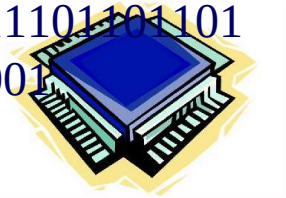
Instructions assembleur

0001011011110110
0001011101101101
10001001



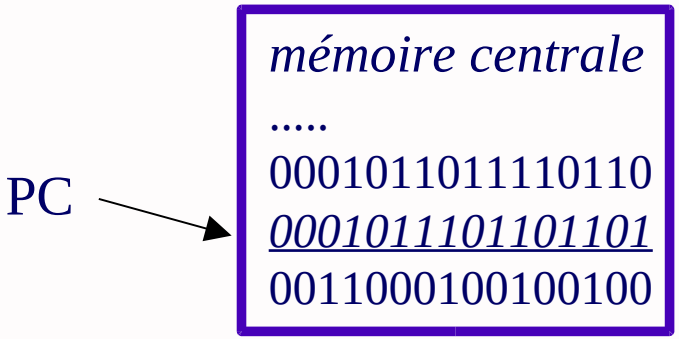
Instruction vue par le programmeur assembleur ou instruction élémentaire cible d'un compilateur

Réalise une modification de l'état interne du processeur ou de la mémoire

1. Programme binaire

- Généré par le compilateur (ou écrit par le programmeur en langage machine)
- Ensemble d'instructions machine chargées en mémoire centrale lors de l'exécution d'un programme
- Les instructions sont exécutées **séquentiellement**

  PC (Program Counter) contient l'adresse de l'instruction à exécuter



si le programme commence à l'adresse 5000h PC vaut 5000h puis 5002h puis 5006h ....

  - les instructions de contrôle (if else, boucle for, while, ...) modifient la valeur de PC

Exemples de programme binaire

```
// Programme C
int i=5;
int j=10;
int k = 0;
main() {
    i= i+j+k;
}
```

```
// code 68000 cross-compilé
gcc2_compiled.:
___gnu_compiled_c:
.data
.globl _i
_i:
        .long 5
.globl _j
_j:
        .long 10
.globl _k
_k:
        .long 0

.text
.globl _main
_main:
        movel _i,d0
        addl _j,d0
        movel d0,d1
        addl _k,d1
        movel d1,_i
```

```
// code Pentium 4
.file  "essai.c"
.globl i
        .data
        .align 4
        .type i,@object
        .size i,4
i:
        .long       5
...
...

        .text
.globl main
        .type       main,@function
main:
        movl        j, %eax
        addl        i, %eax
        addl        k, %eax
        movl        %eax, i
```

directives

étiquettes

Les données

Les instructions

2. Jeu d'instructions

- Une instruction est définie par :
  - l'opération à réaliser *que faire ?*
  - les opérandes : registres et cases mémoire modifiés / utilisés par l'instruction *sur quoi opérer ?*
- Une instruction est représentée par un code binaire qui suit un *format d'instruction*

- Quelles instructions ?
  - quelles opérations élémentaires ? (différentes utilisations : DSP, multimédia, CPU...)
  - quels types de données ? (booléens, flottants, entiers, codes ASCII, BCD)
- Objectif : assurer les meilleures performances en terme de
  - rapidité d'exécution
  - taille du code
  - facilité d'utilisation (écriture des compilateurs)
- Tendance : simplicité du jeu d'instructions et du codage (même format d'instruction, même temps d'exécution)
  - RISC : Reduced Instruction Set Computers

Exemple d'instruction 68000

En binaire : 0010 0010 0011 1001 0000 0000 0000 0000 0000 0000 0010 0000

En hexadécimal : 2239 0000 0020h

La même chose lisible (langage assembleur) :

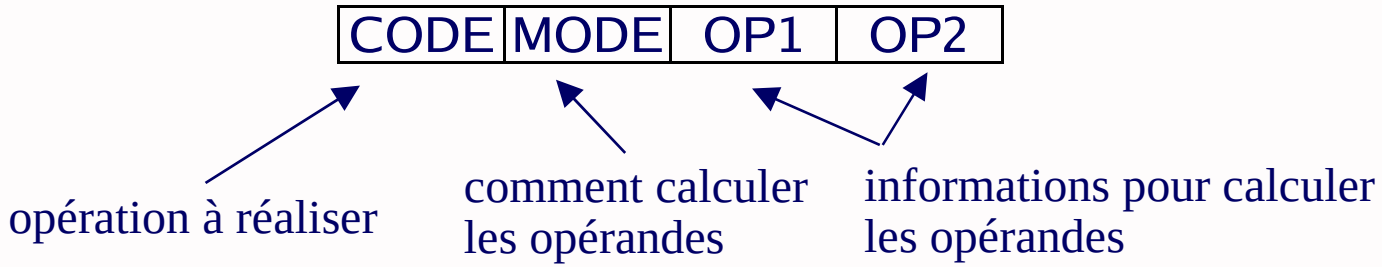
movel _i,d1

- movel : transfert de données (move) de type long (l)
- i : opérande source (une donnée mémoire)
- d1 : opérande destination (un registre interne)

Le registre interne d1 reçoit le contenu de la variable en mémoire centrale i

3. Format d'instruction

- L'instruction est représentée par un code binaire. Ce code binaire est découpé en champs ayant des rôles différents : c'est le format d'instruction.



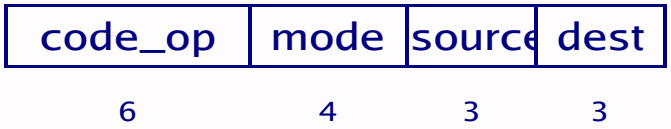
- Le format d'instruction peut varier selon le type des opérandes
- Tendance : un format uniforme

Exemple : PROCSI

- Langage assembleur : <inst> <op-dest> , <op-source>
  Exemple : add R1, R4

- Format d'instruction : 1, 2 ou 3 mots de 16 bits

  1er mot :



code_op : le code de l'opération
mode : le mode d'adressage
source : le code du n° de registre source
dest : le code du n° de registre destination

2ème mot : valeur immédiate ou adresse
3ème mot : adresse

Exemple : 000000 0000 0100 0001

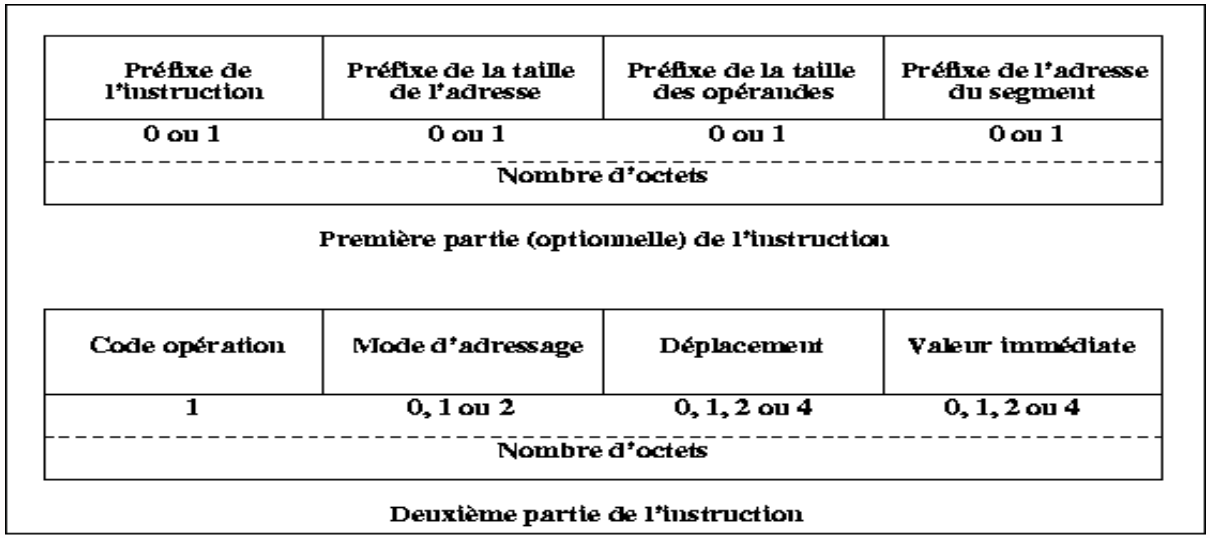
Exemple : Processeur ARM

	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 09 08	07	06	05	04	03 02 01 00
Data processing immediate	cond	0	0	1	op				S	Rn	Rd	rotate	immediate				
Data processing immediate shift	cond	0	0	0	opcode				S	Rn	Rd	shift immediate	shift		0	Rm	
Data processing register shift	cond	0	0	0	opcode				S	Rn	Rd	Rs	0	shift		1	Rm
Multiply	cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Multiply long	cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm
Move from status register	cond	0	0	0	1	0	R	0	0	SBO	Rd	SBZ					
Move immediate to status register	cond	0	0	1	1	0	R	1	0	Mask	SBO	rotate	immediate				
Move register to status register	cond	0	0	0	1	0	R	1	0	Mask	SBO	SBZ				0	Rm
Branch/exchange instruction set	cond	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	0	1	Rm
Load/store immediate offset	cond	0	1	0	P	U	B	W	L	Rn	Rd	immediate					
Load/store register offset	cond	0	1	1	P	U	B	W	L	Rn	Rd	shift immediate	shift		0	Rm	
Load/store halfword/ signed byte	cond	0	0	0	P	U	1	W	L	Rn	Rd	High offset	1	S	H	1	Low offset
Load/store halfword/ signed byte	cond	0	0	0	P	U	0	W	L	Rn	Rd	SBZ	1	S	H	1	Rm
Swap/swap byte	cond	0	0	0	1	0	B	0	0	Rn	Rd	SBZ	1	0	0	1	Rm
Load/store multiple	cond	1	0	0	P	U	S	W	L	Rn	Register list						
Coprocessor data processing	cond	1	1	1	0	op1				CRn	CRd	cp_num	op2		0	CRm	
Coprocessor register transfers	cond	1	1	1	0	op1			L	CRn	Rd	cp_num	op2		1	CRm	
Coprocessor load and store	cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8_bit_offset				
Branch and branch with link	cond	1	0	1	L	24_bit_offset											
Software interrupt	cond	1	1	1	1	swi_number											
Undefined	cond	0	1	1	x	x	x	x	x	x	x	x	x	x	x	1	x x x x

Figure 1-2 ARM instruction set formats

Exemple : famille xx86 (Pentium)

très grand nombre d'instructions (synonymes) et modes d'adressage



Préfixe de l'instruction	Préfixe de la taille de l'adresse	Préfixe de la taille des opérandes	Préfixe de l'adresse du segment
0 ou 1	0 ou 1	0 ou 1	0 ou 1
Nombre d'octets			

Première partie (optionnelle) de l'instruction

Code opération	Mode d'adressage	Déplacement	Valeur immédiate
1	0, 1 ou 2	0, 1, 2 ou 4	0, 1, 2 ou 4
Nombre d'octets			

Deuxième partie de l'instruction

FIG. 5.4 – *Format des instructions du Pentium*[*]

*** Source :**http://www.irisa.fr/caps/projects/TechnologicalSurvey/micro/PI-957-html

syntaxe: op-code op-dest , op-source

```
Ex:   add   cx, bx
      add   cl, bl
      mov   80[bx], ax
```

4. Opérandes

4.1. Registres internes

Registres généraux dans une mémoire statique interne, accessible en un top d'horloge processeur

- moins de traffic mémoire
- programmes plus rapides
- code machine plus dense

A chaque fois que c'est possible

4.2. Opérandes mémoire :

la donnée est dans la mémoire centrale

accès à un octet (8 bits), mot (16 bits), double mot (32 bits), mot long (64 bits)

Initialisation de variables locales, tableaux

Organisation mémoire :

4.3. Modes d'adressage

- Spécification de la façon de calculer un opérande (objet à accéder)
  - constante
  - registre
  - case mémoire
- Codé par un champ d'instruction
- Le codage dépend du nombre de modes possibles et de l'interdépendance entre modes et codes opération
- Exigences contradictoires :
  - Avoir autant de modes d'adressage que possible (facilité de programmation)
  - Impact sur la taille des instructions et donc du programme

Modes d'adressage usuels

Mode d'adressage	Informations codées dans l'instruction	Mode de calcul de l'opérande	Utilisation
Immédiat	op = valeur immédiate	op *la valeur elle-même*	constantes `i = 6;`
Registre	op = numéro registre	reg[op] *le registre interne n° op*	calculs intermédiaires `for (int i =0;i<100;i++)` `  x = x+ y*x +2*y`
Direct ou absolu	op = adresse	mem[op] *la case n° op de la mémoire centrale*	accès variables statiques `int i;`
Indirect	op = numéro registre	mem[reg[op]] *accès à une case de la mémoire centrale. le n° est stoké dans le registre interne n° op*	accès par pointeur ou référence `Train t = new Train("TGV");` `Train tt = t;` `tt.setNom("corail");`
Déplacement	op = numéro du registre op' = adresse de base	mem[reg[op] + op'] *accès à une case de la mémoire centrale. le n° est op' + la valeur stokée dans le registre interne n° op*	accès tableau ou pile `int[] t = {8,2,5};` `int x = t[2];`

Exemple: modes d'adressage de PROCSI

page 4

Représentation	Effet	Mode adressage	Code
ADD R1, R5	**REG[1] <- REG[1] + REG[5]**	**registre/registre**	**000000 0000 101 001**
	Addition des registres 1 et 5 Le résultat va dans le registre 1		000000 : addition 0000 : mode registre / registre 001 : la destination est le registre n°1 011 : la source est le registre n°5
ADD R2, # 24	**REG[2] <- REG[2] + 24**	**registre/immédiat**	**000000 0100 xxx 010 0000000000011000**
	24 est additionné au registre R2		000000 : addition 0100 : mode registre / immédiat 010 : la destination est le registre n°2 XXX : il n'y a pas de registre source 0000000000011000 : 24, la donnée immédiate
ADD [8], # 24	**MEM[8] <- MEM[8] + 24**	**direct/immédiat**	**000000 0101 xxx xxx 0000000000011000 0000000000001000**
	24 est additionné à la case mémoire n°8		000000 : addition 0101 : mode direct / immédiat XXX : il n'y a pas de registre mis en jeu 0000000000011000 : 24, la donnée immédiate 0000000000001000 : 8, l'adresse de la case mémoire

Quel est le code binaire de ADD R5,R7 ?

- 000000 0001 0011 0101 ☐
- 000000 0000 0111 0101 ☐
- 000000 0000 0101 0111 ☐

La variable i est à l'adresse 9 dans la mémoire

Comment traduire l'instruction java “i = i + 3”

- ADD [9],R3 ☐
- ADD [9],#3 ☐
- ADD [9],[3] ☐

Exemple: modes d'adressage de PROCSI suite ...

Représentation	Effet	Mode adressage	Code
ADD R1, [R5]	**REG[1] <- REG[1] + MEM[REG[5]]**	**registre/indirect**	**000000 1100 101 001**
	Additionne le registre 1 et le mot mémoire dont l'adresse est dans R5 Le résultat va dans le registre 1		
ADD R2, 8[R5]	**REG[2] <- REG[2] + MEM[REG[5]+ 8]**	**registre/déplacement**	**000000 1001 xxx 010** **0000 0000 0000 1000**
	Additionne le registre 2 et le mot mémoire Dont l'adresse est le contenu de R5 + 8 Le résultat va dans le registre 2		

Si le tableau “int[] T” est à l'adresse 8 dans la mémoire, si le registre R5 contient la valeur 8 alors [R5] correspond à T[0]

MEM

Adresse	Contenu
8	T[0]
9	
10	
11	
12	T[1]
13	
14	
15	

REG

Numéro	Contenu
0	
1	
2	
3	
4	
5	8
6	
7	

Sachant que les “int” en Java sont codés sur 32 bits (soit 4 cases mémoire), à quoi correspond 8[R5] ?

T[5] ☐
T[8] ☐
T[2] ☐

5. Opérations du jeu d'instruction

5.1. Classes d'opérations

- arithmétiques et logiques (entier ou flottant)

  Calculs
- chargement / rangement

  Transferts de / vers la mémoire
- contrôle

  Rompt l’exécution séquentielle du programme (modifie PC)
- système

  Interruptions : pour appel périphériques
- autres (Ex: MMX)

5.2. Instructions de contrôle

Rompt l'exécution séquentielle du programme

PC: registre «Program Counter» contient l'adresse de la prochaine instruction à exécuter

- exécution séquentielle : PC est incrémenté de la taille du format de l'instruction en cours d'exécution
  il pointe donc sur la prochaine instruction à exécuter

- instruction de contrôle : modifie la valeur de PC
  ce n'est donc pas l'instruction suivante qui est exécutée

Exemple PROCSI

```
8000h : 000000 0000 010 001   ADD R1, R2
8001h : 000000 0100 xxx 001   ADD R1, #24
8002h : 0000000000011000
8003h : 000100 xxxx xxx xxx   JMP 8360h
8004h : 1000 0011 0110 0000
8005h : ..............................
..... : ..............................
8360h : 000000 0000 110 000   ADD R0, R5
```

adresses en hexa dans mémoire instruction

PC = 8000h
PC <- 8000h + 1
PC <- 8001h + 2
} séquentiel

PC <- 8360h contrôle

- **Adresse de branchement**
  - le programme est écrit avec une étiquette symbolique,
  - l'adresse relative du branchement est connue lors de la phase de compilation,
  - l'adresse réelle du branchement est connue lors du chargement du programme en mémoire centrale (dépend de la machine).
- **Condition de branchement**
  - test sur le résultat d'une opération arithmétique ou logique

    Exemple : résultat nul, résultat positif, retenue …
- **La condition est**
  - générée par l'unité de calcul de l'unité centrale (ALU)
  - stockée dans un registre spécial accessible par la partie contrôle (SR)

    -> la partie contrôle prend en entrée la valeur de SR pour savoir si le branchement doit être pris

5.3. Catégories d'instructions de contrôle

- saut (branchement inconditionnel)
  PC <- adresse branchement
  goto

- branchement conditionnel
  si condition vraie PC <- adresse branchement
  sinon PC inchangé
  if then else, while, for

- appel procédure
  "PC_sauv "<- PC
  PC <- adresse branchement

- retour procédure
  PC <- "PC_sauv"

Exemple: PROCSI

jmp : “jump”saut non conditionnel

Non conditionnel

```
8301h          add R1, R2
8302h          jmp suite
8304h          sub R1, R4
…              …
          suite:
8600h          store [80],R1
```

PC = 8301h : PC <- 8302h; R1 <- R1 + R2

PC = 8302h : PC <- 8304h; PC <- 8600h

PC = 8600h : PC <- 8602h; MEM[80] <- R1

Exemple: PROCSI

jeq : “Jump if EQual”saut conditionnel si le résultat est nul

Conditionnel

```
8301h          add R1, R2
8302h          jeq suite
8304h          sub R1, R4
…              …
        suite:
8600h          store [80],R1
```

PC = 8301h : PC <- 8302h; R1 <- R1 + R2

Si R1 = - R2:

PC = 8302h : PC <- 8304h; PC <- 8600h

PC = 8600h : PC <- 8602h; MEM[80] <- R1

Si R1≠ - R2:

PC = 8302h : PC <- 8304h;

PC = 8304h: PC <- 8306h; R1 <- R1 - R4

Ici le résultat est nul, le flag Z de SR est à 1
donc le branchement est pris

On suppose que le compilateur a associé le registre R1 à la variable x, le registre R2 à la variable y et le registre R3 à la variable z

Quel code PROCSI correspond à :

```
x = x + y;
if (x!=10) x = x + 22;
z++;
```

☐

```
        add R1,R2
        cmp R1,#10
        jeq suite
        add R1,#22
suite : add R3,#1
```

☐

```
        add R1,R2
        cmp R1,#10
        jne suite
        add R3,#1
suite : add R1,#22
        add R3,#1
```

☐

```
        add R1,R2
        cmp R1,#10
        jne suite
        add R3,#1
        jmp fin
suite : add R1,#22
        add R3,#1
fin :
```

Exemple : une boucle en C et en 68000

```
int result = 8;
int i=0;

main()
{
  for (i = 5;i < 10; i++)
    result++;
}
```

```
 _main:
        moveq #5,d0          d0 vaut 5
        movel d0,_i          i vaut d0
L2:
        moveq #9,d0          d0 reçoit 9
        cmpl _i,d0           comparaison de i et d0
        jge L5               saut à L5 si i <= d0
        jra L3               saut à L3
L5:
        addql #1,_result     result reçoit result+1
L4:
        addql #1,_i          i reçoit i + 1
        jra L2               saut à L2
L3:
        rts                  retour
```

5.4. Procédures

- **Le code de la procédure utilise les paramètres**
- **La valeur des paramètres est connue lors de l'exécution**
- **Le nombre d'appels n'est pas toujours connu de façon statique : appels dans un if, appels imbriqués de procédures, procédures récursives, ...**

➡ **Il faut un mécanisme de saut pour "aller" à la procédure, et il faut un mécanisme pour en revenir**

➡ **Une pile pour gérer :**

- le passage des paramètres
- l'appel de procédure
- le retour de procédure

Pile: zone contiguë de la mémoire centrale
un registre spécialisé SP (stack pointer) du CPU contient l'adresse du sommet de pile

Communication entre appelant et procédure

1. Séquence d'appel — *Programme appelant*

- empiler les paramètres
- empiler l'adresse de retour (PC courant)
- PC prend l'adresse de la procédure

2. Séquence d'entrée — *Procédure*

- sauvegarde additionnelle de registres
- allocation de pile pour variables locales

3. Séquence de sortie — *Procédure*

- désalloue les variables locales de la pile
- met le résultat du calcul dans un endroit accessible par l'appelant
- PC prend la valeur du sommet de pile

4. Séquence de retour — *Programme appelant*

- l'appelant récupère le résultat et en dispose

Le détail des appels de procédures sera vu en cours d'assembleur ARM

Appel de la procédure dans le programme appelant :

- **“push” pour chaque paramètre**
- **“call <adresse_proc>” : 1. empile l'adresse de retour (PC courant)**
  2. modifie PC : PC ← <adresse_proc>

L'exécution continue dans la procédure

Dans la procédure :

- **“push” pour sauvegarder les registres modifiés par la procédure**
- **récupère les paramètres dans la pile : adressage mémoire avec déplacement par rapport à SP**
- **calcul**
- **“pop” pour restituer les valeurs initiales des registres**
- **“ret” : la valeur en sommet de pile est dépilée et placée dans PC**

L'exécution continue dans le programme appelant

variable à l'adresse 50

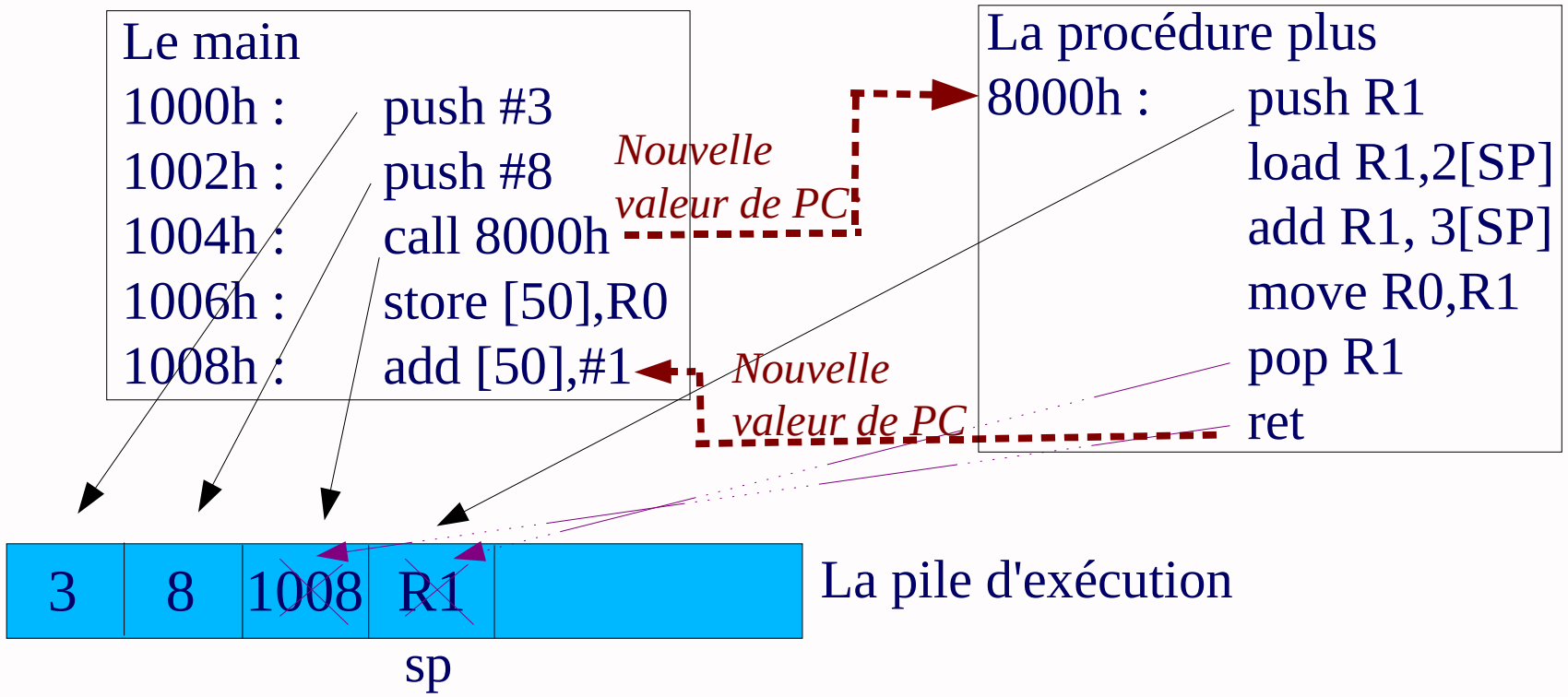
main à l'adresse 1000h

```
public static void main(String [] s){
        int a = plus(3, 8);
        a++;
}
```

procédure à l'adresse 8000h

instruction à l'adresse 1008h

Code PROCSI : le résultat est renvoyé dans R0



5.5. Interruptions

L'exécution normale du programme est interrompue pour raison logicielle ou matérielle

Trois mécanismes de base

- **Interruptions logicielles : appel au noyau du système d'exploitation**
- **Conditions d'erreur : division par 0, stack overflow, défaut de page, ...**
- **Interruptions matérielles : clic souris, clic clavier, ...**

Traitement identique à un appel de procédure mais :

- **Sauvegarde du contexte du programme (pas seulement PC actuel)**
- **La routine d'interruption est stockée dans un endroit protégé de la mémoire**
- **L'appel de la routine d'interruption est fait par la partie contrôle dans le cas des conditions d'erreur et des interruptions matérielles**
- **Table d'interruption : fait le lien entre type d'interruption et adresse de la routine associée**