable. Les adresses de sauts absolus ne sont pas encore à ce niveau-là renseignées.

5 LA CHAINE DE DEVELOPPEMENT.



Le macro assembleur (ASM51) traite chaque fichier source (chaque module) séparément et génère un fichier code objet.
L'éditeur de lien (BL51) traite tous les fichiers objet en une passe pour produire un fichier objet absolu contenant la totalité du code.
Lors de cette opération, trois opérations principales sont exécutées :

- Assignation d'adresses absolues aux segments relogeables.
- Combinaison des segments partiels.
- Résolution des références inter-modules.

Pour terminer, divers utilitaires permettent de réaliser des conversions de format pour diriger le fichier objet absolu vers divers dispositifs (programmateurs de mémoire, émulateurs…..)

Cas particulier de l'environnement de développement Keil µvision ou Silabs IDE.
Ces environnements de développement vont vous faciliter la tâche en gérant automatiquement les étapes d'assemblage et d'édition de liens. Il vous suffit d'insérer les fichiers source ou objet dans un projet pour que ceux-ci soient obligatoirement traités quand on lance le processus « Assemblage-Edition de liens ».

6 ECRITURE DE PROGRAMMES EN ASSEMBLEUR.

Rappelons qu'un fichier source assembleur contient 3 types d'éléments :

- Les **directives d'assemblage**, informent l'assembleur comment traiter certaines instructions. En outre, elles permettent la définition des constantes, et la réservation d'espace mémoire pour les variables.
- Les **paramètres d'assemblage** contrôlent les opérations d'assemblage, notamment sur la forme des fichiers LST (fichier listing) et objet.
  Directives et paramètres ne produisent pas de code, sauf quand on fait de l'assemblage conditionnel avec les paramètres *If/Else* et *Endif* et *Elseif*.
- Les **instructions** sous la forme :

  [label :] mnémonique [operand] [,operand] [,operand] [;commentaire]

Exemple de code :

```
$TITLE (code_example) ; Paramètre
      CSEG at 0000h          ; Directive
      JMP $                  ; Instruction
      END                    ; Directive
```

6.1 Les commentaires.

Ils sont introduits par le caractère point-virgule (;)

6.2 Les symboles.

Leur taille est limitée à 31 caractères.
Les caractères autorisés sont: A-Z, a-z, 0-9, _,?
Un symbole ne peut pas commencer par un chiffre.

Usage :

- Le symbole représente une expression
  ```
  True_flag EQU        0
  ```
- Le symbole peut être une étiquette.
  ```
  Boucle :      MOV      A,R0
  ```
- Le symbole peut faire référence à l'adresse d'une variable.
  ```
  Buffer        DATA     99h.
  ```

6.3 Les étiquettes (LABEL)

Une étiquette définit une adresse dans le programme ou dans un espace de données. Au moment de sa définition, elle doit apparaître en premier sur une ligne, et être suivie par le caractère « deux-points » (:).

```
Label1 :   DS   2
Copy :     MOV  R6,#12h
```

6.4 Les opérandes

Ils sont utilisés aussi bien dans les directives d'assemblage que dans les instructions.

```
Value1          EQU      3
          DS         10h
```

```
MOV         DPTR,#Init
```

Un opérande peut être:

- Une constante numérique
- Un symbole
- Une chaîne de caractères
- Une expression, qui est une combinaison de nombre, de chaîne de caractères, de symboles et d'opérateurs.

Les **constantes numériques** peuvent être exprimées en binaire (suffixe B,b), en octal (suffixe Q,q,O,o), en décimal (valeur par défaut, suffixe D,d) ou en hexadécimal (suffixe H,h). Attention une valeur numérique ne peut commencer que par un chiffre (0 à 9).

Les **caractères** s'écrivent sous la forme: '*A*'.

Les **chaînes de caractères** s'écrivent aussi sous une forme similaire : '*ceci est une chaine de caractères*'.
Pour les déclarer, on utilisera la syntaxe suivante :

```
Keymsg :   DB    'Press any key to continue'
```

La description de la multitude d'**expressions** réalisables sortirait du cadre de ce résumé. Il conviendra de consulter le manuel « *Macro Assembler and Utilities* » de Keil.

6.5 Le pointeur de position (location counter).

Pour chaque segment manipulé, le macro-assembleur crée un pointeur de position, qui contient à chaque instant, l'offset de l'instruction ou de la donnée en cours d'assemblage, par rapport au début du segment. Ce pointeur de position, initialisé en début de segment, est incrémenté à la fin de chaque ligne assemblée, par le nombre d'octets de code ou de données dans la ligne.
Le contenu du pointeur de position peut toutefois être altéré par la directive ORG.
Le signe dollar ($) retourne dans le code, la valeur courante du pointeur de position, dans le segment.
Exemple :

```
JMP         $                           ; est équivalent à      BCL     JMP BCL
Msg :       DB          'Voilà le message',0
Msglen       EQU        $-Msg           ; Calcul de la longueur de la chaîne
```

7 LES DIRECTIVES D'ASSEMBLAGE.

Rappelons que les directives ne produisent pas de code, et qu'elles n'ont pas d'effet sur le contenu de la mémoire de code, sauf avec les directives DB, DW, DD.

7.1 Les directives de segmentation.

7.1.1 Les segments génériques (relogeables).

Ils ont un nom, une classe mémoire, et des attributs auxiliaires optionnels (non traités dans ce document).
Leur création est assurée par la directive SEGMENT selon :

nom_segment	***SEGMENT***	***Classe_mémoire.***
Myprog	*SEGMENT*	*Code* (ou Bit, Data, Idata, Xdata)

On les utilise avec la directive RSEG, selon :

RSEG	***nom_segment.***
RSEG	*Myprog*

Exemple de déclaration du segment destiné à la pile :

```
Stack           SEGMENT     Idata
                RSEG        Stack
                DS          10h         ;réservation de 16 octets
```

A l'issue de cette déclaration, l'assembleur a réservé 16 octets pour la pile, en zone mémoire IDATA. Par contre, cela ne dispense pas d'initialiser le pointeur de pile avec l'instruction :

```
MOV         SP,#stack-1
```

7.1.2 Les segments absolus.

Au moment de leur déclaration, on leur affecte directement une adresse.
Format :

```
BSEG        AT      Address     ;Segment absolu de type BIT
CSEG        AT      Address     ;Segment absolu de type CODE
DSEG        AT      Address     ; Segment absolu de type DATA
ISEG        AT      Address     ; Segment absolu de type IDATA
XSEG        AT      Address     ; Segment absolu de type XDATA
```

7.2 Définition des symboles.

7.2.1 SET et EQU

Ces directives assignent une valeur numérique à un registre ou à un symbole.
Format :

```
Symbol EQU  Expression
Symbol      SET     Expression
Symbol      EQU     Registre (R0….R7)
Symbol      SET     Registre (R0… R7)
```

Exemples :

```
Consigne    EQU     33h
Count       EQU     R5
Value       SET     100
Value       SET     Value/2
```

A la différence d'EQU, SET permet une redéfinition du symbole.

7.2.2 CODE,DATA,IDATA,XDATA

Elles permettent d'assigner une adresse à un symbole, dans la zone mémoire spécifiée par la directive.

```
symbol    BIT     bit_address          defines a BIT symbol
symbol    CODE    code_address         defines a CODE symbol
symbol    DATA    data_address         defines a DATA symbol
symbol    IDATA   idata_address        defines an IDATA symbol
symbol    XDATA   xdata_address        defines a XDATA symbol
```

where

`symbol`	is the name of the symbol to define. The symbol name can be used anywhere an address of this memory class is valid.
`bit_address`	is the address of a bit in internal data memory in the area 20H .. 2FH or a bit address of an 8051 bit-addressable SFR.
`code_address`	is a code address in the range 0000H .. 0FFFFH.
`data_address`	is a data memory address in the range 0 to 127 or a special function register (SFR) address in the range 128 .. 255.
`idata_address`	is an idata memory address in the range 0 to 255.
`xdata_address`	is an xdata memory address in the range 0 to 65535.

7.3 Initialisation de la mémoire code.

7.3.1 DB

Initialisation d'une partie de la mémoire CODE avec des valeurs octet. DB ne peut être utilisée que dans un segment de code.
Format :

Label : DB expression, expression …

Exemple :

```
REQUEST :  DB    'Press Any Key to Continue',0
```

7.3.2 DW

Initialisation d'une partie de la mémoire CODE avec des valeurs sur 16 bits. DW ne peut être utilisée que dans un segment de code.
Format :

Label : DW expression, expression …

Exemple :

TABLE : DW TABLE, TABLE+10,TABLE+20

7.4 Réservation de mémoire.

7.4.1 DBIT

Réservation de bit dans un segment BIT.
Format :

Label : *DBIT* *Expression* ; Expression : nombre de bits à réserver

Exemple :

```
On_Flag    DBIT      3     ;réserve 3 bits
```

7.4.2 DS

Réservation d'un nombre spécifique d'octet dans un segment DATA, IDATA ou XDATA.
Format :

Label : *DS* *Expression* *;* Expression : nombre d'octets à réserver

Exemple :

```
Time :          DS    10h
```

7.5 Directives pour l'édition de liens.

7.5.1 PUBLIC

Liste les symboles déclarés dans le module et disponibles pour d'autres modules. Tout symbole déclaré PUBLIC pourra être réutilisé sans déclaration par d'autres modules.
Format : *PUBLIC symbol, symbol…*

7.5.2 EXTRN

Liste les symboles qui sont déclarés dans d'autres modules. Tout symbole déclaré EXTRN pourra être utilisé dans le module sans déclaration.
Format : *EXTRN* *classe_mémoire* *(symbole, …)*
Avec classe_mémoire : classe mémoire dans laquelle le symbole a été défini. Elle peut être du type BIT, CODE, DATA, IDATA, XDATA ou NUMBER (pour spécifier des symboles non typés).
Exemple :

```
EXTRN      CODE      (putchar,getkey);les routine putchar et getkey ont été écrites
                                      ; dans un autre module
```

7.5.3 NAME

Pour nommer le module.
Format : *NAME* *nom_du_module*

7.6 Contrôle des adresses.

7.6.1 ORG

Permet la modification du pointeur de position dans un segment.
Format :

```
ORG    Expression     ;Avec expression : adresse absolue.
```

Exemple :

```
ORG    73h    ; le pointeur de position est initialisé à 73h
              ; dans cet exemple on pointe ainsi sur le vecteur d'interruption du Timer3
```

7.6.2 USING

Permet de spécifier le banc de registre utilisé.
Format : *USING Expression ;Avec expression : 0, 1, 2 ou 3.*
Attention, *Using* ne génère pas de code, il vous revient de coder la commutation de bancs de registre.

7.7 Autres directives.

7.7.1 END

Indique la fin du programme.

8 LES PARAMETRES D'ASSEMBLAGE

Ils sont introduits par le caractère '$'. Le paramètre le plus couramment utilisé est « include » qui permet d'insérer un fichier à l'intérieur d'un programme source.

Exemple : *$Include (C8051F020.inc)*

Pour les autres paramètres d'assemblage, consulter le manuel « *Macro Assembler and Utilities* ».

9 FICHIER TEMPLATE.A51

Ce fichier donne un exemple d'utilisation de nombreuses directives d'assemblage.

```
;------------------------------------------------------------------------------
; Source code template for A51 assembler modules.
; Copyright (c) 1995-2000 KEIL Software, Inc.
;------------------------------------------------------------------------------
$NOMOD51                ; disable predefined 8051 registers
#include <c8051F020.inc>      // include CPU definition file

;------------------------------------------------------------------------------
; Change names in lowercase to suit your needs.
;
; This assembly template gives you an idea of how to use the A51
; Assembler.  You are not required to build each module this way-this is only
; an example.
;
; All entries except the END statement at the End Of File are optional.
;
; This file cannot provide for every possible use of the A251/A51 Assembler.
; Refer to the A51 User's Guide for more information.
;------------------------------------------------------------------------------

;------------------------------------------------------------------------------
; Module name (optional)
;------------------------------------------------------------------------------
NAME            module_name

;------------------------------------------------------------------------------
; Here, you may import symbols form other modules.
;------------------------------------------------------------------------------
EXTRN   CODE   (code_symbol)    ; May be a subroutine entry declared in
                                ; CODE segments or with CODE directive.

EXTRN   DATA   (data_symbol)    ; May be any symbol declared in DATA segments
```

```
                                ; segments or with DATA directive.

EXTRN   BIT    (bit_symbol)     ; May be any symbol declared in BIT segments
                                ; or with BIT directive.

EXTRN   XDATA  (xdata_symbol)   ; May be any symbol declared in XDATA segments
                                ; or with XDATA directive.

EXTRN   NUMBER (typeless_symbol); May be any symbol declared with EQU or SET
                                ; directive

;------------------------------------------------------------------------------
; You may include more than one symbol in an EXTRN statement.
;------------------------------------------------------------------------------
EXTRN   CODE (sub_routine1, sub_routine2), DATA (variable_1)

;------------------------------------------------------------------------------
; Force a page break in the listing file.
;------------------------------------------------------------------------------
$EJECT
;------------------------------------------------------------------------------
; Here, you may export symbols to other modules.
;------------------------------------------------------------------------------
PUBLIC  data_variable
PUBLIC  code_entry
PUBLIC  typeless_number
PUBLIC  xdata_variable
PUBLIC  bit_variable

;------------------------------------------------------------------------------
; You may include more than one symbol in a PUBLIC statement.
;------------------------------------------------------------------------------
PUBLIC  data_variable1, code_table, typeless_num1, xdata_variable1

;------------------------------------------------------------------------------
; Put the STACK segment in the main module.
;------------------------------------------------------------------------------
?STACK          SEGMENT IDATA           ; ?STACK goes into IDATA RAM.
                RSEG    ?STACK          ; switch to ?STACK segment.
                DS      5               ; reserve your stack space
                                        ; 5 bytes in this example.
$EJECT
;------------------------------------------------------------------------------
; Put segment and variable declarations here.
;------------------------------------------------------------------------------

;------------------------------------------------------------------------------
; DATA SEGMENT--Reserves space in DATA RAM--Delete this segment if not used.
;------------------------------------------------------------------------------
data_seg_name   SEGMENT DATA            ; segment for DATA RAM.
                RSEG    data_seg_name   ; switch to this data segment
data_variable:  DS      1               ; reserve 1 Bytes for data_variable
data_variable1: DS      2               ; reserve 2 Bytes for data_variable1

;------------------------------------------------------------------------------
; XDATA SEGMENT--Reserves space in XDATA RAM--Delete this segment if not used.
;------------------------------------------------------------------------------
xdata_seg_name  SEGMENT XDATA           ; segment for XDATA RAM
                RSEG    xdata_seg_name  ; switch to this xdata segment
xdata_variable: DS      1               ; reserve 1 Bytes for xdata_variable
xdata_array:    DS      500             ; reserve 500 Bytes for xdata_array


;------------------------------------------------------------------------------
; ABSOLUTE XDATA SEGMENT--Reserves space in XDATA RAM at absolute addresses.
; ABSOLUTE segments are useful for memory mapped I/O.
; Delete this segment if not used.
```

```
;-------------------------------------------------------------------------------
                XSEG    AT 8000H        ; switch absolute XDATA segment @ 8000H
XIO:            DS      1               ; reserve 1 Bytes for XIO port
XCONFIG:        DS      1               ; reserve 1 Bytes for XCONFIG port

;-------------------------------------------------------------------------------
; BIT SEGMENT--Reserves space in BIT RAM--Delete segment if not used.
;-------------------------------------------------------------------------------
bit_seg_name    SEGMENT BIT             ; segment for BIT RAM.
                RSEG    bit_seg_name    ; switch to this bit segment
bit_variable:   DBIT    1               ; reserve 1 Bit for bit_variable
bit_variable1:  DBIT    4               ; reserve 4 Bits for bit_variable1

;-------------------------------------------------------------------------------
; Add constant (typeless) numbers here.
;-------------------------------------------------------------------------------
typeless_number EQU     0DH                     ; assign 0D hex
typeless_num1   EQU     typeless_number-8       ; evaluate typeless_num1

$EJECT
;-------------------------------------------------------------------------------
; Provide an LJMP to start at the reset address (address 0) in the main module.
; You may use this style for interrupt service routines.
;-------------------------------------------------------------------------------
                CSEG    AT      0       ; absolute Segment at Address 0
                LJMP    start           ; reset location (jump to start)

;-------------------------------------------------------------------------------
; CODE SEGMENT--Reserves space in CODE ROM for assembler instructions.
;-------------------------------------------------------------------------------
code_seg_name   SEGMENT CODE

                RSEG    code_seg_name   ; switch to this code segment
                USING   0               ; state register_bank used
                                        ; for the following program code.
start:          MOV     SP,#?STACK-1    ; assign stack at beginning

;-------------------------------------------------------------------------------
; Insert your assembly program here.  Note, the code below is non-functional.
;-------------------------------------------------------------------------------
                ORL     IE,#82H         ; enable interrupt system (timer 0)
                SETB    TR0             ; enable timer 0
repeat_label:   MOV     A,data_symbol
                ADD     A,#typeless_symbol
                CALL    code_symbol
                MOV     DPTR,#xdata_symbol
                MOVX    A,@DPTR
                MOV     R1,A
                PUSH    AR1
                CALL    sub_routine1
                POP     AR1
                ADD     A,R1
                JMP     repeat_label

code_entry:     CALL    code_symbol
                RET

code_table:     DW      repeat_label
                DW      code_entry
                DB      typeless_number
                DB      0
$EJECT
;-------------------------------------------------------------------------------
; To include an interrupt service routins, provide an LJMP to the ISR at the
; interrupt vector address.
;-------------------------------------------------------------------------------
                CSEG    AT  0BH         ; 0BH is address for Timer 0 interrupt
```

```
                LJMP    timer0int

;-------------------------------------------------------------------------------
; Give each interrupt function its own code segment.
;-------------------------------------------------------------------------------
int0_code_seg   SEGMENT CODE            ; segment for interrupt function
                RSEG    int0_code_seg   ; switch to this code segment
                USING   1               ; register bank for interrupt routine

timer0int:      PUSH    PSW
                MOV     PSW,#08H        ; register bank 1
                PUSH    ACC
                MOV     R1,data_variable
                MOV     DPTR,#xdata_variable
                MOVX    A,@DPTR
                ADD     A,R1
                MOV     data_variable1,A
                CLR     A
                ADD     A,#0
                MOV     data_variable1+1,A
                POP     ACC
                POP     PSW
                RETI

;-------------------------------------------------------------------------------
; The END directive is ALWAYS required.
;-------------------------------------------------------------------------------
                END             ; End Of File
```

INITIATION A MICROVISION

1 LE DEVELOPPEMENT SOUS µVISION4:

µvision4 est l'outil que vous utiliserez durant toutes ces séances pratiques pour développer des applications en assembleur pour les processeurs de la famille 8051. Avec cet outil, vous allez pouvoir gérer un cycle complet de développement, de l'écriture du code, jusqu'à la phase de mise au point du code.
A l'aide de cet outil vous allez donc pouvoir :

- Ecrire du code assembleur et l'assembler.
- Lier plusieurs fichiers source pour créer un seul fichier binaire exécutable et le simuler.
- Télécharger votre code sur une carte de test et l'exécuter en mode pas à pas ou avec des points d'arrêts.
- Tester votre code sur la carte de test en visualiser registres et mémoires.

Dans les paragraphes à suivre, il vous sera donné quelques précisions sur le fonctionnement de Uvision4, mais **nous vous encourageons vivement à faire preuve de curiosité** et à rechercher des informations dans l'aide en ligne.

Ce chapitre permet de découvrir les fonctionnalités majeures de Microvision4. Cette initiation se fait au travers d'un projet exemple ***Essai_Uvision4.uvproj*** et d'un code assembleur basique ***Demo_TP_SM1.asm.***
Ces 2 fichiers sont accessibles sur le E-campus.

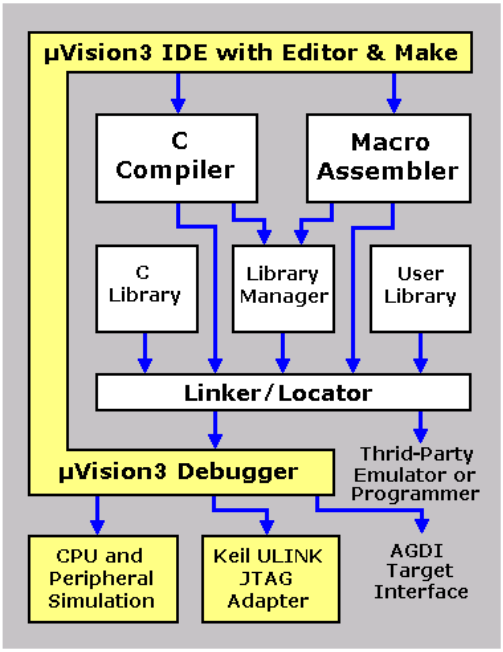
1.1 Les différentes étapes de développement sous µvision4 ?

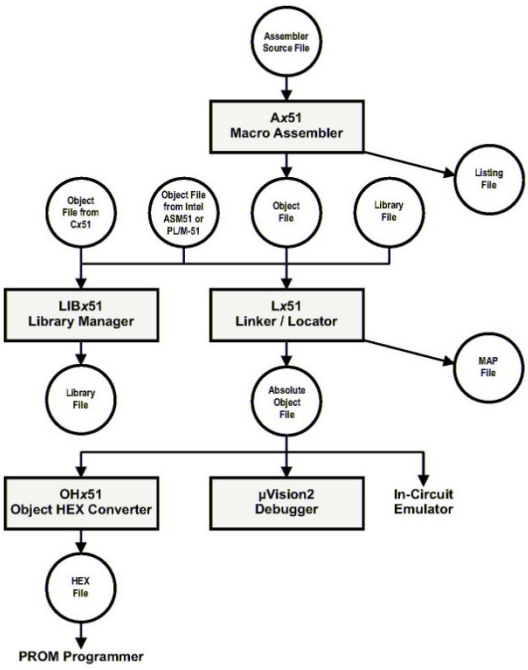
Software Development Cycle

When you use the Keil µVision3, the project development cycle is roughly the same as it is for any other software development project.

1. Create a project, select the target chip from the device database, and configure the tool settings.
2. Create source files in C or assembly.
3. Build your application with the project manager.
4. Correct errors in source files.
5. Test the linked application.

The following block diagram illustrates the complete µVision3 software development cycle. Each component is described below.



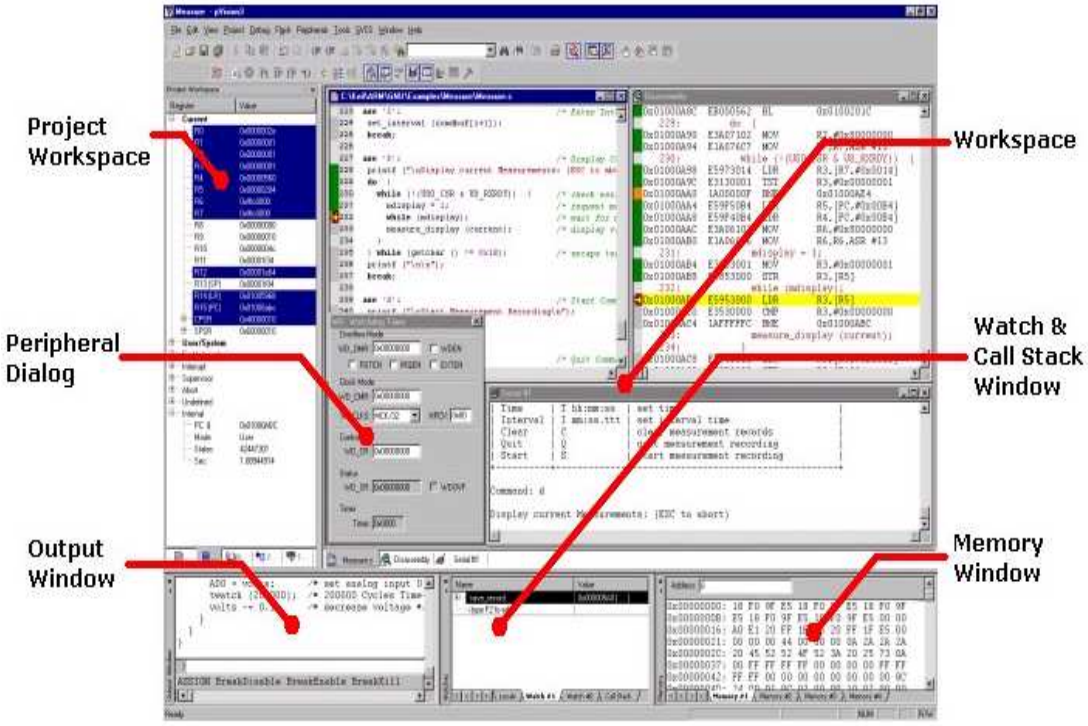


1.2 Les 2 modes de fonctionnement de µvision4 :

µVision3 has two operating modes:

- **Build Mode**: Allows you to translate all the application files and to generate executable programs. The features of the Build Mode are described under Creating Applications.
- **Debug Mode**: Provides you with a powerful debugger for testing your application. The Debug Mode is described in Testing Programs.

In both operating modes you may use the source editor of µVision3 to modify your source code. The Debug mode adds additional windows and stores an own screen layout. The following picture shows a typical configuration of µVision3 in the Debug Mode.



- The tabs of the **Project Workspace** give you access to:
  - **Files** and **Groups** of the project.
  - CPU **Registers** during debugging.
  - Tool and project specific on-line **Books**.
  - Text **Templates** for often used text blocks.
  - **Function** in the project for quick editor navigation.
- The tabs of the **Output Window** provides: **Build** messages and fast error access; Debug **Command** input/output console; **Find in Files** results with quick file access.
- The **Memory Window** gives access to the memory areas in display various formats.
- The **Watch & Call Stack Window** allows you to review and modify program variables and displays the current function call tree.
- The **Workspace** is used for the file editing, disassembly output, and other debug information.
- The **Peripheral Dialogs** help you to review the status of the on-chip peripherals in the microcontroller.

2 PRISE EN MAIN DE L'OUTIL µVISION4

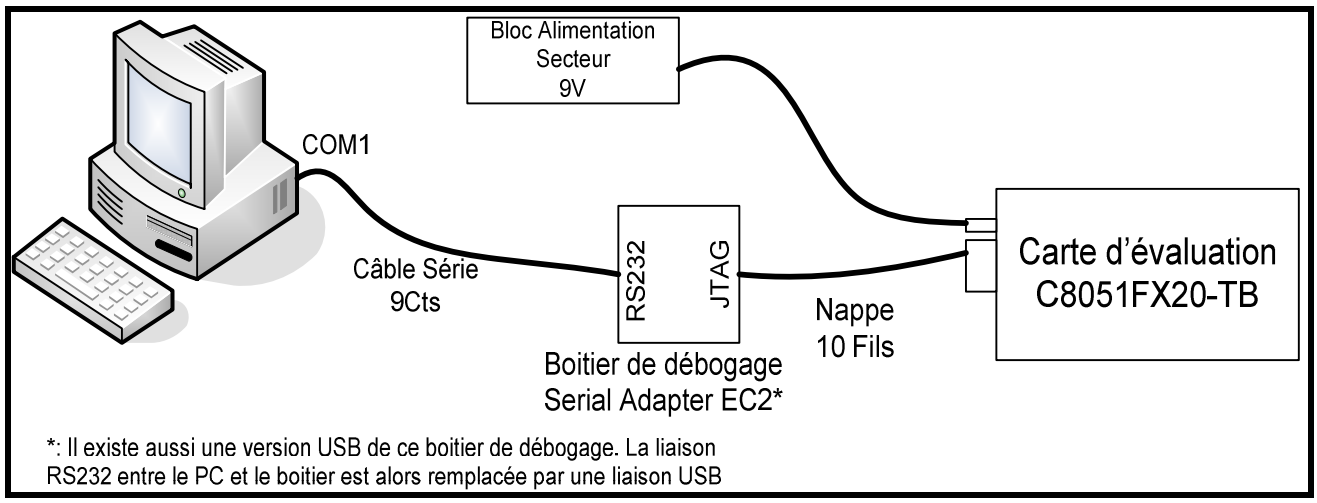
2.1 Démarrer sous µvision4 – Edition et assemblage

- Télécharger depuis le E-campus les fichiers ***Essai_Uvision4.uvproj*** **et** ***Demo_TP_SM1.asm*****.** et les placer dans votre dossier de travail :

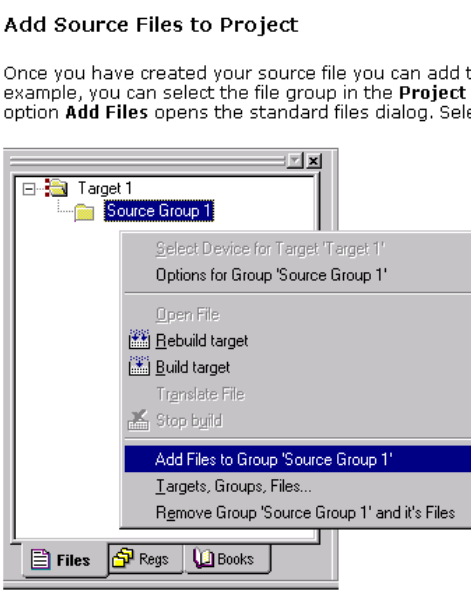
  - Le fichier ***Essai_Uvision4.uvproj*** est un fichier « projet », il configure l'outil Uvision4 pour ces séances de TPs. Il spécifie le type de processeur utilisé et configure 2 modes de débogage : simulation ou exécution du code sur la carte de développement. En simulation, tout est fait par logiciel sur le PC, et le fonctionnement du processeur est reproduit fidèlement. Mais les interactions avec le monde extérieur sont plus difficiles à reproduire. En exécution sur la carte de développement, le code est exécuté sur le processeur de la carte. Le contrôle de l'exécution du code, et le débogage sont assurés par l'intermédiaire d'une liaison JTAG entre le processeur et le PC. Dans ce mode, il est plus facile de placer le processeur dans des conditions réelles de fonctionnement et d'interagir avec l'environnement extérieur.

  - Le fichier ***Demo_TP_SM1.asm*** est un programme élémentaire chargé de faire clignoter la LED verte sur la carte de développement.

- Brancher la carte et le boîtier d'interface.

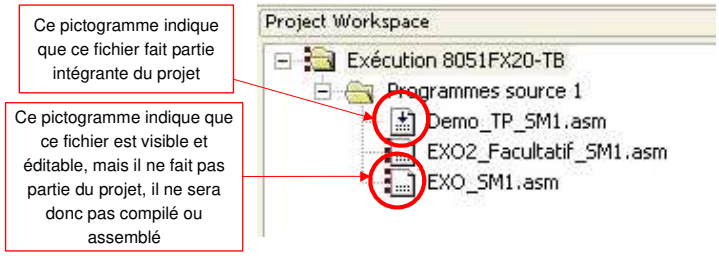


- Démarrer **µvision4**.
- Ouvrir le projet ***Essai_Uvision4*****.**
- Insérer le code source ***Demo_TP_SM1.asm*** dans le projet.
- Essayer de corriger et de comprendre le contenu du fichier ***Demo_TP_SM1.asm***



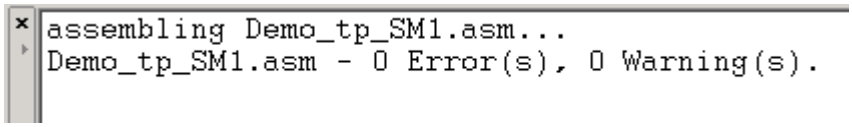
Si vous souhaitez créer un nouveau fichier assembleur, il vous faudra procéder en 2 étapes. D'abord, créer le fichier texte avec *File/new* et l'enregistrer dans le dossier du projet avec l'extension asm, puis ensuite l'insérer dans le projet comme indiqué dans l'illustration ci-contre.

- Après insertion du fichier ***Demo_TP_SM1.asm*** dans le projet, la fenêtre ***Project Workspace*** doit avoir cette allure :



- Assembler le fichier en utilisant la commande ***Build Target*.** Cette opération permet de créer un fichier contenant le code exécutable par le processeur.

- Vérifiez le bon déroulement de cette opération dans la fenêtre ***Output Window***. S'il y a des erreurs, corrigez les jusqu'à obtenir un résultat d'assemblage/édition de liens sans erreur.



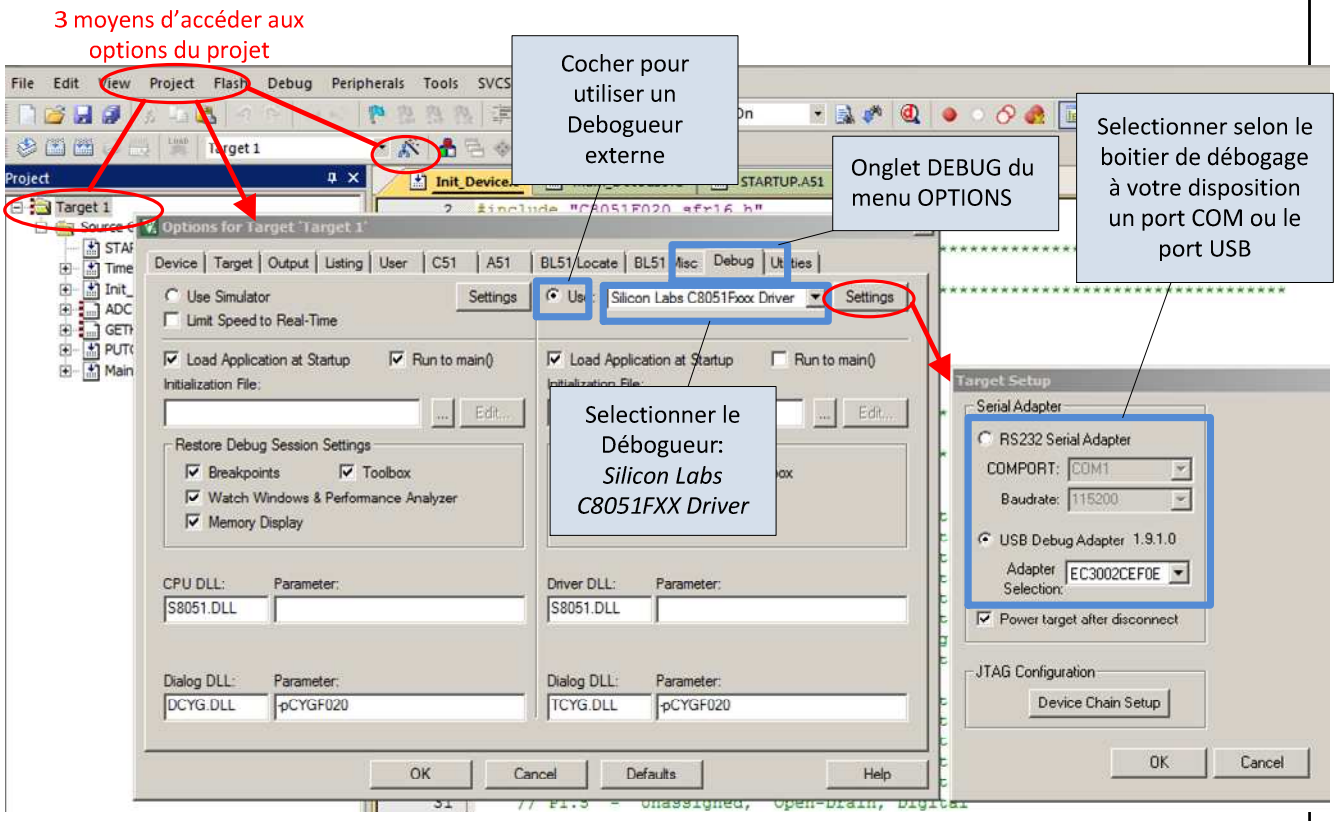
2.2 Le débogage sous µvision4.

Le programme, dorénavant assemblé sans erreur, peut maintenant être testé. **µvision4 nous donne la possibilité de tester en simulation ou en exécution sur la carte de développement**. Durant ces TPs, on utilisera le mode *Exécution 8051FX20-TB* (c'est-à-dire « exécution sur la carte de développement).
Pour information, en simulation, l'outil permet de simuler les signaux d'entrées- sorties et donne accès à des outils d'analyse plus poussée comme par exemple la couverture de code ou l'analyseur de signaux logiques.

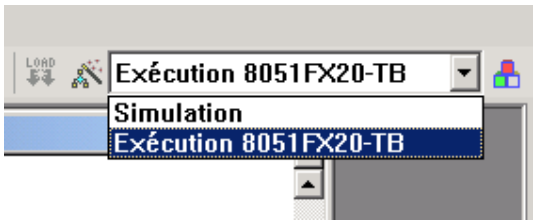
Mais avant de démarrer le débogage sur la carte de développement, il faut veiller à s'assurer que la configuration de la connexion du boitier de débogage avec le PC est correcte.
En effet, il existe 2 types de boitier de débogage, des boitiers à connecter sur un des ports Série RS232 du PC (COM1, COM2 etc….), et des boitiers à connecter sur les ports USB du PC.
Il faut donc informer MicroVision du port sur lequel est connecté le boitier de décodage.

Cette configuration se fait dans le menu « options » du projet.

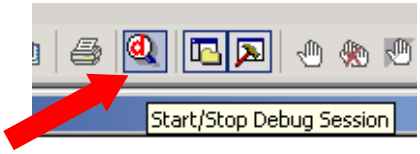
Configuration de la connexion Boitier de débogage <-> PC



- Choisir le mode ***Exécution 8051FX20-TB***.

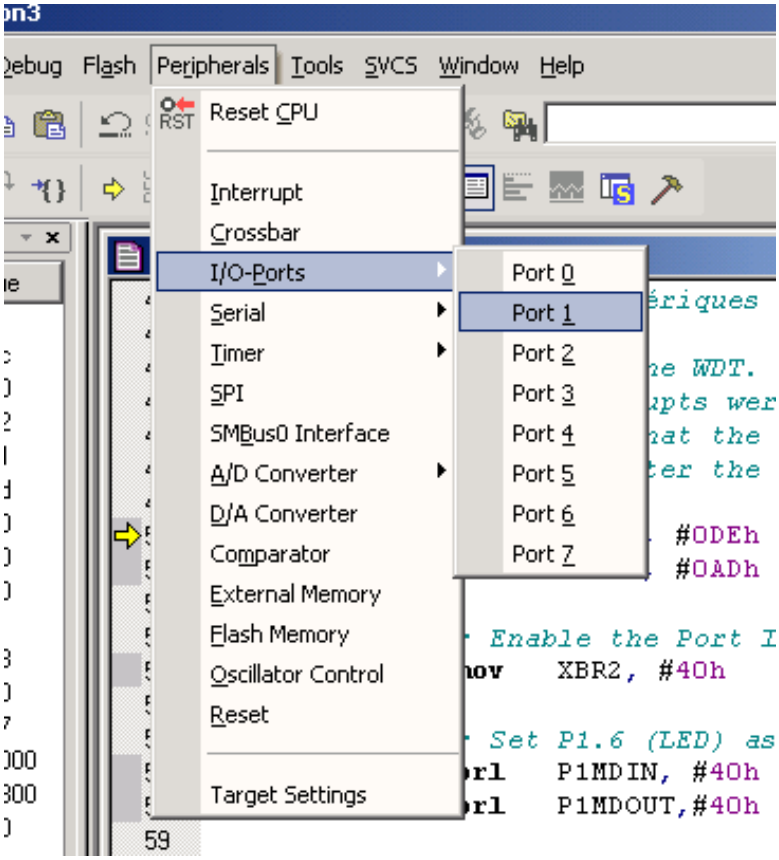


- Démarrer le mode débogage.

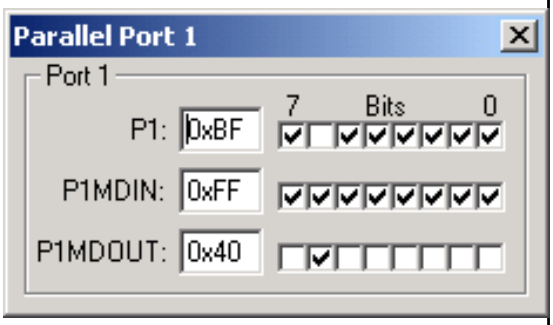


De nouvelles fenêtres apparaissent alors. Elles ont été définies lors de la création de ce projet, rien ne vous empêche de les modifier, de les redimensionner, de les supprimer, ou d'en ajouter d'autres.

Par exemple on peut vouloir afficher les registres de pilotage du port 1, pour visualiser l'état de P1.6, le port qui commande la LED verte sur la carte d'évaluation.

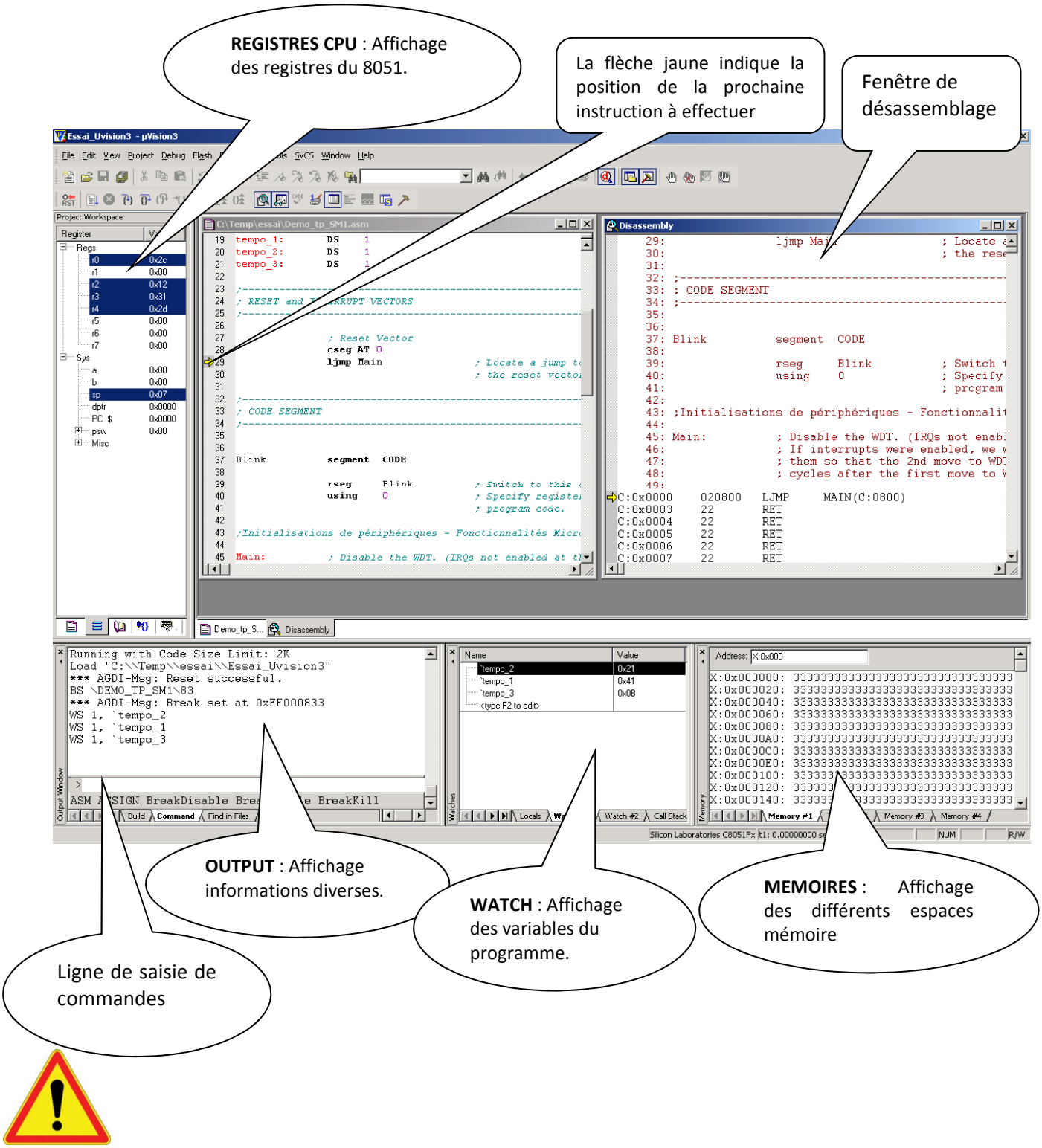


On obtient alors la fenêtre :



Département **ETI - IRC**
23/04/2019 16:04

µvision4 en mode débogage.

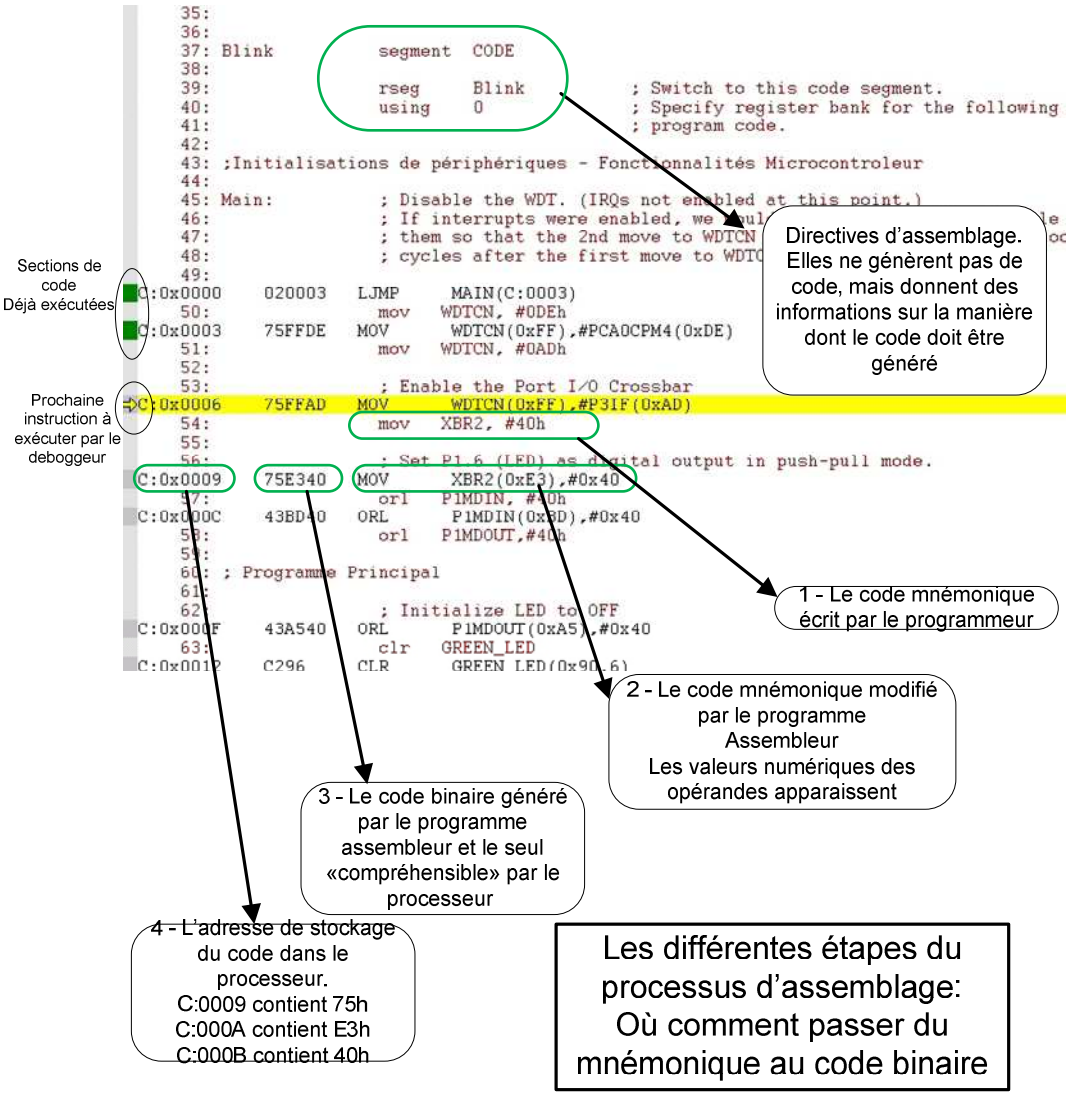


ATTENTION : ces fenêtres permettent de visualiser le contenu de mémoires et de registres, mais à tout moment, il vous est aussi possible de modifier le contenu de ces mémoires et registres.

2.2.1 La fenêtre de désassemblage.

Cette fenêtre peut être affichée avec la commande ***View/Disassembly Window***. Elle contient votre code assembleur sous forme de mnémoniques, mais aussi le code binaire généré par le programme assembleur. Ce code est chargé dans la mémoire CODE du microprocesseur, c'est le seul code compréhensible par le processeur.

Analyse d'une fenêtre de désassemblage



2.2.2 Visualisation des différents espaces-mémoire.

Pour visualiser les différent espaces-mémoire, utiliser la syntaxe suivante :

- C:0x0000, pour visualiser la mémoire CODE à partir de l'adresse 0
- X:0x0000, pour visualiser la mémoire XDATA à partir de l'adresse 0
- D:0x00, pour visualiser la mémoire DATA **directe** à partir de l'adresse 0
- I:0x00, pour visualiser la mémoire DATA **indirecte** à partir de l'adresse 0. (DATA directe et indirecte sont confondues de 00 à 7Fh, mais disjointes de 80h à FFh).

Pour vous «entrainer », essayez de retrouver le code votre programme en le visualisant dans une fenêtre ***Memory***

Pour exécuter le code, on dispose de plusieurs possibilités, à utiliser selon les besoins :

- Exécution en pas à pas, instruction par instruction.
- Exécution en mode curseur.
- Exécution avec points d'arrêt.
- Exécution en continu.

2.2.3 Exécution en pas à pas.

- Avant de commencer à exécuter, faites une remise à zéro du système avec ***Reset CPU***.

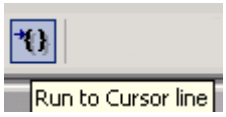
- Dans votre fichier source, la flèche jaune indiquant la prochaine instruction à exécuter doit pointer sur la ligne **`LJMP Code_Test`**.

- Exécuter en mode pas à pas le programme, en cliquant sur ***Step into*** *(*les codes assembleurs placés entre les étiquettes **`Code_Test`** et **`DEB`** sont indispensables, mais ne seront pas étudiés durant ce TP)

- A partir de l'étiquette **`DEB`**, vous pourrez étudier l'évolution du contenu des registres à l'aide de la fenêtre « Registers »(Pour observer le contenu des registres utiliser l'onglet **View**, **Registers Window)**

2.2.4 Exécution en mode curseur.

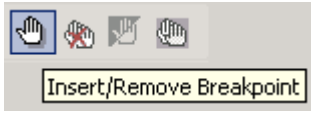
- A partir de l'étiquette **`BCL_Tempo`**, on entre dans une boucle de temporisation, et l'exécution en mode pas à pas peut prendre beaucoup de temps. Pour sortir de la boucle, placer le curseur sur la ligne **`cpl GREEN_LED,`** et utilisez ***Run to cursor line***



- Le débogueur exécute ainsi toutes les instructions, jusqu'à atteindre la ligne où a été positionné le curseur. Vérifiez l'évolution du contenu des registres, de la fenêtre ***watch*** et de la mémoire XDATA (X:0x0000).
- Placer le curseur sur la ligne **`djnz R0,BCL_RAZ_X`** et utilisez de nouveau la commande ***Run to cursor line***. Vérifiez de nouveau l'évolution du contenu des registres, de la fenêtre ***watch*** et de la mémoire XDATA (X:0x0000).
- En réutilisant plusieurs fois consécutivement la commande ***Run to cursor Line***, on parvient ainsi, à parcourir, une boucle complète de programme, et de visualiser les modifications de registres et mémoire à chaque sortie de boucle.

2.2.5 Exécution avec points d'arrêt.

- Par cette méthode, on va pouvoir fixer des conditions d'arrêt sur plusieurs lignes de code. (4 points d'arrêt au maximum). En lançant une exécution de code en continu, le débogeur va automatiquement stopper son exécution avant chaque une ligne de code avec point d'arrêt.
- Se replacer au début du programme en faisant une remise à zéro (Reset CPU).
- Placer un point d'arrêt sur la ligne **`djnz R0,RAZ_RD_WR`** et un point d'arrêt sur la ligne **`djnz R0,BCL_RAZ_X`** .



- Lancer l'exécution en continu avec la commande ***Run***

- Le programme va alors automatiquement stopper son exécution sur les lignes de code contenant un point d'arrêt.
- Relancer plusieurs fois l'exécution et vérifier l'évolution des registres et des mémoires.

2.2.6 Exécution en continu, en temps réel.

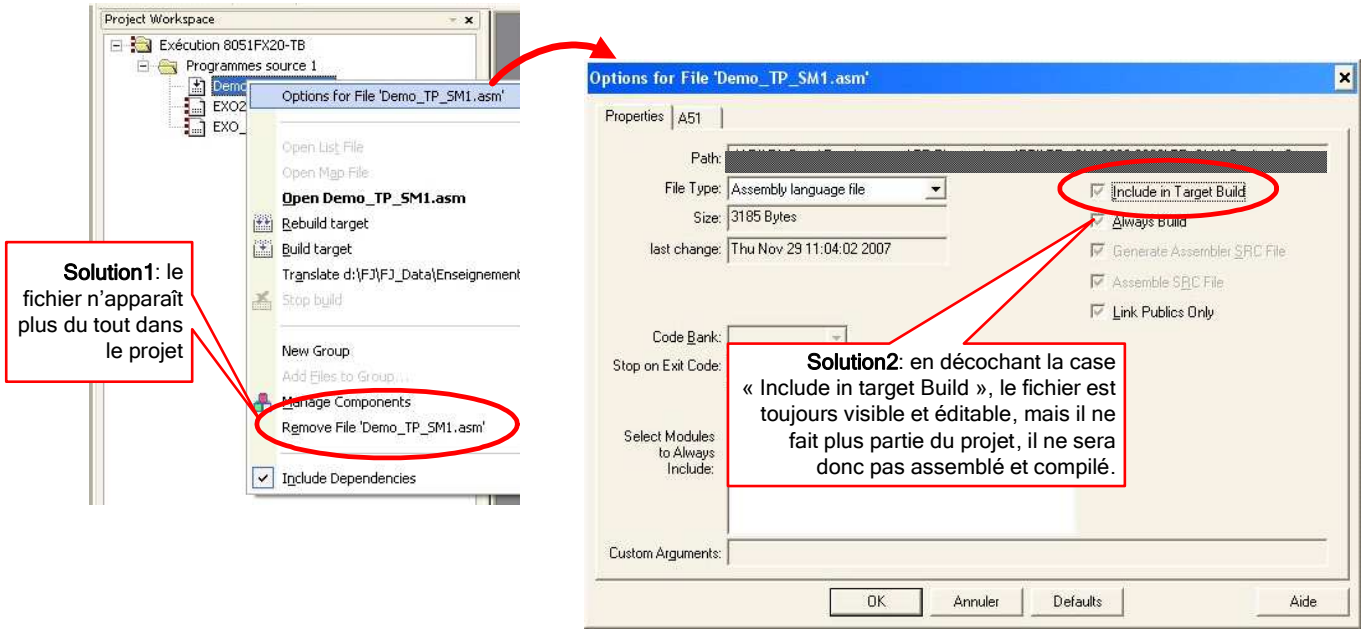
- Pour terminer, après avoir enlevé les points d'arrêt et avoir effectué un reset, une exécution de ***Run***, permet d'obtenir un fonctionnement continu en temps réel du programme, on peut alors observer le clignotement de la LED verte à une fréquence proche de 1 Hz.

> NB : pour réinitialiser des zones mémoire, utiliser la commande `MEMSET` dans la zone de saisie de commandes de débogage.
>
> Syntaxe : `MEMSET` (adresse de départ, taille, valeur)
>
> Exemple : MEMSET `(X :0000h,100,11h)` initialisation de la zone mémoire XDATA de l'adresse 0 à l'adresse (0h+100h) avec la valeur 11h.

3 MANIPULATION, PARTIE 2 : ECRITURE DE PROGRAMMES EN ASSEMBLEUR.

Après être sorti du mode *Debug*, insérer le fichier ***Exercices_SM1.asm*** dans le projet, **tout en enlevant le fichier *Demo_tp_SM1.asm*.**

Pour enlever un fichier du projet: Sélectionner le fichier et « Clic -Droit » avec la souris; Puis 2 solutions se présentent:



> **Précision** : en base 16, on rajoute le suffixe H, en base 2, on ajoute le suffixe B.
> **Piège** : en assembleur, toute valeur numérique doit commencer par un chiffre, aussi on écrit 0FFH au lieu de FFH

ACCES MEMOIRE EXTERNE 8051F020

Ce chapitre, décrit toute la partie interfaçage mémoire du 8051F020.
Il y est décrit notamment :

- La description des signaux de gestion du bus.
- Les Timings du Bus Mémoire
- Les registres utilisés pour configurer les accès mémoire.

16. EXTERNAL DATA MEMORY INTERFACE AND ON-CHIP XRAM

The C8051F020/1/2/3 MCUs include 4k bytes of on-chip RAM mapped into the external data memory space (XRAM), as well as an External Data Memory Interface which can be used to access off-chip memories and memory-mapped devices connected to the GPIO ports. The external memory space may be accessed using the external move instruction (MOVX) and the data pointer (DPTR), or using the MOVX indirect addressing mode using R0 or R1. If the MOVX instruction is used with an 8-bit address operand (such as @R1), then the high byte of the 16-bit address is provided by the External Memory Interface Control Register (EMI0CN, shown in Figure 16.1). Note: the MOVX instruction can also be used for writing to the FLASH memory. See **Section “15. FLASH MEMORY” on page 139** for details. The MOVX instruction accesses XRAM by default. The EMIF can be configured to appear on the lower I/O ports (P0-P3) or the upper I/O ports (P4-P7).

16.1. Accessing XRAM

The XRAM memory space is accessed using the MOVX instruction. The MOVX instruction has two forms, both of which use an indirect addressing method. The first method uses the Data Pointer, DPTR, a 16-bit register which contains the effective address of the XRAM location to be read or written. The second method uses R0 or R1 in combination with the EMI0CN register to generate the effective XRAM address. Examples of both of these methods are given below.

16.1.1. 16-Bit MOVX Example

The 16-bit form of the MOVX instruction accesses the memory location pointed to by the contents of the DPTR register. The following series of instructions reads the value of the byte at address 0x1234 into the accumulator A:

```
MOV    DPTR, #1234h        ; load DPTR with 16-bit address to read (0x1234)
MOVX   A, @DPTR            ; load contents of 0x1234 into accumulator A
```

The above example uses the 16-bit immediate MOV instruction to set the contents of DPTR. Alternately, the DPTR can be accessed through the SFR registers DPH, which contains the upper 8-bits of DPTR, and DPL, which contains the lower 8-bits of DPTR.

16.2. Configuring the External Memory Interface

Configuring the External Memory Interface consists of four steps:

1. Select EMIF on Low Ports (P3, P2, P1, and P0) or High Ports (P7, P6, P5, and P4).
2. Select Multiplexed mode or Non-multiplexed mode.
3. Select the memory mode (on-chip only, split mode without bank select, split mode with bank select, or off-chip only).
4. Set up timing to interface with off-chip memory or peripherals.
5. Select the desired output mode for the associated Ports (registers PnMDOUT, P74OUT).

Each of these four steps is explained in detail in the following sections. The Port selection, Multiplexed mode selection, and Mode bits are located in the EMI0CF register shown in Figure 16.2.

16.3. Port Selection and Configuration

The External Memory Interface can appear on Ports 3, 2, 1, and 0 (C8051F020/1/2/3 devices) or on Ports 7, 6, 5, and 4 (C8051F020/2 devices only), depending on the state of the PRTSEL bit (EMI0CF.5). If the lower Ports are selected, the EMIFLE bit (XBR2.1) must be set to a '1' so that the Crossbar will skip over P0.7 (/WR), P0.6 (/RD), and if multiplexed mode is selected P0.5 (ALE). For more information about the configuring the Crossbar, see **Section "17. PORT INPUT/OUTPUT" on page 161**.

The External Memory Interface claims the associated Port pins for memory operations ONLY during the execution of an off-chip MOVX instruction. Once the MOVX instruction has completed, control of the Port pins reverts to the Port latches or to the Crossbar (on Ports 3, 2, 1, and 0). See **Section "17. PORT INPUT/OUTPUT" on page 161** for more information about the Crossbar and Port operation and configuration. **The Port latches should be explicitly configured to 'park' the External Memory Interface pins in a dormant state, most commonly by setting them to a logic 1**.

During the execution of the MOVX instruction, the External Memory Interface will explicitly disable the drivers on all Port pins that are acting as Inputs (Data[7:0] during a READ operation, for example). The Output mode of the Port pins (whether the pin is configured as Open-Drain or Push-Pull) is unaffected by the External Memory Interface operation, and remains controlled by the PnMDOUT registers. See **Section "17. PORT INPUT/OUTPUT" on page 161** for more information about Port output mode configuration.

Figure 16.1. EMI0CN: External Memory Interface Control

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
PGSEL7	PGSEL6	PGSEL5	PGSEL4	PGSEL3	PGSEL2	PGSEL1	PGSEL0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xAF

Bits7-0: PGSEL[7:0]: XRAM Page Select Bits.
The XRAM Page Select Bits provide the high byte of the 16-bit external data memory address when using an 8-bit MOVX command, effectively selecting a 256-byte page of RAM.
0x00: 0x0000 to 0x00FF
0x01: 0x0100 to 0x01FF
...
0xFE: 0xFE00 to 0xFEFF
0xFF: 0xFF00 to 0xFFFF

Figure 16.2. EMI0CF: External Memory Configuration

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
-	-	PRTSEL	EMD2	EMD1	EMD0	EALE1	EALE0	00000011
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xA3

Bits7-6: Unused. Read = 00b. Write = don’t care.
Bit5: PRTSEL: EMIF Port Select.
0: EMIF active on P0-P3.
1: EMIF active on P4-P7.
Bit4: EMD2: EMIF Multiplex Mode Select.
0: EMIF operates in multiplexed address/data mode.
1: EMIF operates in non-multiplexed mode (separate address and data pins).
Bits3-2: EMD1-0: EMIF Operating Mode Select.
These bits control the operating mode of the External Memory Interface.
00: Internal Only: MOVX accesses on-chip XRAM only. All effective addresses alias to on-chip memory space.
01: Split Mode without Bank Select: Accesses below the 4k boundary are directed on-chip. Accesses above the 4k boundary are directed off-chip. 8-bit off-chip MOVX operations use the current contents of the Address High port latches to resolve upper address byte. Note that in order to access off-chip space, EMI0CN must be set to a page that is not contained in the on-chip address space.
10: Split Mode with Bank Select: Accesses below the 4k boundary are directed on-chip. Accesses above the 4k boundary are directed off-chip. 8-bit off-chip MOVX operations use the contents of EMI0CN to determine the high-byte of the address.
11: External Only: MOVX accesses off-chip XRAM only. On-chip XRAM is not visible to the CPU.
Bits1-0: EALE1-0: ALE Pulse-Width Select Bits (only has effect when EMD2 = 0).
00: ALE high and ALE low pulse width = 1 SYSCLK cycle.
01: ALE high and ALE low pulse width = 2 SYSCLK cycles.
10: ALE high and ALE low pulse width = 3 SYSCLK cycles.
11: ALE high and ALE low pulse width = 4 SYSCLK cycles.

16.4. Multiplexed and Non-multiplexed Selection

The External Memory Interface is capable of acting in a Multiplexed mode or a Non-multiplexed mode, depending on the state of the EMD2 (EMI0CF.4) bit.

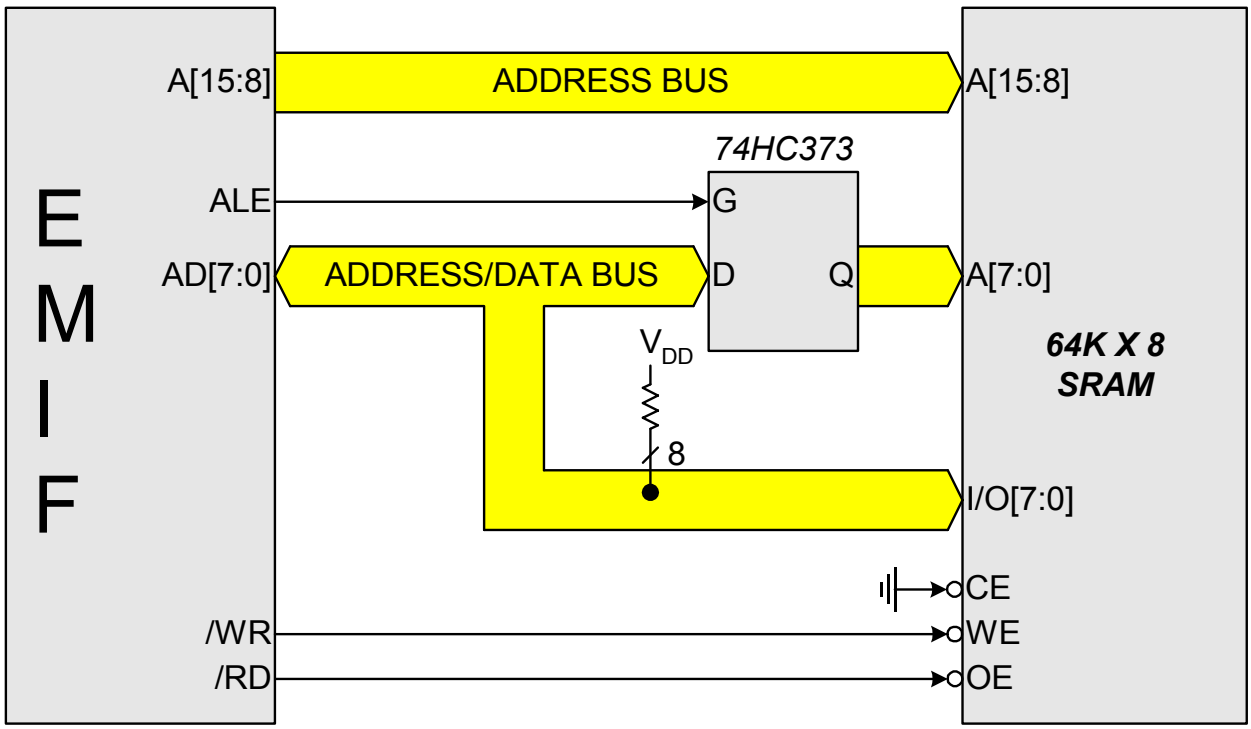
16.4.1. Multiplexed Configuration

In Multiplexed mode, the Data Bus and the lower 8-bits of the Address Bus share the same Port pins: AD[7:0]. In this mode, an external latch (74HC373 or equivalent logic gate) is used to hold the lower 8-bits of the RAM address. The external latch is controlled by the ALE (Address Latch Enable) signal, which is driven by the External Memory Interface logic. An example of a Multiplexed Configuration is shown in Figure 16.3.

In Multiplexed mode, the external MOVX operation can be broken into two phases delineated by the state of the ALE signal. During the first phase, ALE is high and the lower 8-bits of the Address Bus are presented to AD[7:0]. During this phase, the address latch is configured such that the ‘Q’ outputs reflect the states of the ‘D’ inputs. When ALE falls, signaling the beginning of the second phase, the address latch outputs remain fixed and are no longer dependent on the latch inputs. Later in the second phase, the Data Bus controls the state of the AD[7:0] port at the time /RD or /WR is asserted.

See **Section “16.6.2. Multiplexed Mode” on page 156** for more information.

Figure 16.3. Multiplexed Configuration Example



16.5. Memory Mode Selection

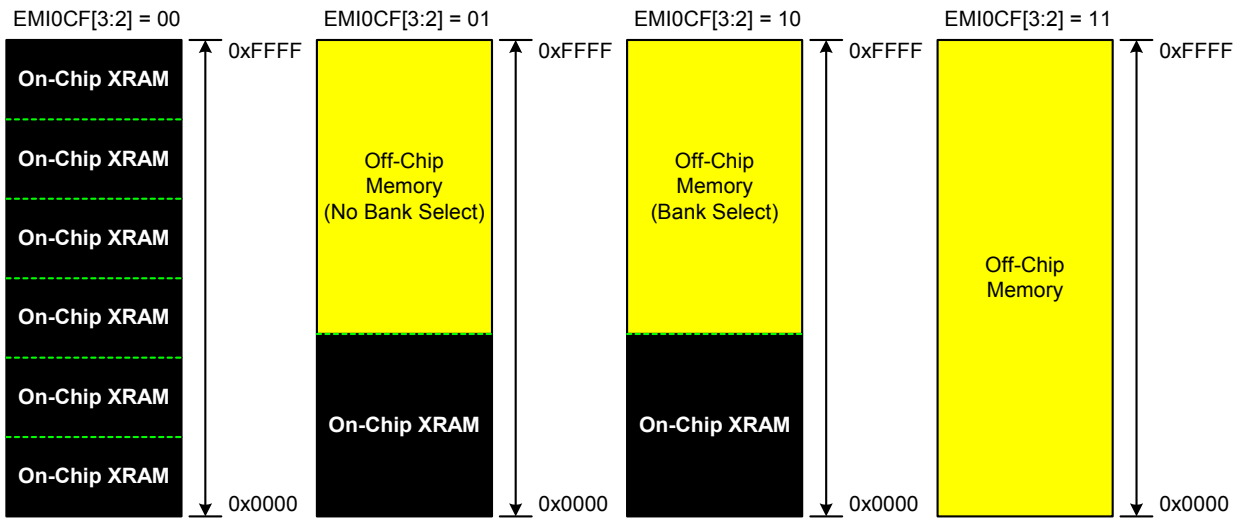
The external data memory space can be configured in one of four modes, shown in Figure 16.5, based on the EMIF Mode bits in the EMI0CF register (Figure 16.2). These modes are summarized below. More information about the different modes can be found in **Section " ." on page 152**.

16.5.2. Split Mode without Bank Select

When EMI0CF.[3:2] are set to '01', the XRAM memory map is split into two areas, on-chip space and off-chip space.

- Effective addresses below the 4k boundary will access on-chip XRAM space.
- Effective addresses beyond the 4k boundary will access off-chip space.
- 8-bit MOVX operations use the contents of EMI0CN to determine whether the memory access is on-chip or off-chip. The lower 8-bits of the Address Bus A[7:0] are driven as defined by R0 or R1. However, in the "No Bank Select" mode, an 8-bit MOVX operation will not drive the upper 8-bits A[15:8] of the Address Bus during an off-chip access. This allows the user to manipulate the upper address bits at will by setting the Port state directly. This behavior is in contrast with "Split Mode with Bank Select" described below.
- 16-bit MOVX operations use the contents of DPTR to determine whether the memory access is on-chip or off-chip, and unlike 8-bit MOVX operations, the full 16-bits of the Address Bus A[15:0] are driven during the off-chip transaction.

Figure 16.5. EMIF Operating Modes



16.6. Timing

The timing parameters of the External Memory Interface can be configured to enable connection to devices having different setup and hold time requirements. The Address Setup time, Address Hold time, /RD and /WR strobe widths, and in multiplexed mode, the width of the ALE pulse are all programmable in units of SYSCLK periods through EMI0TC, shown in Figure 16.6, and EMI0CF[1:0].

The timing for an off-chip MOVX instruction can be calculated by adding 4 SYSCLK cycles to the timing parameters defined by the EMI0TC register. Assuming non-multiplexed operation, the minimum execution time for an off-chip XRAM operation is 5 SYSCLK cycles (1 SYSCLK for /RD or /WR pulse + 4 SYSCLKs). For multiplexed operations, the Address Latch Enable signal will require a minimum of 2 additional SYSCLK cycles. Therefore, the minimum execution time of an off-chip XRAM operation in multiplexed mode is 7 SYSCLK cycles (2 SYSCLKs for /ALE, 1 for /RD or /WR + 4 SYSCLKs). The programmable setup and hold times default to the maximum delay settings after a reset.

Table 16.1 lists the AC parameters for the External Memory Interface, and Figure 16.7 through Figure 16.11 show the timing diagrams for the different External Memory Interface modes and MOVX operations

.

Figure 16.6. EMI0TC: External Memory Timing Control

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
EAS1	EAS0	EWR3	EWR2	EWR1	EWR0	EAH1	EAH0	11111111
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xA1

Bits7-6: EAS1-0: EMIF Address Setup Time Bits.
00: Address setup time = 0 SYSCLK cycles.
01: Address setup time = 1 SYSCLK cycle.
10: Address setup time = 2 SYSCLK cycles.
11: Address setup time = 3 SYSCLK cycles.

Bits5-2: EWR3-0: EMIF /WR and /RD Pulse-Width Control Bits.
0000: /WR and /RD pulse width = 1 SYSCLK cycle.
0001: /WR and /RD pulse width = 2 SYSCLK cycles.
0010: /WR and /RD pulse width = 3 SYSCLK cycles.
0011: /WR and /RD pulse width = 4 SYSCLK cycles.
0100: /WR and /RD pulse width = 5 SYSCLK cycles.
0101: /WR and /RD pulse width = 6 SYSCLK cycles.
0110: /WR and /RD pulse width = 7 SYSCLK cycles.
0111: /WR and /RD pulse width = 8 SYSCLK cycles.
1000: /WR and /RD pulse width = 9 SYSCLK cycles.
1001: /WR and /RD pulse width = 10 SYSCLK cycles.
1010: /WR and /RD pulse width = 11 SYSCLK cycles.
1011: /WR and /RD pulse width = 12 SYSCLK cycles.
1100: /WR and /RD pulse width = 13 SYSCLK cycles.
1101: /WR and /RD pulse width = 14 SYSCLK cycles.
1110: /WR and /RD pulse width = 15 SYSCLK cycles.
1111: /WR and /RD pulse width = 16 SYSCLK cycles.

Bits1-0: EAH1-0: EMIF Address Hold Time Bits.
00: Address hold time = 0 SYSCLK cycles.
01: Address hold time = 1 SYSCLK cycle.
10: Address hold time = 2 SYSCLK cycles.
11: Address hold time = 3 SYSCLK cycles.

16.6.2. Multiplexed Mode

16.6.2.1. 16-bit MOVX: EMI0CF[4:2] = ‘001’, ‘010’, or ‘011’.

Figure 16.10. Multiplexed 16-bit MOVX Timing

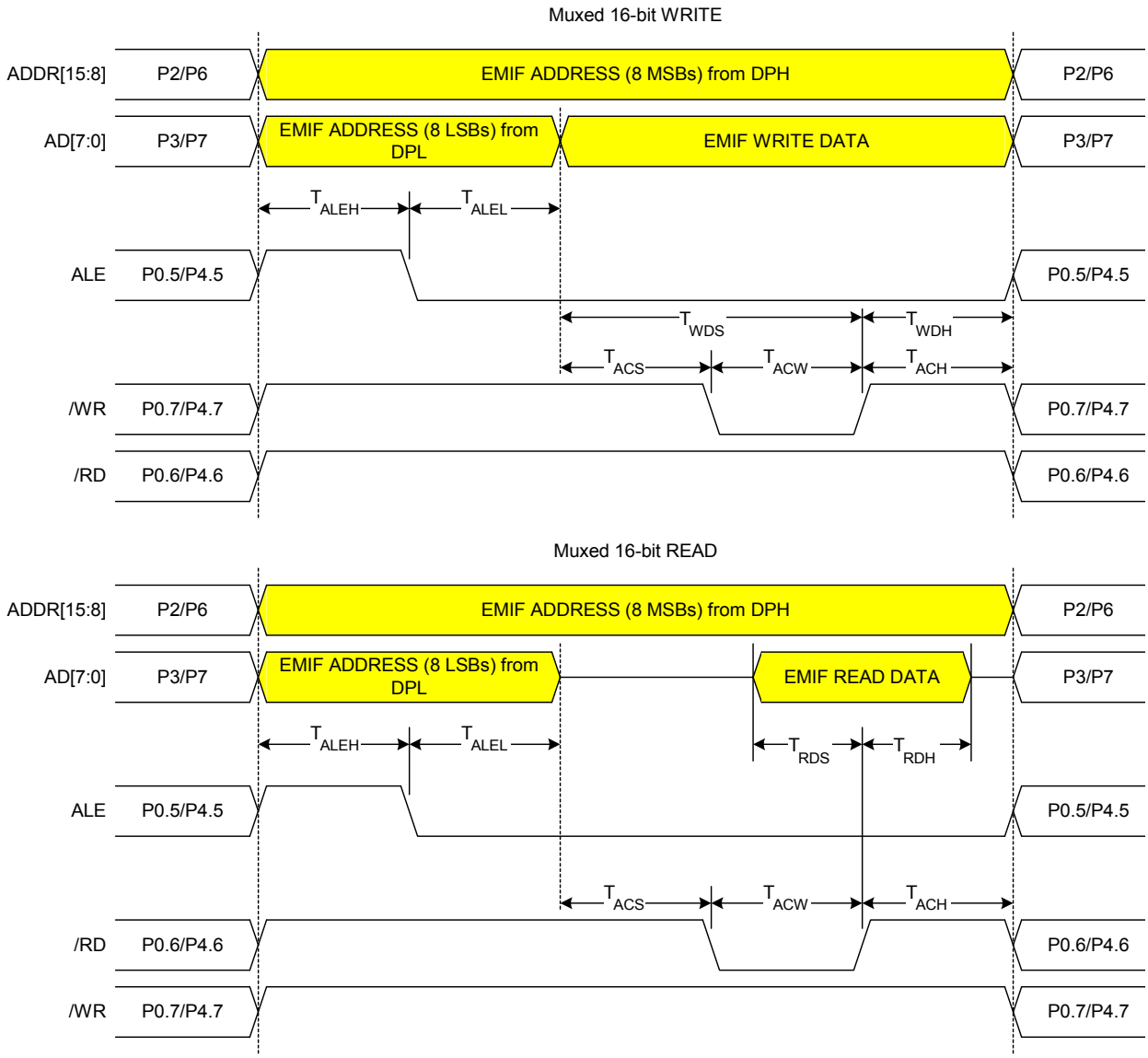


Table 16.1. AC Parameters for External Memory Interface

PARAMETER	DESCRIPTION	MIN	MAX	UNITS
T_{SYSCLK}	System Clock Period	40		ns
T_{ACS}	Address / Control Setup Time	0	$3*T_{SYSCLK}$	ns
T_{ACW}	Address / Control Pulse Width	$1*T_{SYSCLK}$	$16*T_{SYSCLK}$	ns
T_{ACH}	Address / Control Hold Time	0	$3*T_{SYSCLK}$	ns
T_{ALEH}	Address Latch Enable High Time	$1*T_{SYSCLK}$	$4*T_{SYSCLK}$	ns
T_{ALEL}	Address Latch Enable Low Time	$1*T_{SYSCLK}$	$4*T_{SYSCLK}$	ns
T_{WDS}	Write Data Setup Time	$1*T_{SYSCLK}$	$19*T_{SYSCLK}$	ns
T_{WDH}	Write Data Hold Time	0	$3*T_{SYSCLK}$	ns
T_{RDS}	Read Data Setup Time	20		ns
T_{RDH}	Read Data Hold Time	0		ns

INTERRUPTIONS SUR LE 8051F020

Ce chapitre, donne toute la documentation relative à la gestion générale des interruptions et à la gestion des interruptions externes INT0 et INT1.
Il y est décrit notamment :

- La description des interruptions externes.
- Le mécanisme de validation des interruptions et la gestion des priorités
- Les registres utilisés pour configurer les interruptions et plus particulièrement les interruptions INT0 et INT1

12.3. Interrupt Handler

The CIP-51 includes an extended interrupt system supporting a total of 22 interrupt sources with two priority levels. The allocation of interrupt sources between on-chip peripherals and external inputs pins varies according to the specific version of the device. Each interrupt source has one or more associated interrupt-pending flag(s) located in an SFR. When a peripheral or external source meets a valid interrupt condition, the associated interrupt-pending flag is set to logic 1.

If interrupts are enabled for the source, an interrupt request is generated when the interrupt-pending flag is set. As soon as execution of the current instruction is complete, the CPU generates an LCALL to a predetermined address to begin execution of an interrupt service routine (ISR). Each ISR must end with an RETI instruction, which returns program execution to the next instruction that would have been executed if the interrupt request had not occurred. If interrupts are not enabled, the interrupt-pending flag is ignored by the hardware and program execution continues as normal. (The interrupt-pending flag is set to logic 1 regardless of the interrupt's enable/disable state.)

Each interrupt source can be individually enabled or disabled through the use of an associated interrupt enable bit in an SFR (IE-EIE2). However, interrupts must first be globally enabled by setting the EA bit (IE.7) to logic 1 before the individual interrupt enables are recognized. Setting the EA bit to logic 0 disables all interrupt sources regardless of the individual interrupt-enable settings.

Some interrupt-pending flags are automatically cleared by the hardware when the CPU vectors to the ISR. However, most are not cleared by the hardware and must be cleared by software before returning from the ISR. If an interrupt-pending flag remains set after the CPU completes the return-from-interrupt (RETI) instruction, a new interrupt request will be generated immediately and the CPU will re-enter the ISR after the completion of the next instruction.

12.3.1. MCU Interrupt Sources and Vectors

The MCUs support 22 interrupt sources. Software can simulate an interrupt event by setting any interrupt-pending flag to logic 1. If interrupts are enabled for the flag, an interrupt request will be generated and the CPU will vector to the ISR address associated with the interrupt-pending flag. MCU interrupt sources, associated vector addresses, priority order and control bits are summarized in Table 12.4. Refer to the datasheet section associated with a particular on-chip peripheral for information regarding valid interrupt conditions for the peripheral and the behavior of its interrupt-pending flag(s).

12.3.2. External Interrupts

Two of the external interrupt sources (/INT0 and /INT1) are configurable as active-low level-sensitive or active-low edge-sensitive inputs depending on the setting of bits IT0 (TCON.0) and IT1 (TCON.2). IE0 (TCON.1) and IE1 (TCON.3) serve as the interrupt-pending flag for the /INT0 and /INT1 external interrupts, respectively. If an /INT0 or /INT1 external interrupt is configured as edge-sensitive, the corresponding interrupt-pending flag is automatically cleared by the hardware when the CPU vectors to the ISR. When configured as level sensitive, the interrupt-pending flag follows the state of the external interrupt's input pin. The external interrupt source must hold the input active until the interrupt request is recognized. It must then deactivate the interrupt request before execution of the ISR completes or another interrupt request will be generated.

The remaining 2 external interrupts (External Interrupts 6-7) are edge-sensitive inputs configurable as active-low or active-high. The interrupt-pending flags and configuration bits for these interrupts are in the Port 3 Interrupt Flag Register shown in **Figure “17.19 P3IF: Port3 Interrupt Flag Register” on page 177**.

Table 12.4. Interrupt Summary

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag	Bit addressable?	Cleared by HW?	Enable Flag	Priority Control
Reset	0x0000	Top	None	N/A	N/A	Always Enabled	Always Highest
External Interrupt 0 (/INT0)	0x0003	0	IE0 (TCON.1)	Y	Y	EX0 (IE.0)	PX0 (IP.0)
Timer 0 Overflow	0x000B	1	TF0 (TCON.5)	Y	Y	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (/INT1)	0x0013	2	IE1 (TCON.3)	Y	Y	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	0x001B	3	TF1 (TCON.7)	Y	Y	ET1 (IE.3)	PT1 (IP.3)
UART0	0x0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	Y		ES0 (IE.4)	PS0 (IP.4)
Timer 2 Overflow (or EXF2)	0x002B	5	TF2 (T2CON.7)	Y		ET2 (IE.5)	PT2 (IP.5)
Serial Peripheral Interface	0x0033	6	SPIF (SPI0CN.7)	Y		ESPI0 (EIE1.0)	PSPI0 (EIP1.0)
SMBus Interface	0x003B	7	SI (SMB0CN.3)	Y		ESMB0 (EIE1.1)	PSMB0 (EIP1.1)
ADC0 Window Comparator	0x0043	8	AD0WINT (ADC0CN.2)	Y		EWADC0 (EIE1.2)	PWADC0 (EIP1.2)
Programmable Counter Array	0x004B	9	CF (PCA0CN.7) CCFn (PCA0CN.n)	Y		EPCA0 (EIE1.3)	PPCA0 (EIP1.3)
Comparator 0 Falling Edge	0x0053	10	CP0FIF (CPT0CN.4)			ECP0F (EIE1.4)	PCP0F (EIP1.4)
Comparator 0 Rising Edge	0x005B	11	CP0RIF (CPT0CN.5)			ECP0R (EIE1.5)	PCP0R (EIP1.5)
Comparator 1 Falling Edge	0x0063	12	CP1FIF (CPT1CN.4)			ECP1F (EIE1.6)	PCP1F (EIP1.6)
Comparator 1 Rising Edge	0x006B	13	CP1RIF (CPT1CN.5)			ECP1R (EIE1.7)	PCP1F (EIP1.7)
Timer 3 Overflow	0x0073	14	TF3 (TMR3CN.7)			ET3 (EIE2.0)	PT3 (EIP2.0)
ADC0 End of Conversion	0x007B	15	AD0INT (ADC0CN.5)	Y		EADC0 (EIE2.1)	PADC0 (EIP2.1)
Timer 4 Overflow	0x0083	16	TF4 (T4CON.7)			ET4 (EIE2.2)	PT4 (EIP2.2)
ADC1 End of Conversion	0x008B	17	AD1INT (ADC1CN.5)			EADC1 (EIE2.3)	PADC1 (EIP2.3)
External Interrupt 6	0x0093	18	IE6 (P3IF.5)			EX6 (EIE2.4)	PX6 (EIP2.4)
External Interrupt 7	0x009B	19	IE7 (P3IF.6)			EX7 (EIE2.5)	PX7 (EIP2.5)
UART1	0x00A3	20	RI1 (SCON1.0) TI1 (SCON1.1)			ES1	PS1
External Crystal OSC Ready	0x00AB	21	XTLVLD (OSCXCN.7)			EXVLD (EIE2.7)	PXVLD (EIP2.7)

12.3.3. Interrupt Priorities

Each interrupt source can be individually programmed to one of two priority levels: low or high. A low priority interrupt service routine can be preempted by a high priority interrupt. A high priority interrupt cannot be preempted. Each interrupt has an associated interrupt priority bit in an SFR (IP-EIP2) used to configure its priority level. Low priority is the default. If two interrupts are recognized simultaneously, the interrupt with the higher priority is serviced first. If both interrupts have the same priority level, a fixed priority order is used to arbitrate, given in Table 12.4.

12.3.4. Interrupt Latency

Interrupt response time depends on the state of the CPU when the interrupt occurs. Pending interrupts are sampled and priority decoded each system clock cycle. Therefore, the fastest possible response time is 5 system clock cycles: 1 clock cycle to detect the interrupt and 4 clock cycles to complete the LCALL to the ISR. If an interrupt is pending when a RETI is executed, a single instruction is executed before an LCALL is made to service the pending interrupt. Therefore, the maximum response time for an interrupt (when no other interrupt is currently being serviced or the new interrupt is of greater priority) occurs when the CPU is performing an RETI instruction followed by a DIV as the next instruction. In this case, the response time is 18 system clock cycles: 1 clock cycle to detect the interrupt, 5 clock cycles to execute the RETI, 8 clock cycles to complete the DIV instruction and 4 clock cycles to execute the LCALL to the ISR. If the CPU is executing an ISR for an interrupt with equal or higher priority, the new interrupt will not be serviced until the current ISR completes, including the RETI and following instruction.

Figure 17.19. P3IF: Port3 Interrupt Flag Register

R/W	R/W	R	R	R/W	R/W	R/W	R/W	Reset Value
IE7	IE6	-	-	IE7CF	IE6CF	-	-	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xAD

Bit7: IE7: External Interrupt 7 Pending Flag
0: No falling edge has been detected on P3.7 since this bit was last cleared.
1: This flag is set by hardware when a falling edge on P3.7 is detected.
Bit6: IE6: External Interrupt 6 Pending Flag
0: No falling edge has been detected on P3.6 since this bit was last cleared.
1: This flag is set by hardware when a falling edge on P3.6 is detected.
Bits5-4: UNUSED. Read = 00b, Write = don't care.
Bit3: IE7CF: External Interrupt 7 Edge Configuration
0: External Interrupt 7 triggered by a falling edge on the IE7 input.
1: External Interrupt 7 triggered by a rising edge on the IE7 input.
Bit2: IE6CF: External Interrupt 6 Edge Configuration
0: External Interrupt 6 triggered by a falling edge on the IE6 input.
1: External Interrupt 6 triggered by a rising edge on the IE6 input.
Bits1-0: UNUSED. Read = 00b, Write = don't care.

17.2. Ports 4 through 7 (C8051F020/2 only)

All Port pins on Ports 4 through 7 can be accessed as General-Purpose I/O (GPIO) pins by reading and writing the associated Port Data registers (See Figure 17.21, Figure 17.22, Figure 17.23, and Figure 17.24), a set of SFRs which are byte-addressable.

A Read of a Port Data register (or Port bit) will always return the logic state present at the pin itself, regardless of whether the Crossbar has allocated the pin for peripheral use or not. An exception to this occurs during the execution of a *read-modify-write* instruction (ANL, ORL, XRL, CPL, INC, DEC, DJNZ, JBC, CLR, SET, and the bitwise MOV operation). During the *read* cycle of the *read-modify-write* instruction, it is the contents of the Port Data register, not the state of the Port pins themselves, which is read.

17.2.1. Configuring Ports which are not Pinned Out

Although P4, P5, P6, and P7 are not brought out to pins on the C8051F021/3 devices, the Port Data registers are still present and can be used by software. Because the digital input paths also remain active, it is recommended that these pins not be left in a 'floating' state in order to avoid unnecessary power dissipation arising from the inputs floating to non-valid logic levels. This condition can be prevented by any of the following:

1. Leave the weak pull-up devices enabled by setting WEAKPUD (XBR2.7) to a logic 0.
2. Configure the output modes of P4, P5, P6, and P7 to "Push-Pull" by writing P74OUT = 0xFF.
3. Force the output states of P4, P5, P6, and P7 to logic 0 by writing zeros to the Port Data registers: P4 = 0x00, P5 = 0x00, P6= 0x00, and P7 = 0x00.

17.2.2. Configuring the Output Modes of the Port Pins

The output mode of each port pin can be configured to be either Open-Drain or Push-Pull. In the Push-Pull configuration, a logic 0 in the associated bit in the Port Data register will cause the Port pin to be driven to GND, and a logic 1 will cause the Port pin to be driven to VDD. In the Open-Drain configuration, a logic 0 in the associated bit in the

12.3.5. Interrupt Register Descriptions

The SFRs used to enable the interrupt sources and set their priority level are described below. Refer to the datasheet section associated with a particular on-chip peripheral for information regarding valid interrupt conditions for the peripheral and the behavior of its interrupt-pending flag(s).

Figure 12.9. IE: Interrupt Enable

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
EA	IEGF0	ET2	ES0	ET1	EX1	ET0	EX0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0xA8

Bit7: EA: Enable All Interrupts.
This bit globally enables/disables all interrupts. When set to ‘0’, individual interrupt mask settings are overridden.
0: Disable all interrupt sources.
1: Enable each interrupt according to its individual mask setting.

Bit6: IEGF0: General Purpose Flag 0.
This is a general purpose flag for use under software control.

Bit5: ET2: Enabler Timer 2 Interrupt.
This bit sets the masking of the Timer 2 interrupt.
0: Disable Timer 2 interrupt.
1: Enable interrupt requests generated by the TF2 flag (T2CON.7).

Bit4: ES0: Enable UART0 Interrupt.
This bit sets the masking of the UART0 interrupt.
0: Disable UART0 interrupt.
1: Enable UART0 interrupt.

Bit3: ET1: Enable Timer 1 Interrupt.
This bit sets the masking of the Timer 1 interrupt.
0: Disable all Timer 1 interrupt.
1: Enable interrupt requests generated by the TF1 flag (TCON.7).

Bit2: EX1: Enable External Interrupt 1.
This bit sets the masking of external interrupt 1.
0: Disable external interrupt 1.
1: Enable interrupt requests generated by the /INT1 pin.

Bit1: ET0: Enable Timer 0 Interrupt.
This bit sets the masking of the Timer 0 interrupt.
0: Disable all Timer 0 interrupt.
1: Enable interrupt requests generated by the TF0 flag (TCON.5).

Bit0: EX0: Enable External Interrupt 0.
This bit sets the masking of external interrupt 0.
0: Disable external interrupt 0.
1: Enable interrupt requests generated by the /INT0 pin.

Figure 12.12. EIE2: Extended Interrupt Enable 2

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
EXVLD	ES1	EX7	EX6	EADC1	ET4	EADC0	ET3	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE7

Bit7: EXVLD: Enable External Clock Source Valid (XTLVLD) Interrupt.
This bit sets the masking of the XTLVLD interrupt.
0: Disable XTLVLD interrupt.
1: Enable interrupt requests generated by the XTLVLD flag (OSCXCN.7)

Bit6: ES1: Enable UART1 Interrupt.
This bit sets the masking of the UART1 interrupt.
0: Disable UART1 interrupt.
1: Enable UART1 interrupt.

Bit5: EX7: Enable External Interrupt 7.
This bit sets the masking of External Interrupt 7.
0: Disable External Interrupt 7.
1: Enable interrupt requests generated by the External Interrupt 7 input pin.

Bit4: EX6: Enable External Interrupt 6.
This bit sets the masking of External Interrupt 6.
0: Disable External Interrupt 6.
1: Enable interrupt requests generated by the External Interrupt 6 input pin.

Bit3: EADC1: Enable ADC1 End Of Conversion Interrupt.
This bit sets the masking of the ADC1 End of Conversion interrupt.
0: Disable ADC1 End of Conversion interrupt.
1: Enable interrupt requests generated by the ADC1 End of Conversion Interrupt.

Bit2: ET4: Enable Timer 4 Interrupt
This bit sets the masking of the Timer 4 interrupt.
0: Disable Timer 4 interrupt.
1: Enable interrupt requests generated by the TF4 flag (T4CON.7).

Bit1: EADC0: Enable ADC0 End of Conversion Interrupt.
This bit sets the masking of the ADC0 End of Conversion Interrupt.
0: Disable ADC0 Conversion Interrupt.
1: Enable interrupt requests generated by the ADC0 Conversion Interrupt.

Bit0: ET3: Enable Timer 3 Interrupt.
This bit sets the masking of the Timer 3 interrupt.
0: Disable all Timer 3 interrupts.
1: Enable interrupt requests generated by the TF3 flag (TMR3CN.7).

Figure 12.10. IP: Interrupt Priority

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
-	-	PT2	PS0	PT1	PX1	PT0	PX0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0 (bit addressable)	SFR Address: 0xB8

Bits7-6: UNUSED. Read = 11b, Write = don't care.
Bit5: PT2: Timer 2 Interrupt Priority Control.
This bit sets the priority of the Timer 2 interrupt.
0: Timer 2 interrupt priority determined by default priority order.
1: Timer 2 interrupts set to high priority level.
Bit4: PS0: UART0 Interrupt Priority Control.
This bit sets the priority of the UART0 interrupt.
0: UART0 interrupt priority determined by default priority order.
1: UART0 interrupts set to high priority level.
Bit3: PT1: Timer 1 Interrupt Priority Control.
This bit sets the priority of the Timer 1 interrupt.
0: Timer 1 interrupt priority determined by default priority order.
1: Timer 1 interrupts set to high priority level.
Bit2: PX1: External Interrupt 1 Priority Control.
This bit sets the priority of the External Interrupt 1 interrupt.
0: External Interrupt 1 priority determined by default priority order.
1: External Interrupt 1 set to high priority level.
Bit1: PT0: Timer 0 Interrupt Priority Control.
This bit sets the priority of the Timer 0 interrupt.
0: Timer 0 interrupt priority determined by default priority order.
1: Timer 0 interrupt set to high priority level.
Bit0: PX0: External Interrupt 0 Priority Control.
This bit sets the priority of the External Interrupt 0 interrupt.
0: External Interrupt 0 priority determined by default priority order.
1: External Interrupt 0 set to high priority level.

Figure 12.14. EIP2: Extended Interrupt Priority 2

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
PXVLD	EP1	PX7	PX6	PADC1	PT4	PADC0	PT3	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xF7

Bit7: PXVLD: External Clock Source Valid (XTLVLD) Interrupt Priority Control.
This bit sets the priority of the XTLVLD interrupt.
0: XTLVLD interrupt set to low priority level.
1: XTLVLD interrupt set to high priority level.

Bit6: EP1: UART1 Interrupt Priority Control.
This bit sets the priority of the UART1 interrupt.
0: UART1 interrupt set to low priority.
1: UART1 interrupt set to high priority.

Bit5: PX7: External Interrupt 7 Priority Control.
This bit sets the priority of the External Interrupt 7.
0: External Interrupt 7 set to low priority level.
1: External Interrupt 7 set to high priority level.

Bit4: PX6: External Interrupt 6 Priority Control.
This bit sets the priority of the External Interrupt 6.
0: External Interrupt 6 set to low priority level.
1: External Interrupt 6 set to high priority level.

Bit3: PADC1: ADC1 End Of Conversion Interrupt Priority Control.
This bit sets the priority of the ADC1 End of Conversion interrupt.
0: ADC1 End of Conversion interrupt set to low priority.
1: ADC1 End of Conversion interrupt set to low priority.

Bit2: PT4: Timer 4 Interrupt Priority Control.
This bit sets the priority of the Timer 4 interrupt.
0: Timer 4 interrupt set to low priority.
1: Timer 4 interrupt set to low priority.

Bit1: PADC0: ADC End of Conversion Interrupt Priority Control.
This bit sets the priority of the ADC0 End of Conversion Interrupt.
0: ADC0 End of Conversion interrupt set to low priority level.
1: ADC0 End of Conversion interrupt set to high priority level.

Bit0: PT3: Timer 3 Interrupt Priority Control.
This bit sets the priority of the Timer 3 interrupts.
0: Timer 3 interrupt priority determined by default priority order.
1: Timer 3 interrupt set to high priority level.

Figure 22.5. TCON: Timer Control Register

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
							(bit addressable)	0x88

Bit7: TF1: Timer 1 Overflow Flag.
Set by hardware when Timer 1 overflows. This flag can be cleared by software but is automatically cleared when the CPU vectors to the Timer 1 interrupt service routine.
0: No Timer 1 overflow detected.
1: Timer 1 has overflowed.

Bit6: TR1: Timer 1 Run Control.
0: Timer 1 disabled.
1: Timer 1 enabled.

Bit5: TF0: Timer 0 Overflow Flag.
Set by hardware when Timer 0 overflows. This flag can be cleared by software but is automatically cleared when the CPU vectors to the Timer 0 interrupt service routine.
0: No Timer 0 overflow detected.
1: Timer 0 has overflowed.

Bit4: TR0: Timer 0 Run Control.
0: Timer 0 disabled.
1: Timer 0 enabled.

Bit3: IE1: External Interrupt 1.
This flag is set by hardware when an edge/level of type defined by IT1 is detected. It can be cleared by software but is automatically cleared when the CPU vectors to the External Interrupt 1 service routine if IT1 = 1. This flag is the inverse of the /INT1 input signal's logic level when IT1 = 0.

Bit2: IT1: Interrupt 1 Type Select.
This bit selects whether the configured /INT1 signal will detect falling edge or active-low level-sensitive interrupts.
0: /INT1 is level triggered.
1: /INT1 is edge triggered.

Bit1: IE0: External Interrupt 0.
This flag is set by hardware when an edge/level of type defined by IT0 is detected. It can be cleared by software but is automatically cleared when the CPU vectors to the External Interrupt 0 service routine if IT0 = 1. This flag is the inverse of the /INT0 input signal's logic level when IT0 = 0.

Bit0: IT0: Interrupt 0 Type Select.
This bit selects whether the configured /INT0 signal will detect falling edge or active-low level-sensitive interrupts.
0: /INT0 is level triggered.
1: /INT0 is edge triggered.

Département **ETI - IRC**
03/04/2017 16:21

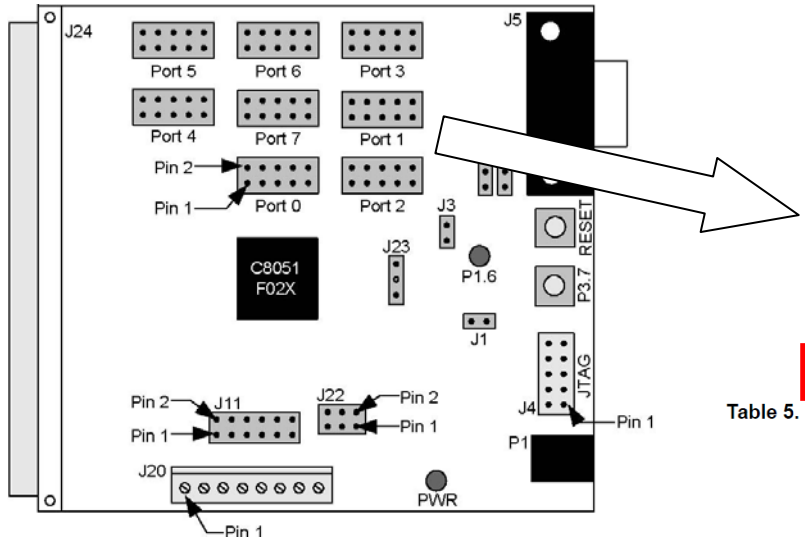
CARTE D'EVALUATION 8051F020TB

1 CARTE C8051F020DK

The C8051F02x Development Kit includes a target board with a C8051F020 device pre-installed for evaluation and preliminary software development. Numerous input/output (I/O) connections are provided to facilitate prototyping using the target board. Refer to Figure 2 for the locations of the various I/O connectors.

P1	Power connector	(accepts input from 7 to 15 VDC unregulated power adapter)
J1	Connects SW2 to P3.7 pin	
J3	Connects LED D3 to P1.6 pin	
J4	JTAG connector for Serial Adapter interface	
J5	DB-9 connector for UART0 RS232 interface	
J6	Jumper to connect UART0 TX	(P0.0) to DB9
J9	Jumper to connect UART0 RX (P0.1) to DB9	
J11	Analog loopback connector	
J12-J19	Port 0 - 7 connectors	
J20	Analog I/O terminal block	
J22	VREF connector	
J23	VDD Monitor Disable	
J24	96-pin Expansion I/O connector	

1.1 C8051F020 Target Board



Pin #	Description
1	Pn.0
2	Pn.1
3	Pn.2
4	Pn.3
5	Pn.4
6	Pn.5
7	Pn.6
8	Pn.7
9	+3 VD (+3.3 VDC)
10	GND (Ground)

Table 5. J12- J19 Port Connector Pin Descriptions

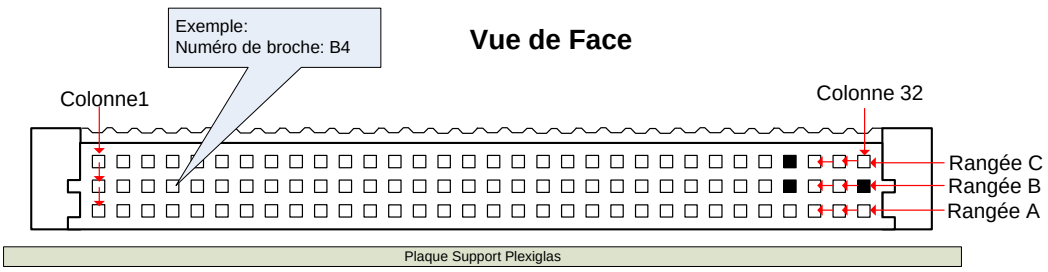
Département **ETI - IRC**
03/04/2017 16:21

Brochage du Connecteur DIN41612 de la carte 8051F020

Pin #	Description	Pin #	Description	Pin #	Description
A-1	+3 VD2 (+3.3 VDC)	B-1	DGND (Digital Gnd)	C-1	XTAL1
A-2	MONEN	B-2	P1.7	C-2	P1.6
A-3	P1.5	B-3	P1.4	C-3	P1.3
A-4	P1.2	B-4	P1.1	C-4	P1.0
A-5	P2.7	B-5	P2.6	C-5	P2.5
A-6	P2.4	B-6	P2.3	C-6	P2.2
A-7	P2.1	B-7	P2.0	C-7	P3.7
A-8	P3.6	B-8	P3.5	C-8	P3.4
A-9	P3.3	B-9	P3.2	C-9	P3.1
A-10	P3.0	B-10	P0.7	C-10	P0.6
A-11	P0.5	B-11	P0.4	C-11	P0.3
A-12	P0.2	B-12	P0.1	C-12	P0.0
A-13	P7.7	B-13	P7.6	C-13	P7.5
A-14	P7.4	B-14	P7.3	C-14	P7.2
A-15	P7.1	B-15	P7.0	C-15	P6.7
A-16	P6.6	B-16	P6.5	C-16	P6.4
A-17	P6.3	B-17	P6.2	C-17	P6.1
A-18	P6.0	B-18	P5.7	C-18	P5.6
A-19	P5.5	B-19	P5.4	C-19	P5.3
A-20	P5.2	B-20	P5.1	C-20	P5.0
A-21	P4.7	B-21	P4.6	C-21	P4.5
A-22	P4.4	B-22	P4.3	C-22	P4.2
A-23	P4.1	B-23	P4.0	C-23	TMS
A-24	TCK	B-24	TDI	C-24	TDO
A-25	/RST	B-25	DGND (Digital Gnd)	C-25	VUNREG
A-26	AGND (Analog Gnd)	B-26	DAC1	C-26	DAC0
A-27	CP1-	B-27	CP1+	C-27	CP0-
A-28	CP0+	B-28	VREF	C-28	VREFD
A-29	VREF0	B-29	VREF1	C-29	AIN0.7
A-30	AIN0.6	B-30	AIN0.5	C-30	AIN0.4
A-31	AIN0.3	B-31	AIN0.2	C-31	AIN0.1
A-32	AIN0.0	B-32	AGND (Analog Gnd)	C-32	AV+ (+3.3 VDC Analog)

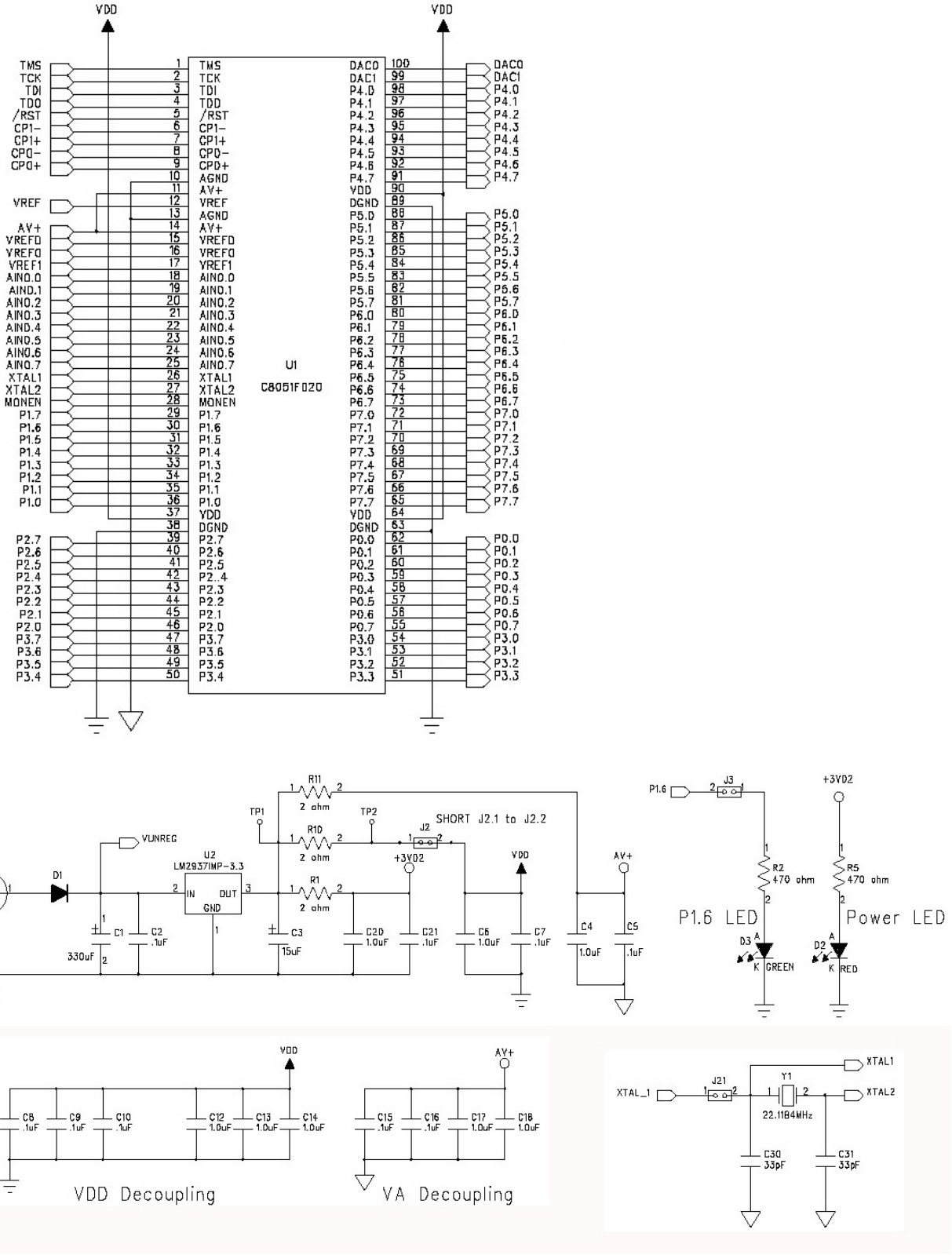
Table 6. J24 Pin Descriptions

Brochage du connecteur D41612 96 Cts de la carte C8051FX20-TB

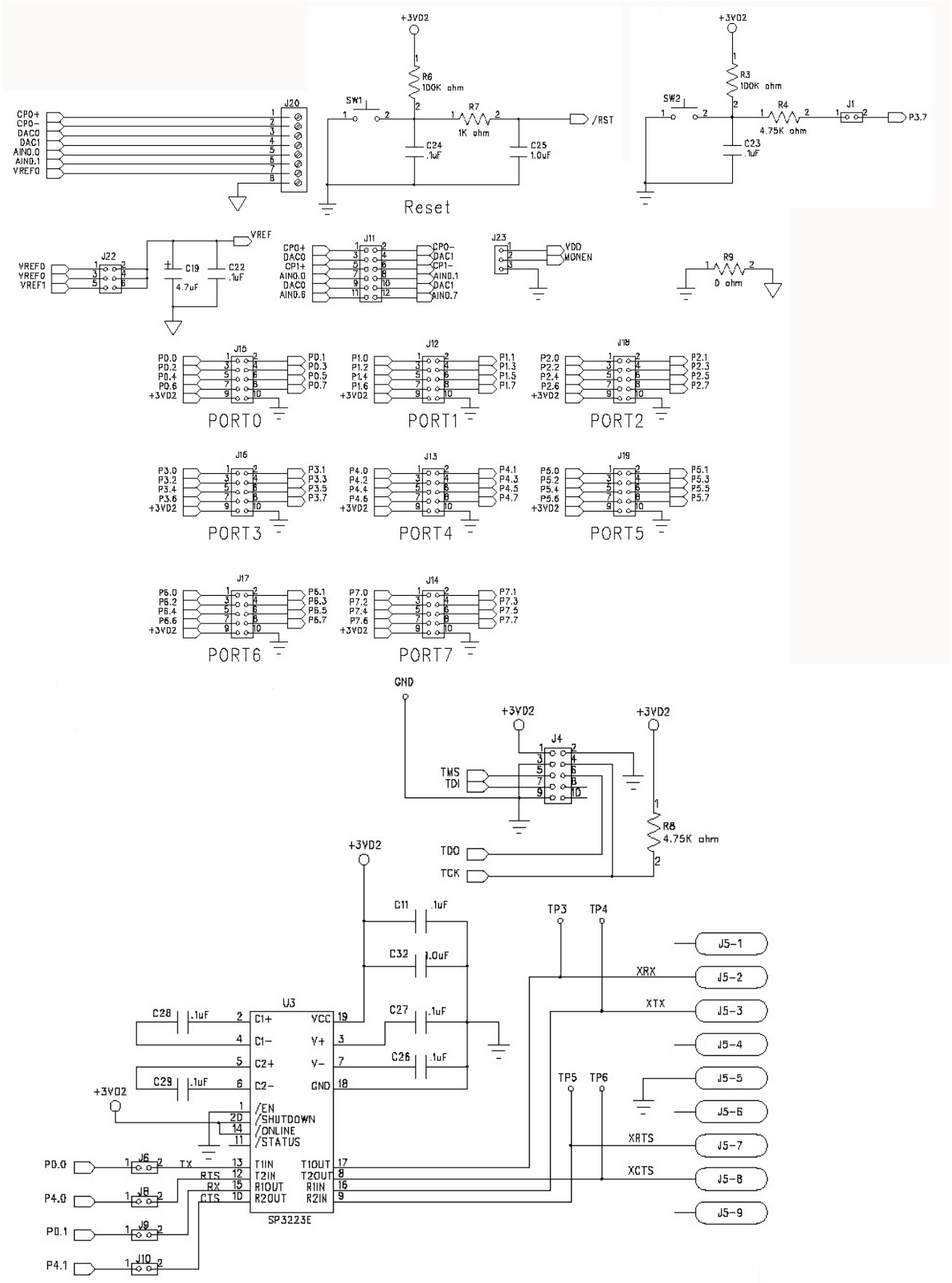


Département **ETI - IRC**
03/04/2017 16:21

1.2 Schéma de la carte C8051FX20-TB



Département **ETI - IRC**
03/04/2017 16:21



L'exécution du programme d'interruption Timer 2 sera visualisée a

8051 Instruction Set Table

Mates (c) 1995, 2002

	0000 .0	0001 .1	0010 .2	0011 .3	0100 .4	0101 .5	0110 .6	0111 .7	1000 .8	1001 .9	1010 .A	1011 .B	1100 .C	1101 .D	1110 .E	1111 .F	
							@R0	@R1	R0	R1	R2	R3	R4	R5	R6	R7	
0	0 1c 1	1 2c 2	2 2c 3	3 1c 1	4 1c 1	5 1c 1	6 1c 1	7 1c 1	8 1c 1	9 1c 1	10 1c 1	11 1c 1	12 1c 1	13 1c 1	14 1c 1	15 1c 1	INC
	NOP	AJMP 0A8	LJMP A16	RR A	INC A	INC A8	INC @R0	INC @R1	INC R0	INC R1	INC R2	INC R3	INC R4	INC R5	INC R6	INC R7	
1	16 2c 3	17 2c 2	18 2c 3	19 1c 1	20 1c 1	21 1c 1	22 1c 1	23 1c 1	24 1c 1	25 1c 1	26 1c 1	27 1c 1	28 1c 1	29 1c 1	30 1c 1	31 1c 1	DEC
	JBC BIT,AR	ACALL 0A8	LCALL A16	RRC A	DEC A	DEC A8	DEC @R0	DEC @R1	DEC R0	DEC R1	DEC R2	DEC R3	DEC R4	DEC R5	DEC R6	DEC R7	
2	32 2c 3	33 2c 2	34 2c 1	35 1c 1	36 1c 2	37 1c 2	38 1c 1	39 1c 1	40 1c 1	41 1c 1	42 1c 1	43 1c 1	44 1c 1	45 1c 1	46 1c 1	47 1c 1	ADD
	JB BIT,AR	AJMP 1A8	RET	RL A	ADD A,#D	ADD A,A8	ADD A,@R0	ADD A,@R1	ADD A,R0	ADD A,R1	ADD A,R2	ADD A,R3	ADD A,R4	ADD A,R5	ADD A,R6	ADD A,R7	
3	48 2c 3	49 2c 2	50 2c 1	51 1c 1	52 1c 2	53 1c 2	54 1c 1	55 1c 1	56 1c 1	57 1c 1	58 1c 1	59 1c 1	60 1c 1	61 1c 1	62 1c 1	63 1c 1	ADDC
	JNB BIT,AR	ACALL 1A8	RETI	RLC A	ADDC A,#D	ADDC A,A8	ADDC A,@R0	ADDC A,@R1	ADDC A,R0	ADDC A,R1	ADDC A,R2	ADDC A,R3	ADDC A,R4	ADDC A,R5	ADDC A,R6	ADDC A,R7	
4	64 2c 2	65 2c 2	66 1c 2	67 1c 3	68 1c 2	69 1c 2	70 1c 1	71 1c 1	72 1c 1	73 1c 1	74 1c 1	75 1c 1	76 1c 1	77 1c 1	78 1c 1	79 1c 1	ORL
	JC AR	AJMP 2A8	ORL A8,A	ORL A8,#D	ORL A,#D	ORL A,A8	ORL A,@R0	ORL A,@R1	ORL A,R0	ORL A,R1	ORL A,R2	ORL A,R3	ORL A,R4	ORL A,R5	ORL A,R6	ORL A,R7	
5	80 2c 2	81 2c 2	82 1c 2	83 1c 3	84 1c 2	85 1c 2	86 1c 1	87 1c 1	88 1c 1	89 1c 1	90 1c 1	91 1c 1	92 1c 1	93 1c 1	94 1c 1	95 1c 1	ANL
	JNC AR	ACALL 2A8	ANL A8,A	ANL A8,#D	ANL A,#D	ANL A,A8	ANL A,@R0	ANL A,@R1	ANL A,R0	ANL A,R1	ANL A,R2	ANL A,R3	ANL A,R4	ANL A,R5	ANL A,R6	ANL A,R7	
6	96 2c 2	97 2c 2	98 1c 2	99 1c 3	100 1c 2	101 1c 2	102 1c 1	103 1c 1	104 1c 1	105 1c 1	106 1c 1	107 1c 1	108 1c 1	109 1c 1	110 1c 1	111 1c 1	XRL
	JZ AR	AJMP 3A8	XRL A8,A	XRL A8,#D	XRL A,#D	XRL A,A8	XRL A,@R0	XRL A,@R1	XRL A,R0	XRL A,R1	XRL A,R2	XRL A,R3	XRL A,R4	XRL A,R5	XRL A,R6	XRL A,R7	
7	112 2c 2	113 2c 2	114 2c 2	115 2c 1	116 1c 2	117 2c 3	118 1c 2	119 1c 2	120 1c 2	121 1c 2	122 1c 2	123 2c 2	124 1c 2	125 1c 2	126 1c 2	127 1c 2	MOV
	JNZ AR	ACALL 3A8	ORL C,BIT	JMP @A+DPTR	MOV A,#D	MOV A8,#D	MOV @R0,#D	MOV @R1,#D	MOV R0,#D	MOV R1,#D	MOV R2,#D	MOV R3,#D	MOV R4,#D	MOV R5,#D	MOV R6,#D	MOV R7,#D	
8	128 2c 2	129 2c 2	130 2c 2	131 2c 1	132 4c 1	133 2c 3	134 2c 2	135 2c 2	136 2c 2	137 2c 2	138 2c 2	139 2c 2	140 2c 2	141 2c 2	142 2c 2	143 2c 2	MOV
	SJMP AR	AJMP 4A8	ANL C,BIT	MOVC A,@A+PC	DIV AB	MOV A8,A8	MOV A8,@R0	MOV A8,@R1	MOV A8,R0	MOV A8,R1	MOV A8,R2	MOV A8,R3	MOV A8,R4	MOV A8,R5	MOV A8,R6	MOV A8,R7	
9	144 2c 3	145 2c 2	146 2c 2	147 2c 1	148 1c 2	149 1c 2	150 1c 1	151 1c 1	152 1c 1	153 1c 1	154 1c 1	155 1c 1	156 1c 1	157 1c 1	158 1c 1	159 1c 1	SUBB
	MOV DPTR,#D16	ACALL 4A8	MOV BIT,C	MOVC A,@A+DPTR	SUBB A,#D	SUBB A,A8	SUBB A,@R0	SUBB A,@R1	SUBB A,R0	SUBB A,R1	SUBB A,R2	SUBB A,R3	SUBB A,R4	SUBB A,R5	SUBB A,R6	SUBB A,R7	
A	208 2c 2	161 2c 2	162 1c 2	163 2c 1	164 4c 1	165	166 2c 2	167 2c 2	168 2c 2	169 2c 2	170 2c 2	171 2c 2	172 2c 2	173 2c 2	174 2c 2	175 2c 2	MOV
	ORL C,/BIT	AJMP 5A8	MOV C,BIT	INC DPTR	MUL AB		MOV @R0,A8	MOV @R1,A8	MOV R0,A8	MOV R1,A8	MOV R2,A8	MOV R3,A8	MOV R4,A8	MOV R5,A8	MOV R6,A8	MOV R7,A8	
B	176 2c 2	177 2c 2	178 1c 2	179 1c 1	180 2c 3	181 2c 3	182 2c 3	183 2c 3	184 2c 3	185 2c 3	186 2c 3	187 2c 3	188 2c 3	189 2c 3	190 2c 3	191 2c 3	CJNE
	ANL C,/BIT	ACALL 5A8	CPL BIT	CPL C	CJNE A,#D,AR	CJNE A,A8,AR	CJNE @R0,#D,AR	CJNE @R1,#D,AR	CJNE R0,#D,AR	CJNE R1,#D,AR	CJNE R2,#D,AR	CJNE R3,#D,AR	CJNE R4,#D,AR	CJNE R5,#D,AR	CJNE R6,#D,AR	CJNE R7,#D,AR	
C	192 2c 2	193 2c 2	194 1c 2	195 1c 1	196 1c 1	197 1c 2	198 1c 1	199 1c 1	200 1c 1	201 1c 1	202 1c 1	203 1c 1	204 1c 1	205 1c 1	206 1c 1	207 1c 1	XCH
	PUSH A8	AJMP 6A8	CLR BIT	CLR C	SWAP A	XCH A,A8	XCH A,@R0	XCH A,@R1	XCH A,R0	XCH A,R1	XCH A,R2	XCH A,R3	XCH A,R4	XCH A,R5	XCH A,R6	XCH A,R7	
D	208 2c 2	209 2c 2	210 1c 2	211 1c 1	212 1c 1	213 2c 3	214 1c 1	215 1c 1	216 2c 2	217 2c 2	218 2c 2	219 2c 2	220 2c 2	221 2c 2	222 2c 2	223 2c 2	DJNZ
	POP A8	ACALL 6A8	SETB BIT	SETB C	DA A	DJNZ A8,AR	XCHD A,@R0	XCHD A,@R1	DJNZ R0,AR	DJNZ R1,AR	DJNZ R2,AR	DJNZ R3,AR	DJNZ R4,AR	DJNZ R5,AR	DJNZ R6,AR	DJNZ R7,AR	
E	224 2c 1	225 2c 2	226 2c 1	227 2c 1	228 1c 1	229 1c 2	230 1c 1	231 1c 1	232 1c 1	233 1c 1	234 1c 1	235 1c 1	236 1c 1	237 1c 1	238 1c 1	239 1c 1	MOV
	MOVX A,@DPTR	AJMP 7A8	MOVX A,@R0	MOVX A,@R1	CLR A	MOV A,A8	MOV A,@R0	MOV A,@R1	MOV A,R0	MOV A,R1	MOV A,R2	MOV A,R3	MOV A,R4	MOV A,R5	MOV A,R6	MOV A,R7	
F	240 2c 1	241 2c 2	242 2c 1	243 2c 1	244 1c 1	245 1c 2	246 1c 1	247 1c 1	248 1c 1	249 1c 1	250 1c 1	251 1c 1	252 1c 1	253 1c 1	254 1c 1	255 1c 1	MOV
	MOVX @DPTR,A	ACALL 7A8	MOVX @R0,A	MOVX @R1,A	CPL A	MOV A8,A	MOV @R0,A	MOV @R1,A	MOV R0,A	MOV R1,A	MOV R2,A	MOV R3,A	MOV R4,A	MOV R5,A	MOV R6,A	MOV R7,A	

A8 ..... address 8 bits AR .... relative address 8 bits D ..... data 8 bits
A16 ... address 16 bits BIT ... bit's address D16 .. data 16 bits

8051 Instruction Set Summary

Rn Register R7-R0 of the currently selected Register Bank.
Data 8-bit internal data location's address. This could be an internal Data RAM location (0-127) or a SFR [i.e. I/O port, control register, status register, etc. (128-255)].
@Ri 8-bit Internal Data RAM location (0-255) addressed indirectly through register R1 or R0.
#data 8-bit constant included in instruction.
#data16 16-bit constant included in instruction.
addr16 16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64k byte Program Memory address space.
addr11 11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2k byte page of Program Memory as the first byte of the following instruction.
rel Signed (two's component) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is –128 to +127 bytes relative to first byte of the following instruction.
bit Direct Addressed bit in Internal Data RAM or Special Function Register.

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

Note that operations on SFR byte address 206 or bit addresses 209-215 (i.e. the PSW or bits in the PSW) will also affect flag settings.

Mnemonic		Description	Byte	Cycle
Arithmetic operations				
ADD	A,Rn	Add register to accumulator	1	1
ADD	A,direct	Add direct byte to accumulator	2	**2**
ADD	A,@Ri	Add indirect RAM to accumulator	1	**2**
ADD	A,#data	Add immediate data to accumulator	2	**2**
ADDC	A,Rn	Add register to accumulator with carry flag	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	**2**
ADDC	A,@Ri	Add indirect RAM to A with carry flag	1	**2**
ADDC	A,#data	Add immediate data to A with carry flag	2	**2**
SUBB	A,Rn	Subtract register to accumulator with borrow	1	1
SUBB	A,direct	Subtract direct byte to A with carry borrow	2	**2**
SUBB	A,@Ri	Subtract indirect RAM to A with carry borrow	1	**2**
SUBB	A,#data	Subtract immediate data to A with carry borrow	2	**2**
INC	A	Increment accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	**2**
INC	@Ri	Increment indirect RAM	1	**2**
DEC	A	Decrement accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	**2**
DEC	@Ri	Decrement indirect RAM	1	**2**
INC	DPTR	Increment data pointer	1	**1**
MUL	AB	Multiply A and B -> [B hi]:[A lo]	1	4
DIV	AB	Divide A by B -> A=result, B=remainder	1	**8**
DA	A	Decimal adjust accumulator	1	1
CLR	A	Clear accumulator	1	1
CPL	A	Complement accumulator	1	1
RL	A	Rotate accumulator left	1	1
RLC	A	Rotate accumulator left through carry	1	1
RR	A	Rotate accumulator right	1	1
RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1
Logic operations				
ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	**2**
ANL	A,@Ri	AND indirect RAM to accumulator	1	**2**
ANL	A,#data	AND immediate data to accumulator	2	**2**
ANL	direct,A	AND accumulator to direct byte	2	**2**
ANL	direct,#data	AND immediate data to direct byte	3	**3**
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	**2**
ORL	A,@Ri	OR indirect RAM to accumulator	1	**2**
ORL	A,#data	OR immediate data to accumulator	2	**2**
ORL	direct,A	OR accumulator to direct byte	2	**2**
ORL	direct,#data	OR immediate data to direct byte	3	**3**
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A,direct	Exclusive OR direct byte to accumulator	2	**2**
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	**2**
XRL	A,#data	Exclusive OR immediate data to accumulator	2	**2**
XRL	direct,A	Exclusive OR accumulator to direct byte	2	**2**
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	**3**
Boolean variable manipulation				
CLR	C	Clear carry flag	1	1
CLR	bit	Clear direct bit	2	**2**
SETB	C	Set carry flag	1	1
SETB	bit	Set direct bit	2	**2**
CPL	C	Complement carry flag	1	1
CPL	bit	Complement direct bit	2	**2**
ANL	C,bit	AND direct bit to carry flag	2	2
ANL	C,/bit	AND complement of direct bit to carry	2	2
ORL	C,bit	OR direct bit to carry flag	2	2
ORL	C,/bit	OR complement of direct bit to carry	2	2
MOV	C,bit	Move direct bit to carry flag	2	**2**
MOV	bit,C	Move carry flag to direct bit	2	2
Program and machine control				
ACALL	addr11	Absolute subroutine call	2	**3**
LCALL	addr16	Long subroutine call	3	**4**
RET		Return from subroutine	1	**5**
RETI		Return from interrupt	1	**5**
AJMP	addr11	Absolute jump	2	**3**
LJMP	addr16	Long jump	3	**4**
SJMP	rel	Short jump (relative address)	2	**3**
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	**3**
JZ	rel	Jump if accumulator is zero	2	**2/3**
JNZ	rel	Jump if accumulator is not zero	2	**2/3**
JC	rel	Jump if carry flag is set	2	**2/3**
JNC	rel	Jump if carry flag is not set	2	**2/3**
JB	bit,rel	Jump if bit is set	3	**3/4**
JNB	bit,rel	Jump if bit is not set	3	**3/4**
JBC	bit,rel	Jump if direct bit is set and clear bit	3	**3/4**
CJNE	A,direct,rel	Compare direct byte to A and jump if not equal	3	**3/4**
CJNE	A,#data,rel	Compare immediate to A and jump if not equal	3	**3/4**
CJNE	Rn,#data,rel	Compare immed. to reg. and jump if not equal	3	**3/4**
CJNE	@Rn,#data,rel	Compare immed. to ind. and jump if not equal	3	**4/5**
DJNZ	Rn,rel	Decrement register and jump in not zero	2	**2/3**
DJNZ	direct,rel	Decrement direct byte and jump in not zero	3	**3/4**
NOP		No operation	1	1
Data transfer				
MOV	A,Rn	Move register to accumulator	1	1
MOV	A,direct*)	Move direct byte to accumulator	2	**2**
MOV	A,@Ri	Move indirect RAM to accumulator	1	**2**
MOV	A,#data	Move immediate data to accumulator	2	**2**
MOV	Rn,A	Move accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	**2**
MOV	Rn,#data	Move immediate data to register	2	**2**
MOV	direct,A	Move accumulator to direct byte	2	**2**
MOV	direct,Rn	Move register to direct byte	2	**2**
MOV	direct,direct	Move direct byte to direct byte	3	**3**
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	**3**
MOV	@Ri,A	Move accumulator to indirect RAM	1	**2**
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2
MOV	@Ri,#data	Move immediate data to indirect RAM	2	**2**
MOV	DPTR,#data16	Load data pointer with a 16-bit constant	3	**3**
MOVC	A,@A+DPTR	Move code byte relative to DPTR to accumulator	1	**3**
MOVC	A,@A+PC	Move code byte relative to PC to accumulator	1	**3**
MOVX	A,@Ri	Move external RAM (8-bit addr.) to A	1	**3**
MOVX	A,@DPTR	Move external RAM (16-bit addr.) to A	1	**3**
MOVX	@Ri,A	Move A to external RAM (8-bit addr.)	1	**3**
MOVX	@DPTR,A	Move A to external RAM (16-bit addr.)	1	**3**
PUSH	direct	Push direct byte onto stack	2	2
POP	direct	Pop direct byte from stack	2	2
XCH	A,Rn	Exchange register to accumulator	1	1
XCH	A,direct	Exchange direct byte to accumulator	2	**2**
XCH	A,@Ri	Exchange indirect RAM to accumulator	1	**2**
XCHD	A,@Ri	Exchange low-order nibble indir. RAM with A	1	**2**

*) MOV A,ACC is not a valid instruction

Version Cycles Compatibles 8051F020

Enjoy It! *Mates*

This paper was created by Štěpán Matějka alias Mates for anybody who needs it. Mates, Prague – Czech Republic 1998,2002