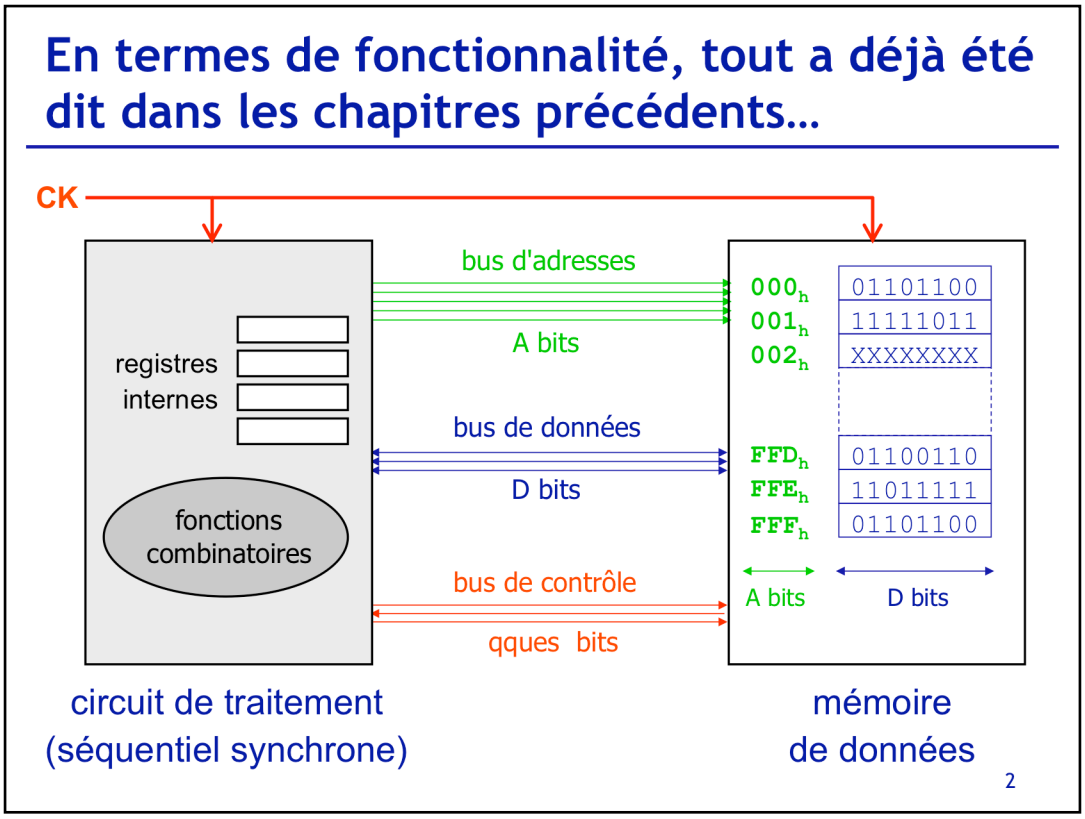
Chapitre 14:
Systèmes à microprocesseur
(Logique programmée)

14.1 - Introduction

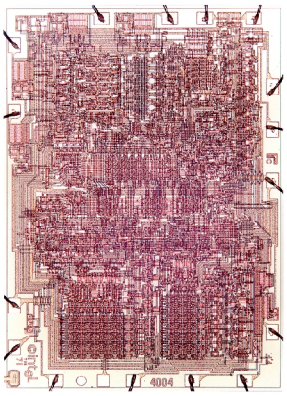


Dans les chapitres précédents nous avons vu, en plus des circuits de mémorisation, toute une série de logiques:
- la logique combinatoire et la logique séquentielle
- la logique asynchrone et la logique synchrone

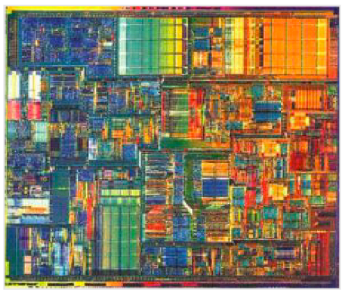
Ces différentes logiques numériques apportent un large éventail de possibilités de traitement. En particulier, toute opération à réaliser sur des données peut être considérée comme combinatoire ou comme séquentielle, de sorte qu'en termes de traitement toute opération est a priori possible sur base de ce que nous avons vu.

N.B.: la structure présentée ci-dessus a effectivement été utilisée dans les premiers ordinateurs (ceux des années 1940…), où le traitement était câblé (donc non modifiable). La seule mémoire présente était celle des données, comme ci-dessus.

… et pourtant on a inventé le microprocesseur (µP)!



Intel 4004
(1971)
2300 transistors
108 kHz



Intel Pentium IV
(2000)
100.000.000 transistors
1 GHz

3

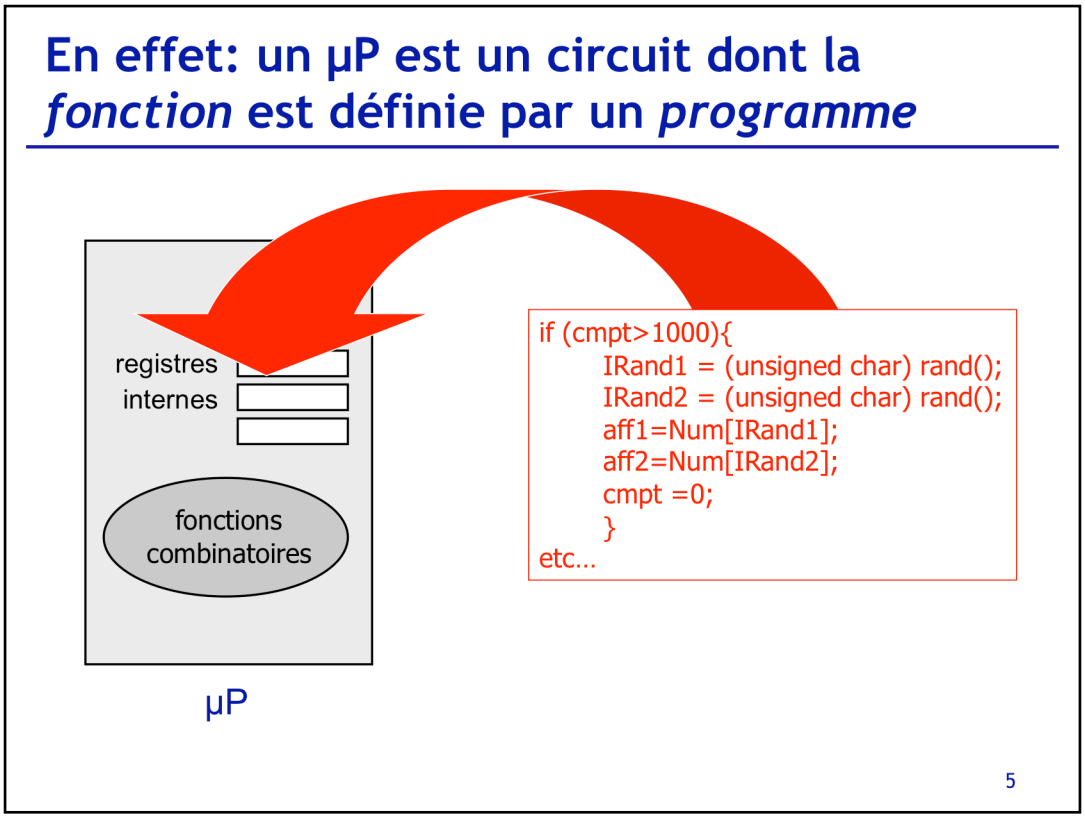
Et pourtant l'invention du microprocesseur (ou "processeur" ou µP) en 1971 a constitué une avancée majeure dans l'histoire de l'électronique... au point que l'ordinateur est devenu un équipement tout-à-fait courant (sans parler des microprocesseurs "embarqués", encore plus nombreux, qui se cachent dans les équipements les plus divers: voir plus loin).

Qu'ont donc les microprocesseurs de si particulier?

L'intérêt majeur d'un microprocesseur, c'est sa capacité à être **programmé**: en d'autres termes, le traitement de données réalisé par un microprocesseur n'est pas défini par le fabricant de celui-ci (comme pour les autres circuits) mais par son utilisateur final. Mieux encore: ce traitement peut varier autant de fois qu'on veut!

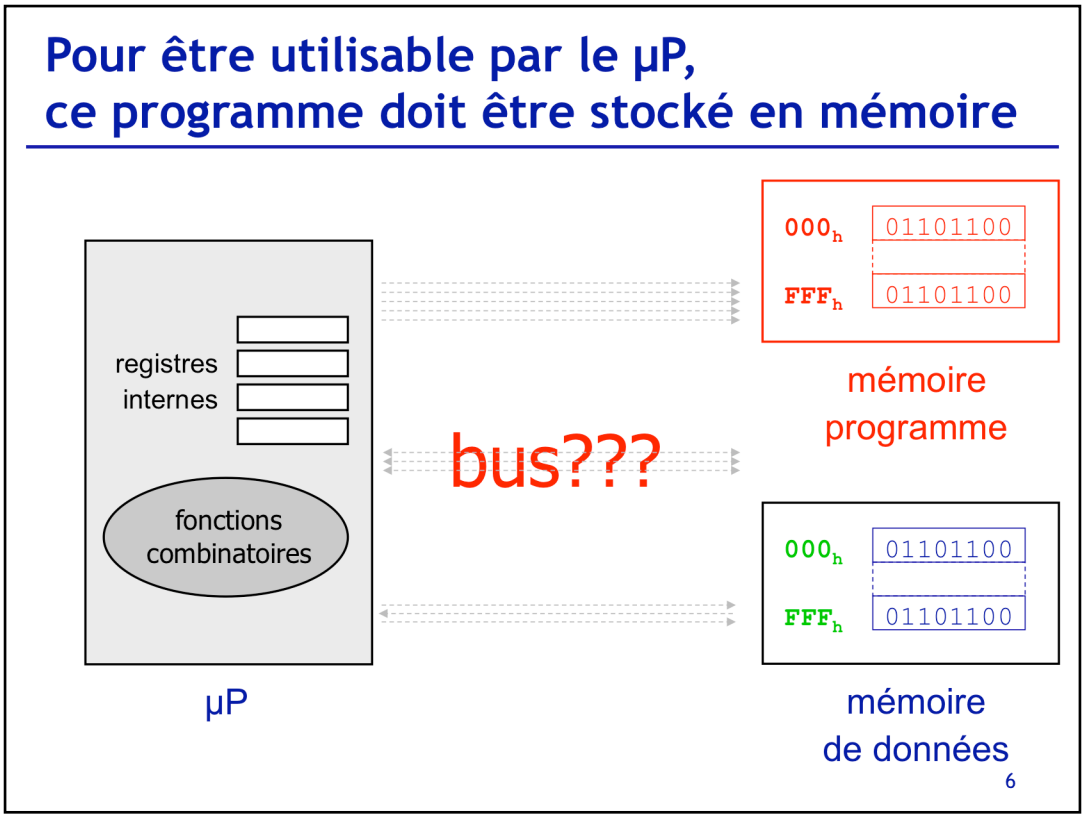
Ce qui fait le succès des microprocesseurs, c'est donc la flexibilité: un µP peut, dans une certaine mesure, s'adapter à tous les usages.

Sur base de ce critère, on distingue trois nouvelles logiques:
- la logique câblée: la fonction du circuit est définie une fois pour toutes par le fabricant, au moment de la conception
- la logique programmée (microprocesseurs et apparentés): la fonction du circuit est fixée par un programme au moment de l'utilisation
- la logique programmable (FPGAs): la fonction du circuit est définie par l'utilisateur mais elle ne peut pas être modifiée en fonctionnement (il s'agit donc d'un intermédiaire entre les deux cas précédents).



Concrètement, ce qui caractérise la logique programmée est le fait que le traitement à réaliser sur les données est précisé au moment même de ce traitement par un **programme**.

On peut donc dire que la fonction d'un microprocesseur, c'est d'exécuter un programme qui définira à son tour quel traitement effectuer sur des données.

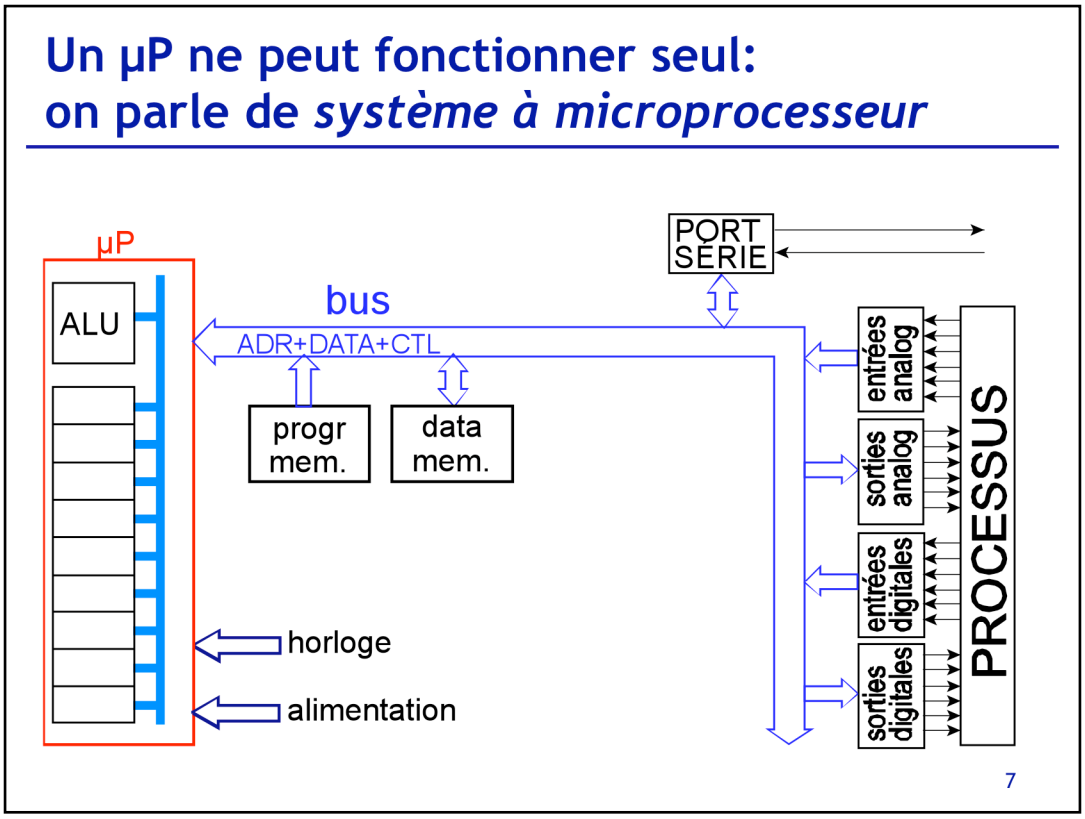


Par rapport au schéma de base donné précédemment (circuit séquentiel + bus + mémoire de données), l'existence d'un programme constitue une différence majeure que nous allons discuter tout au long de ce chapitre.

On peut néanmoins déjà noter que ce programme doit être "écrit" quelque part, à la disposition du microprocesseur. En pratique, ce programme sera placé dans une mémoire spécifique (*) appelée "**mémoire programme**" (par opposition à la mémoire de données qui contient les données à traiter).

Le fait qu'un seul processeur doit s'adresser à plusieurs mémoires posera certains problèmes spécifiques (qui seront expliqués par la suite) au niveau des bus.

(*) ou plus souvent dans un morceau spécifique de la mémoire



Un microprocesseur ne peut fonctionner seul: nous parlerons donc de "système à microprocesseur" (d'où le titre de ce chapitre) pour insister sur cet aspect.

Un **système à microprocesseur** regroupe l'ensemble des équipements nécessaires autour d'un microprocesseur (et le µP lui-même) pour obtenir un traitement de l'information.
Un système à microprocesseur comprend en principe:
- un microprocesseur (dont nous préciserons l'architecture interne par la suite)
- une ou plusieurs mémoires de données
- une mémoire programme
- des périphériques (*)
- des bus (données, adresses, contrôle) permettant aux quatre types de dispositifs précédents d'échanger des informations numériques
- un décodeur d'adresse (voir + loin)
- au moins une horloge
- un dispositif d'alimentation électrique

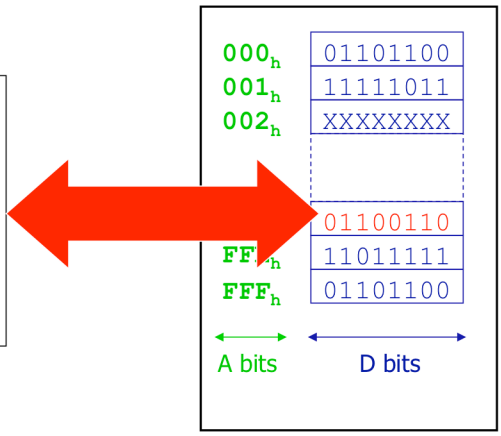
(*) Tout système à microprocesseur échange forcément des données avec un autre équipement et/ou avec un opérateur humain. Un système à microprocesseur comprend donc toujours un certain nombre de dispositifs "d'entrées/sorties", appelés **périphériques**, qui permettent la communication avec le monde extérieur.
Dans le domaine de l'informatique, sont considérés comme des périphériques: écran, clavier, souris, imprimante, scanner, lecteurs de disques CD ou DVD, lecteurs de carte FLASH, manettes de jeux, modem, toutes les cartes d'extension PC (audio, vidéo, réseau et autres), ports série, ports parallèles, etc…
Dans le domaine industriel, on peut encore citer: toutes les interfaces homme/machine (boutons, claviers, écrans en tous genres), les cartes d'acquisition, les convertisseurs analogique/numérique et numérique/analogique, etc.
Nous verrons que les périphériques apparaissent simplement, pour le processeur, comme des mémoires supplémentaires (indépendemment de la fonction du périphérique).

Chapitre 14:
Systèmes à microprocesseur
(Logique programmée)

14.2 - Programme, instructions et langages

Un *programme* est une suite d'informations définissant un traitement sur des données

```
if (cmpt>1000){
        IRand1 = (unsigned char) rand();
        IRand2 = (unsigned char) rand();
        aff1=Num[IRand1];
        aff2=Num[IRand2];
        cmpt =0;
        }
etc…
```

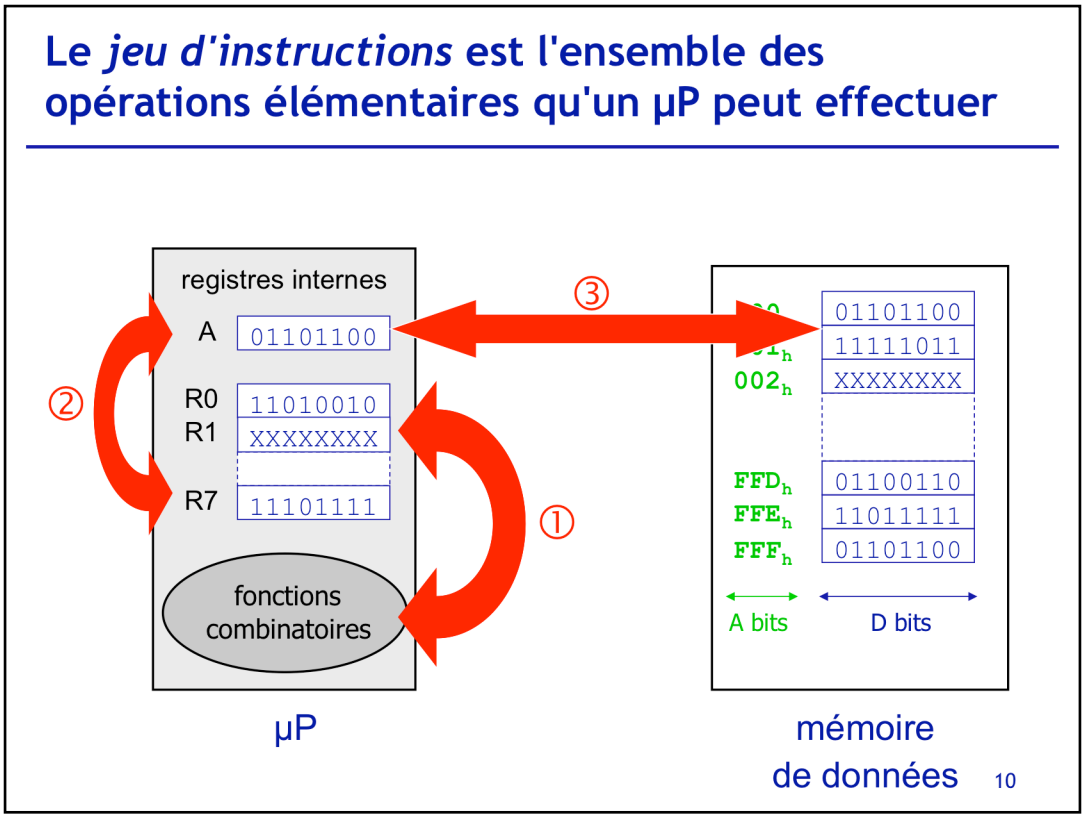


mémoire de données

9

Tout système à microprocesseur exécute un programme qui est une suite d'informations (dans un langage adéquat) définissant un traitement à effectuer sur des données (stockées par ailleurs dans une mémoire de données, typiquement une RAM).

Un tel programme est souvent écrit en langage de haut niveau (ex: C) comme illustré ci-dessus.



Voyons maintenant ce qui se passe du côté du microprocesseur: chaque type de microprocesseur possède un **jeu d'instructions** déterminé. Une instruction est une opération *élémentaire* (voir ci-dessous) exécutable par le microprocesseur. Le jeu d'instructions d'un µP comporte typiquement quelques dizaines à quelques centaines d'instructions.

Les instructions envisageables sont globalement de quatre types:
- la première catégorie regroupe toutes les instrucitons de *traitement* proprement dit (opérations logiques ou arithmétiques élémentaires sur des bits ou des mots binaires). Ces opérations ne peuvent cependant agir que sur des valeurs qui sont stockées dans les registres internes du processeur (elles ne peuvent pas agir directement sur les valeurs qui sont dans la mémoire de données externe), d'où la raison d'être des deux catégories suivantes:
- la deuxième catégorie regroupe les instructions qui permettent de *déplacer* des valeurs entre des registres internes du processeur
- la troisième catégorie regroupe les instructions qui permettent d'*échanger* des valeurs (lecture et écriture) *avec les mémoires externes* au processeur (mémoire de données et périphériques)
- enfin il existe une quatrième catégorie qui regroupe les instructions permettant de modifier l'ordre d'exécution d'un programme (instructions de branchement, notamment les boucles)

Un microprocesseur n'a besoin que de ces quatre types d'instructions pour faire… à peu près tout ce que vous voulez.

Jeu d'instructions (1): transfert de données

Mnémonique	Opération	Modes adress dir	ind	reg	imm	Cycles
MOV A,<src>	A = <src>	X	X	X	X	1
MOV <dest>,A	<dest> = A	X	X	X		1
MOV <dest>,<src>	<dest> = <src>	X	X	X	X	2
MOV DPTR,#data16	DPTR = 16-bit immediate constant				X	2
PUSH <src>	INC SP:MOV"@SP",<src>	X				2
POP <dest>	MOV <dest>,"@SP":DEC SP	X				2
XCH A,<byte>	ACC and <byte> exchange data	X	X	X		1
XCHD A,@Ri	ACC and @Ri exchange low nibbles			X		1

MOVX = MOVe eXternal

Mnémonique	Opération	Bits d'adresse	Cycles
MOVX A,@Ri	Read external RAM @Ri	(P2)+8 bits	2
MOVX @Ri,A	Write external RAM @Ri	(P2)+8 bits	2
MOVX A,@DPTR	Read external RAM @DPTR	16 bits	2
MOVX @DPTR,A	Write external RAM @DPTR	16 bits	2

11

A titre d'information, on a explicité en cinq slides un jeu complet d'instructions (µP 8051).

Le premier tableau ci-dessus reprend les instructions correspondant aux échanges de données entre registres internes au µP.

Le second tableau reprend les instructions d'échange de données entre le µP et une mémoire de données externe.

Jeu d'instructions (2): opérations arithmétiques

Mnémonique	Opération	Modes adress. dir	ind	reg	imm	Cycles
ADD A,<byte>	A = A + <byte>	X	X	X	X	1
ADDC A,<byte>	A = A + <byte> + C	X	X	X	X	1
SUBB A,<byte>	A = A - <byte> - C	X	X	X	X	1
INC A	A = A + 1	Accumulator only				1
INC <byte>	<byte> = <byte> + 1	X	X	X		1
INC DPTR	DPTR = DPTR + 1	Data Pointer only				2
DEC A	A = A - 1	Accumulator only				1
DEC <byte>	<byte> = <byte> - 1	X	X	X		1
MUL AB	B:A = B x A	ACC and B only				4
DIV AB	A = Int[A/B] B = Mod[A/B]	ACC and B only				4
DA A	Decimal Adjust	Accumulator only				1

12

Le tableau ci-dessus reprend les opérations arithmétiques réalisables sur des mots binaires.

Jeu d'instructions (3): opérations logiques

Mnémonique	Opération	Modes adress. dir	ind	reg	imm	Cycles
ANL A,<byte>	A = A.AND. <byte>	X	X	X	X	1
ANL <byte>,A	<byte> = <byte> .AND.A	X				1
ANL <byte>,#data	<byte> = <byte> .AND.#data	X				2
ORL A,<byte>	A = A.OR.<byte>	X	X	X	X	1
ORL <byte>,A	<byte> = <byte> .OR.A	X				1
ORL <byte>,#data	<byte> = <byte> .OR.#data	X				2
XRL A,<byte>	A = A.XOR. <byte>	X	X	X	X	1
XRL <byte>,A	<byte> = <byte> .XOR.A	X				1
XRL <byte>,#data	<byte> = <byte> .XOR.#data	X				2
CRL A	A = 00H	Accumulator only				1
CPL A	A = .NOT.A	Accumulator only				1
RL A	Rotate ACC Left 1 bit	Accumulator only				1
RLC A	Rotate Left through Carry	Accumulator only				1
RR A	Rotate ACC Right 1 bit	Accumulator only				1
RRC A	Rotate Right through Carry	Accumulator only				1
SWAP A	Swap Nibbles in A	Accumulator only				1

13

Le tableau ci-dessus représente les opérations logiques réalisables sur des mots binaires.

Jeu d'instructions (4): manipulation de bits

Mnémonique	Opération	Modes adres	Cycles
ANL C,bit	C = C.AND.bit	direct bit	2
ANL C,/bit	C = C.AND..NOT.bit		2
ORL C,bit	C = C.OR.bit		2
ORL C,/bit	C = C.OR..NOT.bit		2
MOV C,bit	C = bit		1
MOV bit,C	bit = C		2
CLR C	C = 0		1
CLR bit	bit = 0		1
SETB C	C = 1		1
SETB bit	bit = 1		1
CPL C	C = .NOT.C		1
CPL bit	bit = .NOT.bit		1
JC rel	Jump if C = 1		2
JNC rel	Jump if C = 0		2
JB bit,rel	Jump if bit = 1		2
JNB bit,rel	Jump if bit = 0		2
JBC bit,rel	Jump if bit = 1;CLR bit		2

14

Le tableau ci-dessus représente les opérations logiques réalisables sur des bits.

Jeu d'instructions (5): branchements inconditionnels et conditionnels

Mnémonique	Opération	Modes adress	Cycles
JMP addr	Jump to addr		2
JMP @A+DPTR	Jump to A + DPTR		2
CALL addr	Call subroutine at addr		2
RET	Return from subroutine		2
RETI	Return from interrupt		2
NOP	No operation		1

Mnémonique	Opération	Modes adress dir	ind	reg	imm	Cycles
JZ rel	Jump if A = 0	Accumulator only				2
JNZ rel	Jump if A ... 0	Accumulator only				2
DJNZ <byte>,rel	Decr.and jump if not zero	X		X		2
CJNE A,<byte>,rel	Jump if A ... <byte>	X			X	2
CJNE <byte>,#data,rel	Jump if <byte> ... #data		X	X		2

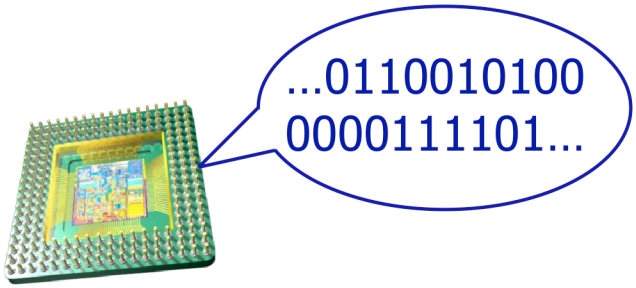
15

Les tableaux ci-dessus reprennent respectivement les instructions de saut inconditionnel et les instructions de saut conditionnel.

Mais un µP ne "comprend" que des "0" et des "1"

```
if (cmpt>1000){
     IRand1 = (unsigned char) rand();
     IRand2 = (unsigned char) rand();
     aff1=Num[IRand1];
     aff2=Num[IRand2];
     cmpt =0;
     }
etc...
```

= ???

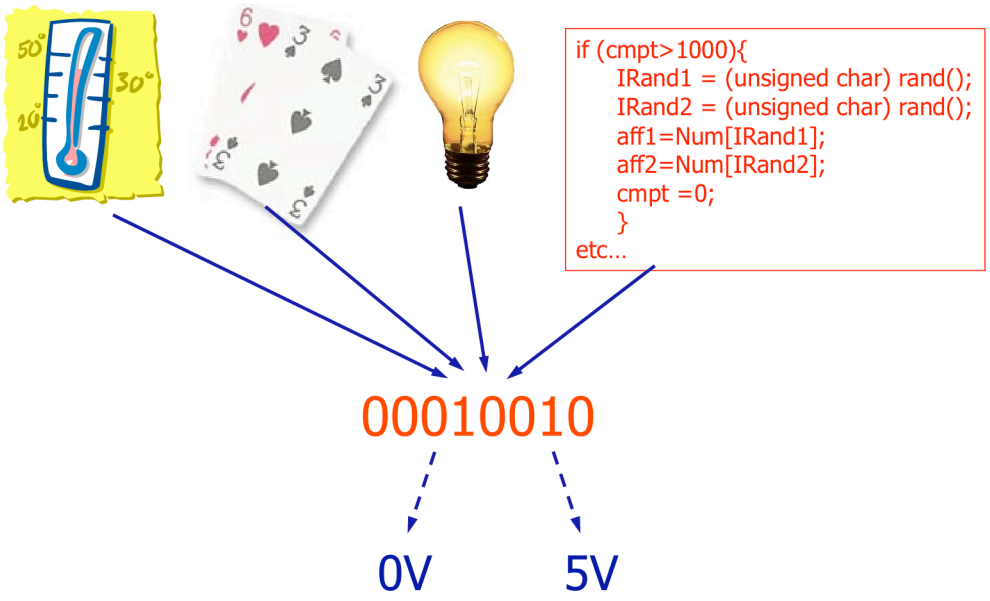


16

A ce stade, nous avons:
- d'une part un microprocesseur qui possède un certain jeu d'instructions (dont on a vu qu'elles sont élémentaires)
- d'autre part un programme, souvent écrit en langage de haut niveau, qui manipule un grand nombre de variables et qui est basé sur des opérations beaucoup plus complexes que celles figurant dans le jeu d'instructions du µP

Le microprocesseur, étant un circuit électronique numérique, ne peut manipuler que des "0" et des "1" logiques. Un programme comme celui représenté ci-dessus lui est donc incompréhensible…

Toute information peut être traduite sous forme de bits: le programme aussi!

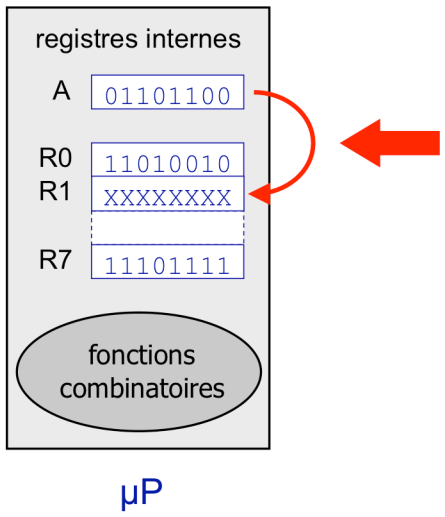


17

Néanmoins nous avons vu (chap. 10) que toute information peut être traduite sous forme numérique, et en particulier sous forme de 0 et de 1 (qui, à leur tour, vont être traduits en signaux électriques).

C'est également ce que nous allons faire avec le programme…

A chaque instruction sont associés un mnémonique et un *op-code*



mnémonique (assembleur)	op-code (lang. machine)
MOV A,R0	11101000
MOV A,R1	11101001
MOV A,R2	11101010
MOV A,R3	11101011
...	...
MOV A,R7	11101111
MOV R0,A	11111000
MOV A,@R0	11100110

18

Le principe utilisé par un µP pour "comprendre" un programme est le suivant:

Plaçons-nous d'abord au niveau des instructions, c'est-à-dire au niveau des opérations élémentaires reconnues par le µP (par exemple: aller chercher une valeur dans la mémoire de données, ou encore: additionner le contenu de deux registres internes, etc).

A chacune des instructions sont associés (voir exemples dans les tableaux ci-dessus):
- à destination des humains: un mnémonique, c'est-à-dire un code de quelques lettres qui rappelle la fonction de l'instruction
- à destination du microprocesseur: un mot binaire unique, appelée "op-code" ou "code opératoire", qui désigne l'instruction de manière univoque.

Chacune de ces manières de formuler une instruction (et donc un programme plus vaste) constitue en soi un "langage" au sens informatique:
- la formulation en mnémoniques constitue le **langage assembleur**.
- la formulation en op-codes constitue le **langage machine**, en fait le seul que le µP peut "comprendre".

N.B.: Ci-dessus, on a notamment représenté l'instruction "MOV A,R1" qui consiste à copier le contenu du registre interne A dans le registre interne R1. Dans le µP 8051, cette instruction particulière possède l'op-code "11101001".

Les langages de haut niveau doivent être traduits en langage machine (compilation)

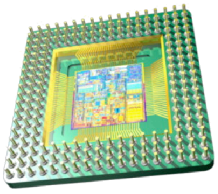
algorithme

JAVA / C++

```
if (cmpt>1000)
    {aff1=Num[IRand1];}
```

C / pascal

Compilation



assembleur

```
MOV A,R1
```

langage machine

```
11101001
```

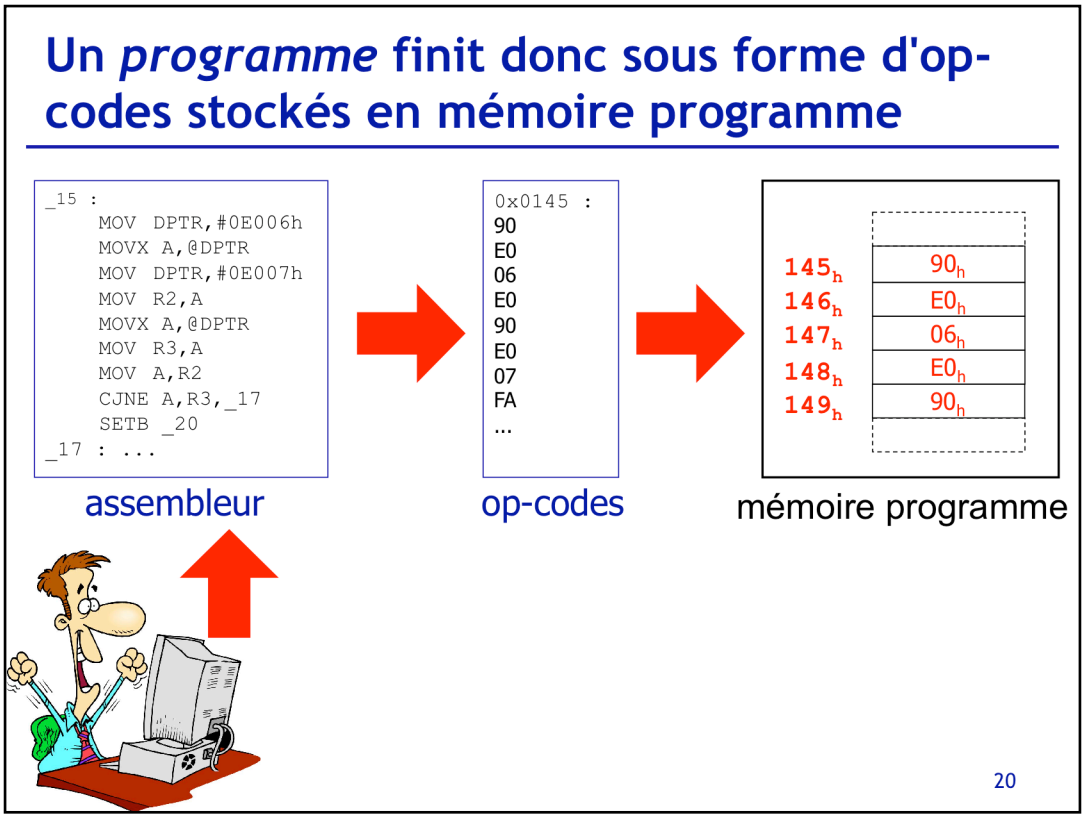
19

En pratique, on programme généralement dans un langage de beaucoup plus haut niveau que l'assembleur ou le langage machine. L'utilisation de langages de haut niveau permet d'écrire de manière concise des instructions très complexes, donc permet de gagner du temps de programmation. Il existe dans ce domaine de très nombreux langages dont nous ne discuterons pas les avantages et inconvénients ici.

Mais quel que soit le langage de haut niveau utilisé, un programme écrit dans un tel langage doit, pour être utilisé par le microprocesseur, être traduit en instructions de bas niveau (langage machine). Cette étape est l'étape de compilation (*).

Dans l'absolu, il est possible de programmer en langage machine ou en assembleur, l'avantage de ces langages étant qu'ils sont beaucoup plus puissants et rapides (car plus proches de ce que fait réellement le µP) que les langages de haut niveau.
En pratique, il arrive qu'on programme directement aux niveaux les plus bas des parties de code particulièrement critiques en termes de sécurité des équipements ou des personnes ou en termes de délai d'exécution.

(*) en réalité il a plusieurs étapes et/ou variantes que nous ne détaillerons pas ici pour passer d'un langage de haut niveau au langage machine.



Dans tous les cas de figure (programmation de haut niveau ou programmation de bas niveau), le programme finit donc toujours par être traduit en langage machine, c'est-à-dire finit sous la forme d'une suite d'op-codes, qui ne sont rien d'autre qu'une suite de chiffres généralement exprimés en hexadécimal.

Ce sont ces op-codes qui vont être stockés dans la mémoire programme et qui vont être interprétés par le µP comme une suite d'instructions.

En termes de support physique de l'information, rien ne distingue donc les données des op-codes (il s'agit dans les deux cas de nombres binaires stockés dans une mémoire), sinon que:
- données et op-codes sont stockés dans des parties différentes de la mémoire
- les op-codes sont interprétés par le µP comme des instructions à effectuer (tandis que les données n'ont aucune "signification" pour le µP).

Remarque: chaque "0" ou "1" qui existe dans une donnée ou dans un op-code existe en fait finalement sous forme de tension (0V ou 5V) dans un circuit.