

**INF 227**

***Introduction***

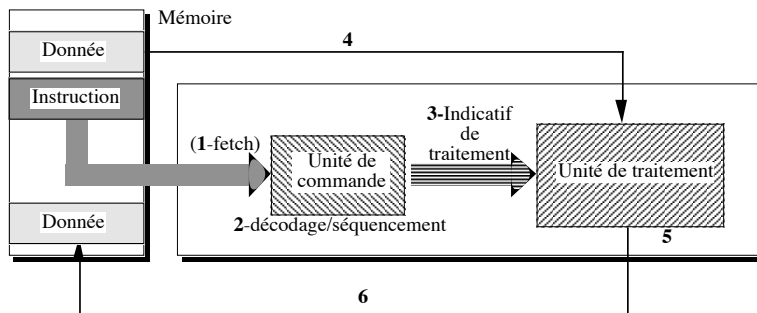
**B. Dupouy**

## *Plan*

- Historique, modules de base : processeur, mémoire, unités d'échange, bus.
- ☞ • Processeur : -Exécution d'une instruction.
  - Codage des instructions
- Mémoire.
- Entrées/sorties : gestion des unités d'échange. Interruptions.
- Représentation des données

### Exécution d'une instruction

- Comment est exécutée une instruction :



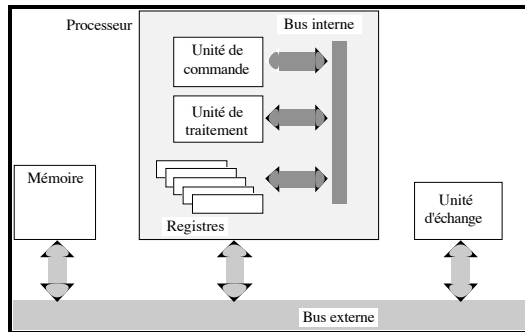
- 1 Lire une instruction en mémoire (*fetch*).
- 2 Comprendre l'instruction (*decode*).
- 3 Sélectionner l'opération à effectuer dans "unité de traitement".
- 4 Charger les données si nécessaire (*fetch operands*).
- 5 Effectuer le calcul (*execute*).
- 6 Ranger le résultat du calcul (*store*).
- 7 Aller à 1.

**L'utilisation de caches accélère les accès mémoire (*fetch, store*).**

### Processeur : récapitulatif

- L'unité de commande doit :
  - Aller chercher les instructions en mémoire (Fetch),
  - Les reconnaître (Decodage).
  - Indiquer à l'unité de traitement quels sont les traitements arithmétiques (+, -, ...) et logiques (ou, et, ...) à effectuer (séquencement).
- Ces unités utilisent des **registres** pour mémoriser les informations manipulées. Aucune opération ne se fait directement en mémoire.
- Ces unités et les registres sont reliés par des voies de communication appelées **bus internes** ou **chemin de données**.

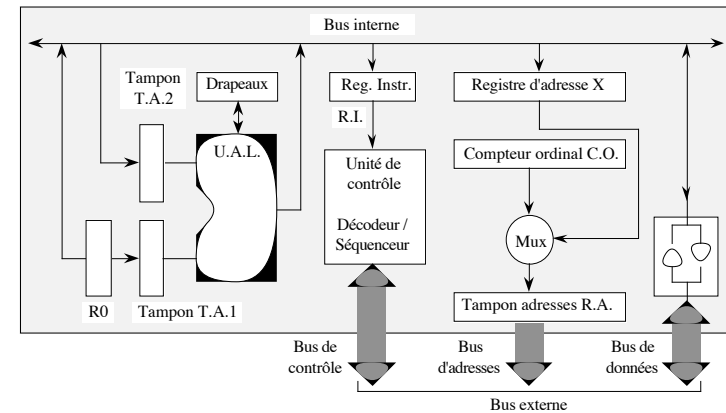
### Processeur : les registres



- Registres importants :
  - Compteur Ordinal, CO (Program Counter - PC) : adresse de l'instruction à exécuter.
  - Pointeur de pile (stack pointer).
  - Registre de drapeaux pour traiter les branchements :
    - Retenue (Carry ou CY),
    - Indicateur de nullité (Z),
    - Indicateur de parité (P),
    - Signe (S),
    - Overflow.
  - Registre d'instruction (RI) qui contient l'instruction à exécuter.

### Exemple d'exécution d'une instruction

- On rappelle les quatre étapes pour exécuter une instruction :
  - Fetch (CO sur le bus, **attente**, saisie de l'info. sur le bus)
  - Decode
  - Execute + fetch operands
  - Store
- Que se passe-t-il sur ce processeur très simple :

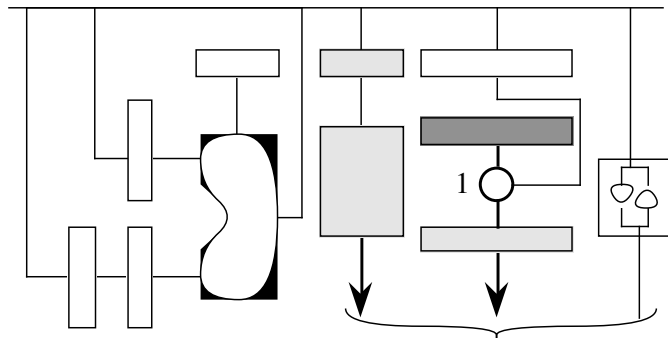


### Cycles d'exécution sur le processeur précédent

- Soit l'instruction :

ADD R0, (X)

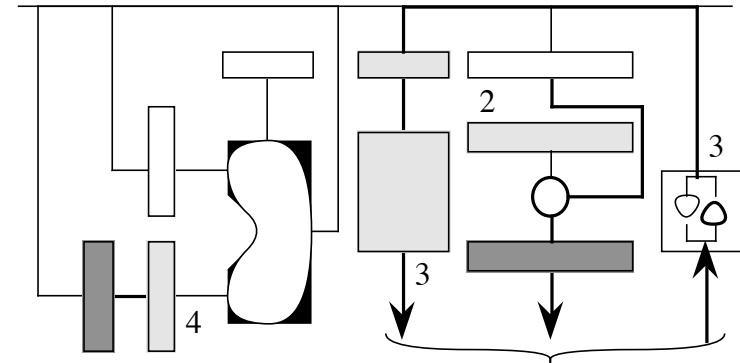
(Ajouter le contenu de l'adresse indiquée dans X à R0 et ranger le résultat dans R0)



- Phase 1 : lecture de l'instruction à exécuter (Fetch).

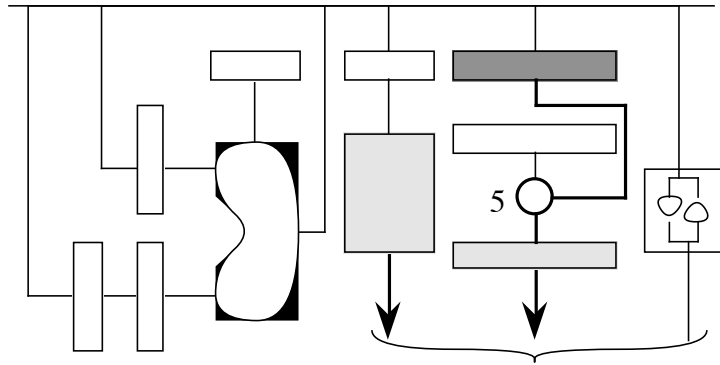
(On place le contenu de CO sur le bus d'adresse via le registre tampon d'adresse RA).

### Cycles d'exécution sur le processeur précédent



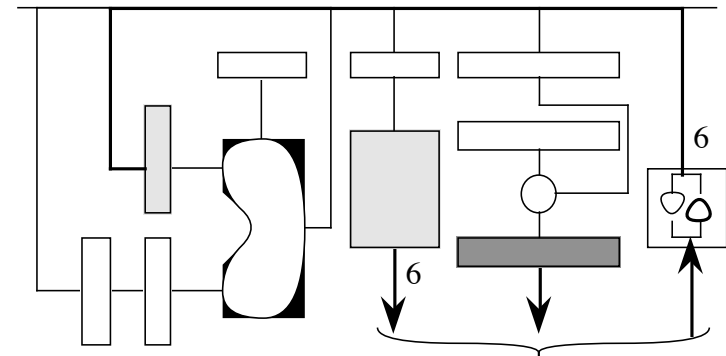
- Phase 2 : on incrémente le compteur ordinal.
- Phase 3 : on charge l'instruction à exécuter dans le registre d'instruction RI.
- Phase 4 : on transfère le contenu de R0 dans le tampon TA1

**Cycles d'exécution  
sur le processeur précédent**



- Phase 5 : on aiguille le contenu du registre X vers le tampon d'adresses RA.

**Cycles d'exécution  
sur le processeur précédent**



- Phase 6 : on lit le deuxième opérande (l'adresse dans X).



### Codage des instructions : code opération, opérandes

- Code opération :

Il y a différents types d'instructions à coder dans le champ code opération :

- arithmétiques
- logiques
- branchements

- Différents types d'opérandes :

- constantes
- cellules mémoire
- registres

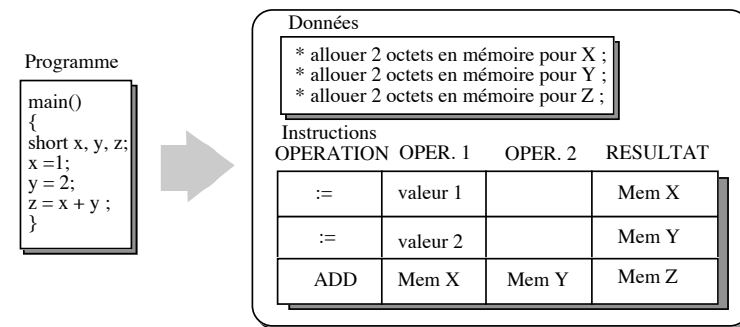
- Codage d'instructions à deux opérandes au plus :



- Suivant les **objectifs et le nombre de bits** dont on dispose pour coder une instruction, on fait figurer le type des opérandes dans les champs opérandes ou dans le code opération.

### Rôle du compilateur

- Génération de code et implantation des variables en mémoire (le langage machine est fait d'instructions à deux opérandes) :



### Codage des instructions : Accès mémoire (1)

• On donne ci-dessous le code d'une fonction factorielle et son implantation en mémoire (même jeu d'instruction) :

```

int fact (int n)
{
int Tmp = 1;

do
{
Tmp = Tmp * n;
}
while((--n) > 1);

return (Tmp);
}
    
```

0x1A				
0x20				
0x100	:=	1	0x1A	
	:= *	0x1A	0x20	0x1A
	:= -	0x20	1	0x20
	< ?	0x20	1	0x100

• Le code est implanté à partir de l'adresse 0x100 et les variables Tmp et n sont implantées aux adresses 0x1A et 0x20.

• Combien d'accès à la mémoire pour exécuter une itération de la boucle du code ci-dessus (chaque accès demande 32 bits) :

Instruction 1 : 1 accès instr. + 1 accès opérande -> 2 accès  
 Instruction 2 : 1 accès instr. + 3 accès opérande -> 4 accès  
 Instruction 3 : 1 accès instr. + 3 accès opérande -> 4 accès  
 Instruction 4 : 1 accès instr. + 1 accès opérande -> 2 accès  
 Total : 12 accès de 32 bits

• Si le bus fait 32/n bits (en général 8, 16 ou 32 bits), les accès bus sont au nombre de : 12 \* n accès au bus pour une itération

### Codage des instructions : Accès mémoire (2)

• On rappelle le schéma précédent :

```

int fact (int n)
{
int Tmp = 1;

do
{
Tmp = Tmp * n;
}
while((--n) > 1);

return (Tmp);
}
    
```

0x1A				
0x20				
0x100	:=	1	0x1A	
	:= *	0x1A	0x20	0x1A
	:= -	0x20	1	0x20
	< ?	0x20	1	0x100

• Si on plante les variables sur des registres on passe à :

Instruction 1 : 1 accès instr. + 0 accès opérande -> 1 accès

Instruction 2 : 1 accès instr. + 0 accès opérande -> 1 accès

Instruction 3 : 1 accès instr. + 0 accès opérande -> 1 accès

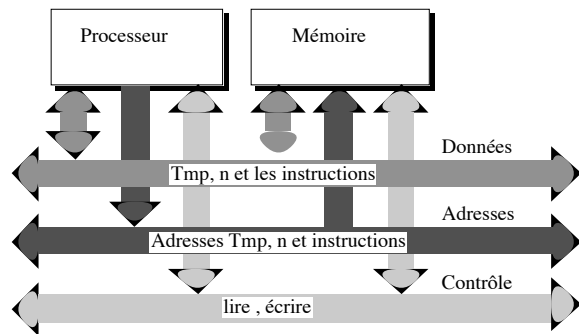
Instruction 4 : 1 accès instr. + 0 accès opérande -> 1 accès

Total : 4 \* n accès au bus pour une itération

**Les performances du programme sont considérablement améliorées : le rôle du compilateur est primordial.**

## Les accès mémoire à l'exécution : utilisation de registres

- Lors de l'exécution de la fonction précédente, le trafic sur le bus est intense, surtout s'il fait 8 bits :



Pour limiter le trafic sur le bus lors de l'exécution des programmes, le compilateur va implanter les variables sur des REGISTRES.

## Opérandes : différents modes d'adressage

- Comment passe-t-on du contenu du champ adresse (mode d'adressage) à l'adresse effective :

<p><b>- direct</b></p> <p>Le champ adresse détermine complètement la localisation d'un opérande (mémoire ou registre)</p>	<pre>void main ( void) {     int i;     i = 1; }</pre>
<p><b>- indirect</b></p> <p>L'adresse qui figure dans l'instruction ne désigne pas l'opérande, mais le numéro de la cellule mémoire ou du registre qui contient l'adresse effective de cet opérande</p>	<pre>void main ( void) {     int i;     int * Ptr;      Ptr = &amp;i;     *Ptr = 1; }</pre>
<p><b>- indexé</b></p> <p>L'adresse effective de l'opérande est obtenue en ajoutant à l'adresse indiquée par le champ opérande le contenu d'un index</p>	<pre>void main ( void) {     int Tab [10];      Tab [4] = 0; }</pre>

## Mode d'adressage et accès mémoire

- Nombre d'accès à la mémoire pour obtenir l'adresse effective (on ne compte pas les accès mémoire pour aller chercher l'instruction elle-même) :

Adressage immédiat (utilisation de constantes dont la valeur figure dans le champ instruction)	▣▣▣▣ pas d'accès mémoire
Adressage direct par registre	▣▣▣▣ pas d'accès mémoire
Adressage direct par cellule mémoire	▣▣▣▣ un accès mémoire
Adressage indirect par registre	▣▣▣▣ un accès mémoire
Adressage indirect par cellule mémoire	▣▣▣▣ deux accès mémoire

## Accès à la mémoire : exemples

- Exemple 1 :

si la constante 1 est rangée dans une cellule mémoire, il faut un accès supplémentaire.

```
void main ( void)
{
  int i;
  ...
  i = 1;
}
```

i elle même peut être rangée dans un registre, dans ce cas il n'y a pas d'accès mémoire concernant les opérandes

- Exemple 2 :

si Ptr est dans un registre, le calcul de l'adresse effective demande un accès mémoire, sinon deux.

```
void main ( void)
{
  int * Ptr;
  ...
  *Ptr = 1;
}
```

▣▣▣▣ **Importance des mémoires caches**

## **Opérandes : différents modes d'adressage**

- Le champ opérande doit donc indiquer :
  - TYPE de l'opérande (registre, case mémoire ou constante)
  - QUEL opérande (numéro de registre, de case mémoire ou valeur de constante)
  - MODE d'adressage (direct, indirect, indexé).

## **Plan**

- Historique, modules de base : processeur, mémoire, unités d'échange, bus.
- Processeur : -Exécution d'une instruction.
  - Codage des instructions
- Mémoire.
- ☞ • Entrées/sorties : gestion des unités d'échange. Interruptions.
- Représentation des données

### *Unités d'échange*

- Rôle des UE : adaptation matérielle (débit, granularité des informations) et adaptation fonctionnelle :



- Rôle qui peut aller de :
  - la simple adaptation matérielle entre l'ordinateur et le périphérique ;
  - à la gestion de tout l'échange d'information entre le couple processeur-mémoire et le périphérique.
- Adaptation matérielle :
  - niveaux de tension,
  - mode de commande (courant / tension),
  - mise en forme des signaux (niveaux, impulsions).
- Adaptation fonctionnelle :
  - type de donnée (mot, octet, bloc),
  - vitesse de transfert (ou débit),

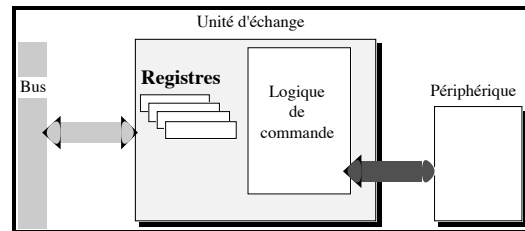
- synchronisation des informations, gestion du protocole d'échange.

### ***Entrées-sorties: généralités***

- Comment adresser un périphérique ?
  - espace d'adressage unique (le périphérique est vu comme de la mémoire)
  - espace d'adressage séparé (instructions spéciales)
- Comment transférer les données ?
  - méthode du test d'état (busy waiting, polling) et gestion du transfert par le processeur,
  - interruptions
  - interruptions et accès direct mémoire (gestion par l'unité d'échange).
  - canaux d'échanges de données (non traité ici, pour les grands systèmes).
- Gestion par interruption :
  - QUI a émis le signal ?
  - QUE faire ?

- Les interruptions servent à gérer tous types d'événements asynchrones (cf. boucle d'événements de X, les signaux en C).

## **Structure des unités d'échange**



- Les unités d'échange se comparent à des processeurs spécialisés : les plus simples d'entre elles travaillent sous contrôle du processeur, les autres dialoguent directement avec la mémoire,

## **Structure des unités d'échange**

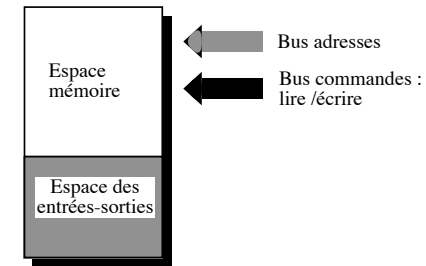
### Registres principaux :

- Registre d'état : indique l'état de l'unité :
  - transmetteur prêt / récepteur prêt / erreur de parité, etc.
- Registres de données : l'unité centrale vient y écrire ou lire les données à envoyer ou reçues (respectivement).
- Registre de commande : le processeur vient y écrire l'E/S à exécuter (cf. registre d'instructions),
- Registres de paramètres : servent à préciser les paramètres des tâches d'E/S à réaliser :
  - numéros de piste ou secteurs sur disques, vitesse de transmission, adresses de transfert en mémoire, etc.

## Espace d'adressage

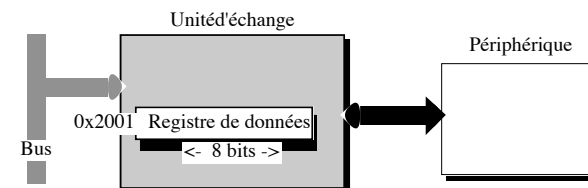
- Les registres des unités d'échange sont repérés par une adresse :
  - dans l'espace mémoire  $\Rightarrow$  espace d'adressage unique,
  - dans un espace d'adressage propre (numérotation des périphériques...)  $\Rightarrow$  espaces d'adressage séparés

## Espace d'adressage unique



- Accès banalisés : toute la gestion des E/S se fait par écritures et lectures en mémoire.

- exemple :



code C pour récupérer ce qui est dans le registre de données :

```
char i;
```

## Introduction

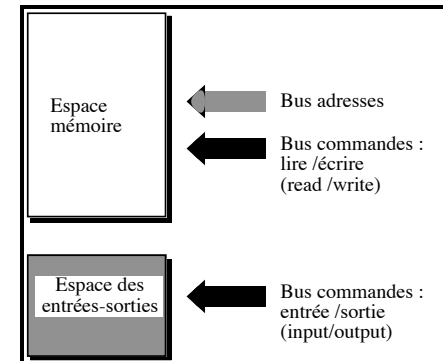
---

```
char * PtrDonnees = (char *)0x2001;  
i = *PtrDonnees;
```

## Introduction

---

### ***Espaces d'adressage séparés***

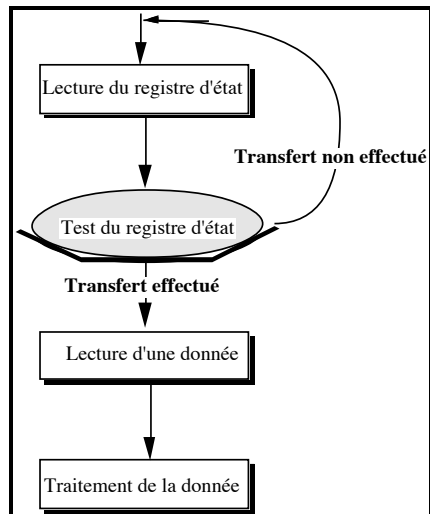


- La commande des unités d'échange se fait via des lignes spécifiques du bus de contrôle (input, output), différentes de celles qui pilotent les accès mémoire (read, write).

## Modes d'échange : interrogation

- interrogation ou encore :

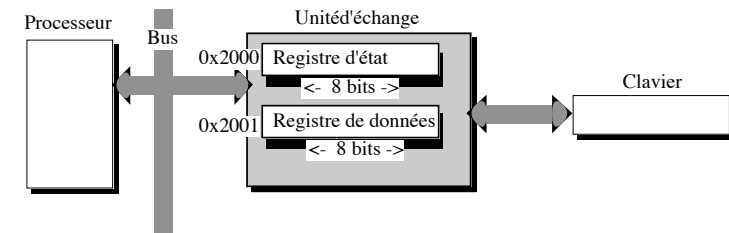
*polling* , test d'état, attente active, *busy waiting*;



- Problème : mobilise l'unité centrale pour les E/S. Utilisé dans les machines "bas de gamme".

## E/S par interrogation : exemple

- Sur cette machine à espace d'adressage unique, lire un caractère frappé au clavier en mode *polling*.



```

/*
 bit poids fort du registre d'état :
 1 --> carac. présent, 0 --> carac. absent
*/

char LireClav (void);
{
 char *Etat = 0x2000; /* adresse du registre d'état */
 char *Donnée = 0x2001; /* adresse du reg. de données */

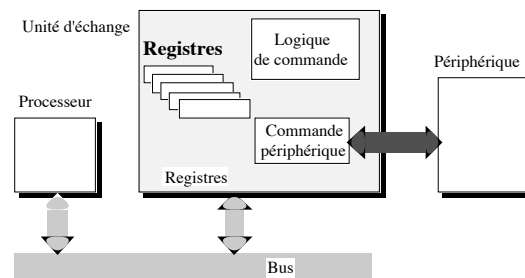
 while (*Etat > 0) { }; /* attente active */

 *Etat = 0x7f & (*Etat); /*bit "present" remis à 0 */

 return (*Donnée)
}
  
```

## **Accès direct à la mémoire (DMA - Direct Memory Access)**

- Principe : l'unité d'échange va lire ou écrire dans la mémoire sans passer par l'unité centrale.



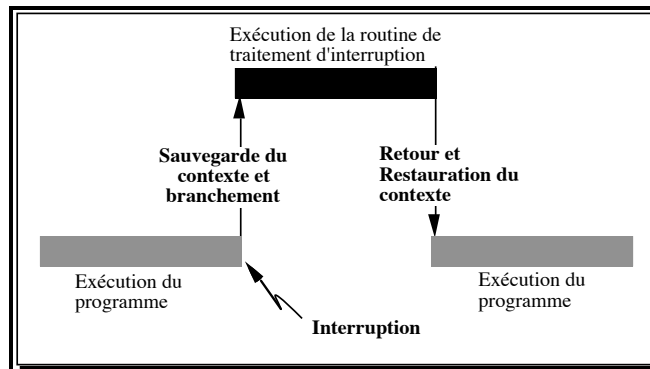
- Quand l'unité d'échange dispose d'une donnée :
  - elle demande le bus ;
  - sur réception d'un message d'acquittement du processeur, elle **prend le contrôle du bus**.
- Avantage : libère le processeur de la gestion des E/S.
- Problème : conflit d'accès au bus.
  - solution : "vol de cycles" : l'unité d'échange ne prend le bus que lorsqu'elle **effectue** un transfert

## **E/S gérées par interruptions**

- Principe : l'UC est prévenue par **un signal** (variation de tension sur une ligne entrant dans l'UC) que l'événement attendu est arrivé,
- Ceci évite à l'unité centrale de faire de l'attente active,
- L'interruption provoque l'arrêt du travail courant du processeur en vue d'exécuter une autre tâche.
- En pratique :
  - le cycle de chaque instruction commence par un test d'une bascule mémorisant les variations de tension sur la ligne d'interruption.
  - S'il y a une interruption, on va la traiter.
  - Sinon on poursuit par l'exécution de cette instruction.

## **Entrées-sorties: interruptions**

- Déroutement vers la procédure de traitement de l'interruption :



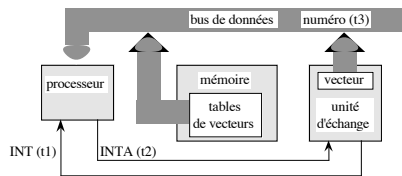
- On rappelle le problème de la gestion par interruption :

- QUI a émis le signal ?
- QUE faire ?

- L'unité d'échange envoie au processeur un numéro sur le bus de données : son vecteur d'interruption, indice dans une table d'adresses de fonctions.

## Entrées-sorties: interruptions

- schéma de base :



- L'appel d'un programme traitant une interruption est géré comme un appel de procédure classique.
- Mais cet appel se fait de façon asynchrone (cf. les signaux en C), et utilise une pile spéciale.

## Entrées-sorties: interruptions

- émission du signal d'interruption (t1),
- l'instruction en cours se termine, (**ATOMICITE** d'une instruction)
- l'adresse de l'instruction suivante et le contenu des registres sont **sauvegardés** dans la pile d'interruption (cf. appel de procédure),
- les interruptions de niveau inférieur sont masquées,
- le processeur envoie un signal d'acquiescement sur une ligne de commande du bus (temps t2),
- le processeur lit sur le bus le "vecteur d'interruption",  $N$ , (réponse à QUI ?) qui est un index dans la table des vecteurs d'interruption  $TVI$ ,
- la fonction (réponse à QUE faire ?) dont l'adresse est en  $TVI[N]$  est exécutée,
- restauration des registres , démasquage des interruptions et retour.

### **Entrées-sorties, exemple : interruption**

- Exemple : procédure de gestion de l'interruption venant du clavier.

Auparavant on avait initialisé l'entrée correspondante de la table des vecteurs d'interruptions TVI par :

```
TVI[KBD] = KbdInt;
```

- Ci-dessous, fonction de gestion de l'interruption :

```
void KbdInt ();  
{  
    char *Ptr1 = 0x2000; /* adresse du registre d'état */  
    char *Ptr2 = 0x2001; /* adresse registre données */  
    Save ();           /* sauver le contexte */  
    Disable ();        /* interdire les interruptions  
                        de niveau inférieur ou égal */  
    *Ptr1 = PRESENT;   /* mise à jour registre d'état */  
    Tamp = *Ptr2;      /* mise à jour du tampon d'E/S */  
    Enable ();         /* réautoriser les interruptions  
                        de niveau inférieur ou égal */  
    V (Sem_Clav)      /* libérer le verrou */  
    Restaure ();       /* restaurer le contexte */  
}
```

On suppose ici que les différents modules d'entrées-sorties communiquent comme suit :

- le programme gérant l'interruption prend les informations dans les registres du périphérique, les dépose dans un tampon Tamp et fait une opération V sur le verrou associé à ce périphérique. Ce verrou est initialisé à 0 quand la machine démarre.
- la fonction de plus haut niveau (c'est à dire du type read (... ,TAB, N)) fait une opération P sur le verrou associé au périphérique. Le processus utilisant cette fonction sera donc bloqué tant que le programme gérant l'interruption n'aura pas été déclenché par un événement externe, ici la frappe du clavier. Si le verrou est

passant (compteur différent de zéro), alors la fonction va prendre les informations décodées dans le tampon Tamp et les met dans le tableau utilisateur TAB.  
Ici Tamp est une variable globale et TAB est réduit à un caractère.

### ***Entrées-sorties, exemple : interruption***

- Fonction de lecture d'un caractère frappé au clavier :

```
int ReadKbd (char * TAB);
{
  if (BLOCKING_IO)
  {
    /*
     Si le caract. n'a pas été frappé, le processus appelant
     read passe à l'état BLOQUE :
     */
    P (Sem_Clav);

    *TAB = Tamp;

    return (1);
  }
  else
  {
    /* Ici code pour les E-S non bloquantes */
  }
}
```

Chaque fonction d'entrée-sortie peut être BLOQUANTE ou non.

Par défaut elle est bloquante. Dans le cas non-bloquant, si aucun caractère n'est disponible, la fonction renvoie un code spécial et le processus passe à l'instruction suivante.

### ***Entrées-sorties: interruptions exemples***

- mécanisme général, le processeur est très souvent interrompu :
  - E/S réseau,
  - E/S disques,
  - E/S clavier,
  - souris,
  - problèmes logiciels (segmentation fault...).

### **La table des vecteurs : pointeurs de fonctions**

- Définition et utilisation de la table des vecteurs d'interruptions en C :

- Définition :

```
#define NVECT 256;
/*-----
On rappelle la syntaxe pour définir un
pointeur (ici Ptrfonc) sur une fonction
renvoyant un int:
        int (*Ptrfonc)();
-----*/
```

```
int (*TVI[NVECT]) ();
```

- Initialisations :

```
TVI[34] = disk_int; /* interr. disque en 34 */
TVI[67] = mouse_int; /* interr. souris en 67 */
```

- Appel pour l'interruption N :

```
TVI[N]();
```

### **Mécanismes voisins : les exceptions (1)**

- Événement provoqué par une instruction **propre** au programme (*overflow*, division par zéro, etc). Alors que les interruptions sont provoquées par des événements **externes** au programme.
- Traitement immédiat dans le contexte du programme (certaines exceptions sont masquables).
- Les interruptions logicielles (software interrupts) sont traitées comme des exceptions.
- On distingue 3 types d'exceptions : les trappes, les fautes et les abandons.

## **Mécanismes voisins : les exceptions (2)**

- Exception de type trappe (trap) :
  - provoquée par les erreurs arithmétiques, erreur de codes instructions...
  - La prise en compte s'effectue **après** l'exécution de l'instruction qui a causé le problème.
  - L'exécution peut reprendre à la fin du traitement de la trappe.
- Exception de type faute (fault) :
  - l'exception est détectée et traitée **avant** l'exécution de l'instruction qui cause problème (défaut de page).
  - L'exécution peut reprendre une fois la faute corrigée
- Exception de type abandon (abort) :
  - tentative d'accès à une zone mémoire protégée, erreur matérielle, etc.
  - l'exécution est **définitivement arrêtée**.

## **Evaluation des performances**

- un ordinateur ne se **réduit pas** à un processeur, c'est aussi un ou plusieurs bus, de la mémoire, et beaucoup de logiciel,
- benchmarks pour tester :
  - le processeur (Mips et Mflops),
  - les capacités de bus (débit, largeur),
  - la vitesse de la mémoire (latence),
  - accès disque, réseau (débit),
  - le COMPILATEUR (optimisation du code).