SIPRO: un PROcesseur SImple

Nicolas Bedon

8 mars 2022

1 Introduction

L'architecture d'un ordinateur est centrée autour d'un composant : le processeur. Le processeur est un automate sachant réaliser des opérations élémentaires. A l'intérieur d'un processeur se trouvent de petites mémoires ayant toutes la même taille, de l'ordre de quelques bits. Ces mémoires sont appelées *registres*. Un *processeur n bits* est un processeur dont la taille de chaque registre est n bits. Un *mot machine* est un groupe de bits de même taille qu'un registre. Les opérations élémentaires que sait exécuter le processeur sont souvent réalisées sur des données contenues dans ces registres. Par exemple, pour faire une addition, le processeur exécute l'instruction suivante : `add ax,bx`, où `ax` et `bx` sont les noms symboliques de deux registres du processeur. Pour réaliser cette opération, le processeur regarde les contenus des registres `ax` et `bx`, en fait la somme, et stocke le résultat dans le registre `ax`.

La taille d'un registre étant petite et les registres étant eux-mêmes en petits nombres, un processeur peut accéder à un autre composant essentiel de l'ordinateur pour y stocker ou y lire des données en nombre ou taille importants : la *mémoire centrale*. La mémoire centrale est une suite de cellules indicées. La *cellule* (*byte* en anglais) est une entité de mémoire de taille constante (par exemple, 8 bits). L'indice d'une cellule s'appelle son *adresse*. Certaines instructions permettent au processeur d'amener le contenu de cellules mémoire dans un registre, et inversement, de copier le contenu d'un registre dans la mémoire centrale. Ces instructions spécifient (directement ou indirectement) l'adresse des cellules et le nom du registre concernés. Par exemple, l'instruction (*load byte*) `loadb ax,bx` copie dans `ax` le contenu de la cellule dont l'adresse est dans `bx`. Des instructions permettent également de mettre une constante dans un registre.

Pour le processeur, un programme est une suite d'instructions élémentaires à exécuter. Les instructions à exécuter sont stockées dans la mémoire centrale, codées avec des valeurs entières d'octets : les *opcodes*. On peux imaginer, par exemple, que le codage de l'instruction `add ax, bx` est réalisé par la suite de valeurs `20 04 05`. La valeur `20` représente le code de l'instruction d'addition, les valeurs `04` et `05` représentent respectivement les registres `ax` et `bx`. Il est très difficile pour un être humain de programmer le processeur en utilisant directement des opcodes : il faut en effet se souvenir du code de chaque instruction, et de la signification et du codage des opérandes éventuels de l'instruction. On utilise plutôt pour programmer un processeur un langage appelé *assembleur*. Avec l'assembleur, le programmeur édite un fichier texte et y met la suite des instructions élémentaires composant son programme, sous la forme `add ax,bx`, par exemple, à raison d'une instruction élémentaire par ligne. Le programmeur procède à la traduction de son fichier texte en suite d'opcodes en utilisant un programme appelé *programme d'assemblage*. Le résultat du programme d'assemblage constitue le *programme assemblé*.

Nous appellerons *SIPRO* le processeur simplifié décrit dans ce document.

2 SIPRO

2.1 Registres

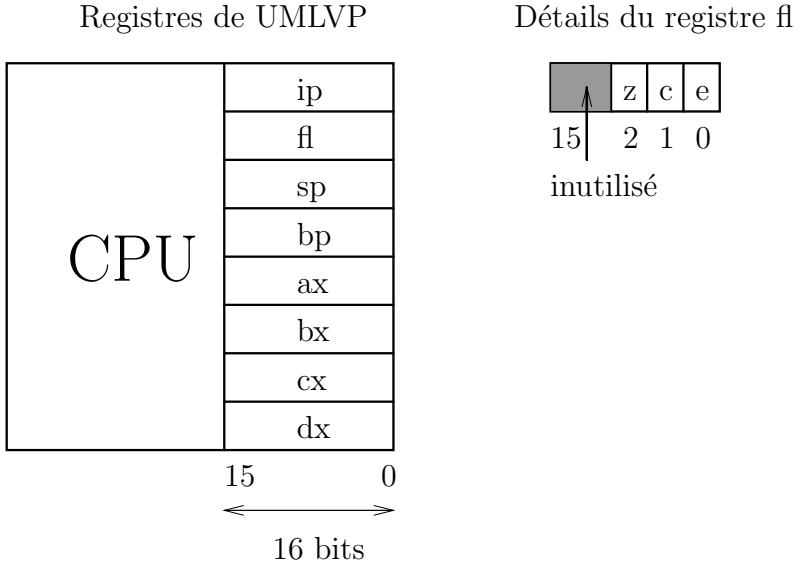
Bien que n'existant pas, SIPRO a une architecture et un comportement très proches de certains processeurs réels.

SIPRO est un processeur 16 bits. Le codage des entiers positifs ou nuls se fait directement par leur écriture en base 2, et le codage des entiers négatifs par un complément à deux. Le codage des caractères se fait en utilisant le code ASCII étendu. L'ordre de stockage des octets d'un mot machine est gros boutiste : le premier octet contient les bits de poids fort. Pour simplifier nous ne manipulerons pas de réels.

SIPRO est capable de manipuler une pile de mots machines, dont les valeurs sont stockées dans la mémoire centrale. Le processeur repère la localisation de cette pile dans la mémoire centrale à l'aide de deux adresses, celles du fond de pile et du sommet de pile. Ces deux adresses sont stockées dans deux registres du processeur, que nous appellerons *registres de pile*, et dont les noms symboliques sont `bp` (pour le fond de pile) et `sp` (pour le sommet de pile).

SIPRO mémorise l'adresse de l'opcode de la prochaine instruction à exécuter dans un registre appelé *registre d'instruction*, de nom symbolique `ip`.

Certaines opérations élémentaires exécutées par le processeur peuvent générer, en dehors de leur résultat, des informations qui peuvent être utiles au bon déroulement de la suite du programme. Par exemple, il est possible, lors de l'exécution d'une addition, que se produise un dépassement de capacité : le résultat de l'addition est alors erroné. Ce genre d'information est signalé à travers un registre nommé *registre de flags*, et dont le nom symbolique est `fl`. Les bits de ce registre ont une signification particulière. Le bit de poids le plus faible, nommé `e` (pour "error"), sert à indiquer que l'exécution de la dernière instruction a provoqué une erreur. C'est le cas, par exemple, lors d'une division par 0. Le bit de poids immédiatement supérieur, nommé `c` (pour "carry"), sert à propager une retenue sortante de la dernière opération. Ce bit peut être positionné, par exemple, par l'opération d'addition. Le bit de poids immédiatement supérieur à `c`, nommé `z` (pour "zero"), sert à indiquer que le résultat de la dernière opération exécutée est nul, et est également utilisé par les opérations de comparaisons. Les autres bits du registre `fl` ne sont pas utilisés, ils doivent toujours être positionnés à 0.



2.2 Les accès à la mémoire

2.2.1 Adressage

Pour simplifier, les adresses des cellules mémoire sont numérotées et représentées de la même façon par le processeur et par la mémoire. L'adresse d'une cellule mémoire tient dans un registre et est codé par sa représentation en base 2. On est donc dans le cas le plus simple possible. Par exemple, si le registre `bx` contient la valeur 2 codée en base 2, alors l'instruction `loadb ax,bx` copiera le contenu de la cellule 2 de la mémoire dans le registre `ax`.

2.2.2 Accès à la mémoire

Le processeur communique avec la mémoire en utilisant un *bus*. Ce bus est une série de liaisons électriques entre les deux composants. La taille du bus, en nombre de bits, est définie comme le nombre maximum de bits pouvant transiter en même temps entre la mémoire et le processeur. Dans notre cas, la taille du bus sera de 16 bits. Le processeur dispose de deux catégories d'opérations d'accès (en lecture ou en écriture) à la mémoire. La première catégorie permet de lire ou d'écrire une cellule, soit 8 bits, dans la mémoire, à une adresse donnée. La seconde catégorie permet de lire ou d'écrire un mot, soit deux cellules (16 bits), dans la mémoire, à une adresse donnée. Il est clair que pour écrire ou lire un mot dans la mémoire à une adresse donnée, le processeur a avantage à utiliser les opérations de la seconde catégorie : en effet, lire ou écrire 16 bits d'un coup est plus rapide que de lire ou écrire les 8 premiers bits, puis de recommencer l'opération sur les 8 suivants. Aussi, pour exécuter certaines instructions, le processeur `SIPRO` utilisera les opérations d'accès à la mémoire de la seconde catégorie. Ces instructions sont signalées dans la documentation des instructions. Des circuits spéciaux sont intégrés dans les puces des mémoires pour augmenter la vitesse des opérations de lecture/écriture de mots. Ces circuits utilisent la propriété que l'adresse du mot est un multiple de la taille d'un mot : cette propriété est connue sous le nom de propriété d'*alignement*. Si jamais l'adresse du mot qu'on essaie de lire ou d'écrire par une opération de la seconde catégorie ne vérifie pas cette propriété d'alignement, alors les puces de mémoire provoquent une erreur matérielle, appelée "Erreur de bus". Cette erreur est en général détectée par le système d'exploitation de l'ordinateur, qui arrête le programme l'ayant provoqué. L'émulateur simule cette propriété de la mémoire.

2.3 Instructions

Nous présentons maintenant le jeu d'instructions élémentaires qu'est capable d'exécuter SIPRO. Les opcodes, comme toutes les constantes de ce document, sont donnés en hexadécimale. L'opcode d'une instruction ne dépend pas de ses opérandes. Les registres en argument des instructions sont identifiés par un code, de taille 8 bits : 00 : `ip`, 01 : `fl`, 02 : `sp`, 03 : `bp`, 04 : `ax`, 05 : `bx`, 06 : `cx`, 07 : `dx`.

2.3.1 Manipulation de bits

shiftr

Nom : shiftr Opcode : 10

Syntaxe : `shiftr <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx` ou `dx`.

Taille totale de l'instruction : 2 cellules (1 pour l'opcode, 1 pour identifier le registre).

Sémantique : décale d'un bit vers la droite tous les bits du registre `registre`. Le bit perdu est mis dans le bit `c` du registre `fl`. Le bit introduit à gauche est 0.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$ et `fl` contient $(0000000000000111)_2$. Après exécution de `shiftr ax`, `ax` contient $(0110100010101001)_2$ et `fl` contient $(0000000000000010)_2$: les bits `z` et `e` ont été remis à zéro, et le bit `c` contient le bit anciennement le plus à droite de `ax` qui a été perdu par le décalage.

shiftl

Nom : shiftl Opcode : 11

Syntaxe : `shiftl <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx` ou `dx`.

Taille totale de l'instruction : 2 cellules (1 pour l'opcode, 1 pour identifier le registre).

Sémantique : décale d'un bit vers la gauche tous les bits du registre `registre`. Le bit perdu est mis dans le bit `c` du registre `fl`. Le bit introduit à droite est 0.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$ et `fl` contient $(0000000000000111)_2$. Après exécution de `shiftl ax`, `ax` contient $(1010001010100110)_2$ et `fl` contient $(0000000000000010)_2$:

les bits `z` et `e` ont été remis à zéro, et le bit `c` contient le bit anciennement le plus à gauche de `ax` qui a été perdu par le décalage.

and

Nom : and Opcode : 15

Syntaxe : `and <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise un ET logique bit à bit entre le contenu des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, tous les bits de `fl` sont mis à zéro.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$, `bx` contient $(0101111010101010)_2$ et `fl` contient $(0000000000000010)_2$. Après exécution de `and ax,bx`, `ax` contient $(0101000000000010)_2$, `bx` est inchangé, `fl` contient $(0000000000000000)_2$.

or

Nom : or Opcode : 16

Syntaxe : `or <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise un OU logique bit à bit entre le contenu des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, tous les bits de `fl` sont mis à zéro.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$, `bx` contient $(0101111010101010)_2$ et `fl` contient $(0000000000000010)_2$. Après exécution de `or ax,bx`, `ax` contient $(1101111111111011)_2$, `bx` est inchangé, `fl` contient $(0000000000000000)_2$.

xor

Nom : xor Opcode : 17

Syntaxe : `or <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise un OU EXCLUSIF logique bit à bit entre le contenu des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, tous les bits de `fl` sont mis à zéro.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$, `bx` contient $(0101111010101010)_2$ et `fl` contient $(0000000000000010)_2$. Après exécution de `or ax,bx`, `ax` contient $(1000111111111001)_2$, `bx` est inchangé, `fl` contient $(0000000000000000)_2$.

not

Nom : not Opcode : 1A

Syntaxe : `not <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules (1 pour l'opcode, 1 pour identifier le registre).

Sémantique : réalise un NON logique bit à bit sur le contenu du registre. Après exécution, tous les bits de `fl` sont mis à zéro.

Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$ et `fl` contient $(0000000000000010)_2$. Après exécution de `not ax`, `ax` contient $(0010111010101100)_2$ et `fl` contient $(0000000000000000)_2$.

2.3.2 Opérations arithmétiques

Les opérations arithmétiques sont des opérations entières, avec un codage des valeurs négatives par complément à deux. Pour toutes les opérations arithmétiques, les contenus des registres sont

interprétés comme des valeurs signées. Les opérations arithmétiques modifient la valeur de leur premier argument (registre), dans lequel elles mettent leur résultat. En cas d'erreur (dépassement de capacité), la valeur de cet argument au retour de l'opération est indéterminée.

add

Nom : add Opcode : 20

Syntaxe : `add <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `sp`, `bp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise la somme des contenus des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, le bit `c` de `fl` contient un éventuel bit de retenue (zéro si pas de retenue), `e` est positionné si et seulement si un dépassement de capacité s'est produit, et `z` est positionné si et seulement si le résultat est nul. Exemple : Avant exécution, `ax` contient $(1101000101010011)_2$, `bx` contient $(0101111010101010)_2$ et `fl` contient $(0000000000000010)_2$. Après exécution de `add ax,bx`, `ax` contient $(0010111111111101)_2$, `bx` est inchangé, `fl` contient $(0000000000000010)_2$. Le bit `z` est à zéro car le résultat est non nul, le bit `c` est à 1 car une retenue s'est propagée en sortie de l'addition, et le bit `e` est à zéro car il n'y a pas eu de dépassement de capacité malgré la propagation de la dernière retenue : en effet, les deux opérandes ne sont pas de même signe, et lors de la somme sur les deux bits de poids le plus fort, le bit de retenue entrant est égal au bit de retenue sortant.

sub

Nom : sub Opcode : 21

Syntaxe : `sub <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `sp`, `bp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise la différence des contenus des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. La différence de deux valeurs doit se comporter exactement comme la somme de la première valeur et de l'opposé de la seconde. Attention cependant, il existe un cas particulier : la plus petite valeur négative représentable dans un registre n'a pas d'opposé représentable dans un registre ! On fera attention à ce que le résultat soit quand même correct dans ce cas.

mul

Nom : mul Opcode : 22

Syntaxe : `mul <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise le produit des contenus des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, le bit `c` de `fl` contient 0, `e` est positionné si et seulement si un dépassement de capacité s'est produit, et `z` est positionné si et seulement si le résultat est nul.

div

Nom : div Opcode : 23

Syntaxe : `div <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : réalise la division entière des contenus des deux registres. Le résultat est placé dans le premier registre. Le contenu du second registre est inchangé. Après exécution, le bit `c` de `fl` contient 0, `e` est positionné si et seulement si une division par 0 s'est produite, et `z` est positionné si et seulement si le résultat est nul. On rappelle qu'en C la division entière avec un

opérande négatif est machine dépendante. On impose, dans la division entière de `SIPRO`, que le résultat soit celui de la division entière des valeurs absolues, avec le signe correctement rétablit.

2.3.3 Chargement et déchargement

Les opérations de chargement et déchargement permettent de mettre des valeurs dans des registres, à partir de valeurs prisent dans d'autres registres ou de valeurs prisent dans la mémoire. Elles permettent également de copier dans la mémoire des valeurs contenues dans des registres.

cp

Nom : cp Opcode : 30

Syntaxe : `cp <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : Copie dans le premier registre la valeur contenue dans le second registre. Le second registre reste inchangé. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction.

loadw

Nom : loadw Opcode : 31

Syntaxe : `loadw <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : Copie dans le premier registre le mot machine dont l'adresse est dans le second registre. Le second registre reste inchangé. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction. Pour exécuter cette instruction, le processeur accède à la mémoire en utilisant la fonction lui permettant de lire un mot.

storew

Nom : storew Opcode : 32

Syntaxe : `storew <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : Copie le mot machine contenu dans le premier registre dans la mémoire à l'adresse spécifiée dans le second registre. Les deux registres restent inchangés. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction. Pour exécuter cette instruction, le processeur accède à la mémoire en utilisant la fonction lui permettant d'écrire un mot.

loadb

Nom : loadb Opcode : 33

Syntaxe : `loadb <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : Copie dans les bits de poids faible du premier registre la cellule dont l'adresse est dans le second registre. Les autres bits du premier registre sont tous mis à zéro. Le second registre reste inchangé. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction.

storeb

Nom : storeb Opcode : 34

Syntaxe : `storeb <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules (1 pour l'opcode, 1 pour identifier le premier registre, 1 pour identifier le second registre).

Sémantique : Copie les 8 bits de poids faible contenus dans le premier registre dans la mémoire à l'adresse spécifiée dans le second registre. Les deux registres restent inchangés. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction.

const

Nom : const Opcode : 35

Syntaxe : `const <registre>,<valeur>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`, une constante de même taille qu'un mot machine donnée en base 10.

Taille totale de l'instruction : variable ! 1 cellule pour l'opcode, 1 cellule pour l'identifiant de registre, 2 cellules (16 bits) pour la valeur, qui est un mot machine. L'adresse à laquelle est stockée ce mot machine dans la mémoire doit être d'alignement valide pour la mémoire (ça doit être un multiple de la taille d'un mot machine (2), en octets). Ainsi, la taille totale de l'instruction peut varier de 4 à 5 cellules.

Sémantique : Copie `valeur` dans le registre. Les bits du registre `fl` sont tous remis à zéro après exécution de l'instruction.

Exemple : Supposons que le contenu de la mémoire soit `00000035064DFF00`, et que le registre IP, qui désigne l'adresse de la prochaine instruction à exécuter, contienne 03. L'opcode de la prochaine instruction à exécuter est donc 35, qui correspond à l'instruction `const`. Lors de l'exécution de l'instruction `const`, le processeur va charger le contenu de la cellule suivante, qui contient l'identifiant du registre concerné : ici 06, le registre concerné est donc `cx`. Le processeur va alors demander à la mémoire la valeur du mot machine suivant immédiatement, en utilisant la fonction de la mémoire lui permettant d'y lire un mot. L'identifiant de registre étant à l'adresse 04, la prochaine adresse de mot machine valide (i.e. le prochain multiple de 2), est 06. Le processeur va donc charger la constante FF00 dans le registre `cx`, et augmenter le pointeur d'instruction de manière à ce qu'il désigne l'opcode de la prochaine instruction à exécuter (ici 08). Dans cet exemple, la valeur 4D, à l'adresse 5, n'a pas d'importance : n'importe quelle autre valeur en remplacement de 4D aurait donné le même résultat. La valeur à l'adresse 5 ne peut pas être utilisée, car const doit lire un mot machine et, par les contraintes d'alignement, les mots machines ne peuvent pas être lus/écrits à des adresses impaires.

2.3.4 Opérations de piles

Le processeur dispose de deux registres, `bp` et `sp`, qui lui permettent de gérer une pile de mots machine. Les registres `bp` et `sp` contiennent respectivement l'adresse en mémoire du mot machine en bas de la pile, et l'adresse du mot mémoire en haut de la pile. Comme nous le verrons par la suite, l'utilisation principale de la pile est de permettre à un programme assembleur de mettre de coté des valeurs pendant l'exécution d'une fonction, ou de passer des valeurs en argument des fonctions. L'utilisation de la pile permet en particulier d'écrire des fonctions récursives.

Les opérations de manipulation de pile sont au nombre de deux.

push

Nom : push Opcode : 40

Syntaxe : `push <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Copie le mot machine contenu dans le registre `<registre>` à l'adresse désignée par `sp` plus la taille d'un mot machine, puis augmente `sp` de la taille d'un mot machine. Remet tous les bits de `fl` à zéro. La valeur de `registre` reste inchangée. Bien faire attention à l'ordre, à cause du cas particulier de `push sp`, qui doit sauver la valeur de `sp` sur la pile avant le changement de `sp`. Pour exécuter cette instruction, le processeur utilise la fonction permettant d'écrire un mot machine dans la mémoire.

pop

Nom : pop Opcode : 41

Syntaxe : `pop <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Si `sp`>=`bp`, diminue `sp` de la taille d'un mot machine, puis copie dans le registre `registre` le mot machine contenu à l'adresse désignée par `sp` avant d'être diminué. Remet ensuite les bits de `fl` à zéro. Bien faire attention à l'ordre, à cause du cas particulier de `pop sp` : après exécution de `pop sp`, le registre `sp` doit contenir la valeur qui était au sommet de la pile avant exécution. Pour exécuter cette instruction, le processeur utilise la fonction permettant de lire un mot machine dans la mémoire. Si `sp`<`bp`, se contente de positionner le bit `e` de `fl` à 1.

2.3.5 Comparaison

Les instructions de comparaison permettent de comparer deux mots machines entre eux.

cmp

Nom : cmp Opcode : 50

Syntaxe : `cmp <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules : 1 pour l'opcode, 1 pour le premier identifiant de registre, 1 pour le second identifiant de registre.

Sémantique : Après exécution, le bit `c` de `fl` est positionné si et seulement si les deux mots machines contenus dans les deux registres sont identiques. Le bit `z` de `fl` est positionné si et seulement si les deux registres contiennent tous les deux la valeur 0. Le bit `e` de `fl` est toujours remis à zéro. Ne change pas la valeur des deux registres.

uless

Nom : uless Opcode : 51

Syntaxe : `uless <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules : 1 pour l'opcode, 1 pour le premier identifiant de registre, 1 pour le second identifiant de registre.

Sémantique : Les mots machine contenus dans les deux registres sont interprétés comme des valeurs non signées. Après exécution, le bit `c` de `fl` est positionné si et seulement si le contenu du premier registre est strictement inférieur au contenu du second. Les autres bits de `fl` sont remis à zéro. Ne change pas la valeur des deux registres.

sless

Nom : sless Opcode : 52

Syntaxe : `sless <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`, `bp`, `sp`.

Taille totale de l'instruction : 3 cellules : 1 pour l'opcode, 1 pour le premier identifiant de registre, 1 pour le second identifiant de registre.

Sémantique : Les mots machine contenus dans les deux registres sont interprétés comme des valeurs signées. Après exécution, le bit `c` de `fl` est positionné si et seulement si le contenu du premier registre est strictement inférieur au contenu du second. Les autres bits de `fl` sont remis à zéro. Ne change pas la valeur des deux registres.

2.3.6 Branchements

Les instructions de branchement permettent de réaliser des sauts pendant l'exécution d'un programme. On en trouve de deux catégories : celles qui s'apparentent à des `goto`, et celles qui s'apparentent à des appels (et des retours) de fonctions. Attention : les instructions de saut dont le nom commence par *jmp* ont toutes un comportement exceptionnel : elles ne modifient pas `fl`.

Le processeur `SIPRO` dispose de deux instructions de branchement permettant d'exécuter du code d'une manière qu'on peut assimiler à un appel de fonction. L'instruction

`call <addresse de la première instruction de la fonction>`

permet de réaliser cet appel de fonction. L'instruction `ret` permet de réaliser le retour de fonction. L'instruction qui est exécutée immédiatement après le `ret` est celle suivant immédiatement le `call` ayant appelé la fonction. Le mécanisme permettant de retrouver, après un `ret`, l'adresse de l'instruction suivant le `call` est le suivant. L'instruction `call` utilise la pile : pendant son exécution, le processeur place en sommet de pile l'adresse de l'instruction qui suit immédiatement le `call`. Pour exécuter l'instruction `ret`, le processeur récupère cette adresse, qu'il suppose toujours au sommet de la pile. Il connait ainsi l'adresse de la prochaine instruction à exécuter après le `ret`. Attention cependant : pour la bonne marche de ce procédé, le programmeur ne doit pas avoir retiré par inadvertance cette adresse de la pile, ni ajouté quelque chose en sommet de pile pendant la fonction (ou alors, il a retiré ce quelque chose avant le `ret`). Au début de l'exécution de la fonction, i.e. juste après l'éxécution de l'instruction `call`, le sommet de la pile contient l'adresse de l'instruction qu'il faudra exécuter après le `ret`. À la fin de la fonction, juste avant d'exécuter l'instruction `ret`, le sommet de la pile est supposé contenir cette même valeur.

Le passage d'arguments aux fonctions peut se faire de deux manières différentes. C'est au programmeur de choisir celle qui lui convient le mieux. La première manière consiste à placer les valeurs qu'on veut passer en argument à la fonction dans des registres juste avant l'appel. La fonction récupèrera alors ces valeurs dans les registres. Cette façon de procéder suppose que le nombre de paramètres est plus petit ou égal au nombre de registres. La seconde manière de procéder utilise la pile : avant d'appeler la fonction, on met les arguments sur la pile. La fonction ira alors récupérer ces valeurs. Attention, on rappelle qu'au début de l'exécution de la fonction, le sommet de la pile contient l'adresse de retour, et il faut que cette valeur se trouve également au sommet de la pile juste avant le `ret`. Aussi, en général, pour récupérer les arguments dans la pile, qui sont sous cette adresse de retour, on n'utilise pas l'opération de dépilement : on va directement lire le contenu de la pile, sans dépiler.

Le passage de la valeur de retour peut se faire de manière similaire au passage des arguments. Cependant, comme la valeur de retour est en général unique, pour plus de simplicité le rpogrammeur peut se fixer une convention du style : une fonction place sa valeur de retour dans le registre `ax` avant le `ret`.

L'exemple donné plus loin dans le sujet (calcul de factorielle récursif) illustre ce mécanisme d'appel de fonction en lui passant ses arguments par l'intermédiaire de la pile. Juste vant d'exécuter la première instrution de la fonction `factorielle`, au sommet de pile se trouve l'adresse de l'instruction qui suit immédiatement le `call` ayant exécuté la fonction, et juste sous cette adresse se trouve dans la pile l'argument de la fonction. Les toutes première instruction de la fonction ont pour objet de récupérer cet argument. On observe que pour ce faire on n'utilise pas d'opération de dépilement. À son retour, le résultat de la fonction `factorielle` se trouve dans le registre `ax`.

Afin de rendre l'émulateur réellement utilisable, des opérations de branchement particulières permettant de réaliser des entrées/sorties du genre `scanf` ou `printf` ont été ajoutées. Ces fonctions n'existent pas dans un processeur réel : c'est au programmeur de les écrire, et elles sont très complexes. Dans l'implantation de l'émulateur, l'exécution d'une de ces instructions revient tout simplement à faire un `printf` ou un `scanf`, en prenant éventuellement quelques précautions.

Les instructions de branchement sont également appelées “instructions de saut”.

jmp

Nom : jmp Opcode : 60

Syntaxe : `jmp <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Copie le contenu de `registre` dans `ip`.

jmpz

Nom : jmpz Opcode : 61

Syntaxe : `jmpz <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Copie le contenu de `registre` dans `ip` si le bit `z` de `fl` est positionné. Ne change ni la valeur du registre, ni `fl`.

jmpc

Nom : jmpc Opcode : 62

Syntaxe : `jmpc <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Copie le contenu de `registre` dans `ip` si le bit `c` de `fl` est positionné. Ne change ni la valeur du registre, ni `fl`.

jmpe

Nom : jmpe Opcode : 63

Syntaxe : `jmpe <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Copie le contenu de `registre` dans `ip` si le bit `e` de `fl` est positionné. Ne change ni la valeur du registre, ni `fl`.

call

Nom : call Opcode : 65

Syntaxe : `call <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Fait l'équivalent d'un `push ip+2` (même si cette dernière instruction n'existe pas, on comprend ce qu'elle réalise), puis copie le contenu de `registre` dans `ip`. Ne modifie pas la valeur du registre. Remet tous les bits de `fl` à zéro.

ret

Nom : ret Opcode : 66

Syntaxe : `ret`

Arguments : aucun

Taille totale de l'instruction : 1 cellule.

Sémantique : Dépile un mot machine, et se branche à l'adresse désignée par ce mot pour l'exécution de la prochaine instruction. Ne modifie pas les bits de `fl`. Pour exécuter cette instruction, le processeur utilise la fonction qui permet de lire un mot machine dans la mémoire à une adresse donnée.

callprintfd

Nom : callprintfd Opcode : 6A

Syntaxe : `callprintfd <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Réalise l'affichage de la valeur du mot machine dont l'adresse est dans `registre`, sous la forme d'un entier signé. Ne modifie pas la valeur du registre. Remet tous les bits de `fl` à zéro. Pour exécuter cette instruction, le processeur utilise une fonction qui permet de lire/écrire un mot machine dans la mémoire à une adresse donnée.

callprintfu

Nom : callprintfu Opcode : 6B

Syntaxe : `callprintfu <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Réalise l'affichage de la valeur du mot machine dont l'adresse est dans `registre`, sous la forme d'un entier non signé. Ne modifie pas la valeur du registre. Remet tous les bits de `fl`

à zéro. Pour exécuter cette instruction, le processeur utilise une fonction qui permet de lire/écrire un mot machine dans la mémoire à une adresse donnée.

callprintfs

Nom : callprintfs Opcode : 6C

Syntaxe : `callprintfs <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Réalise l'affichage de la chaîne de caractères dont l'adresse est dans `registre`. Ne modifie pas la valeur du registre. Remet tous les bits de `fl` à zéro.

callscanfd

Nom : callscanfd Opcode : 6D

Syntaxe : `callscanfd <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Réalise une saisie de valeur de mot machine au clavier, sous la forme d'un entier signé. Positionne le bit `e` de `fl` si et seulement si la saisie s'est mal passée ou la valeur est en dehors des intervalles représentables pour un mot machine de `SIPRO`. Les autres bits de `fl` sont remis à zéro. Après exécution, la valeur saisie se trouve à l'adresse contenue dans `registre`. Pour exécuter cette instruction, le processeur utilise une fonction qui permet de lire/écrire un mot machine dans la mémoire à une adresse donnée.

callscanfu

Nom : callscanfu Opcode : 6E

Syntaxe : `callscanfu <registre>`

Arguments : 1 identifiant de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 2 cellules : 1 pour l'opcode, 1 pour l'identifiant de registre.

Sémantique : Réalise une saisie de valeur de mot machine au clavier, sous la forme d'un entier non signé. Positionne le bit `e` de `fl` si et seulement si la saisie s'est mal passée ou la valeur est en dehors des intervalles représentables pour un mot machine de `SIPRO`. Les autres bits de `fl` sont remis à zéro. Après exécution, la valeur saisie se trouve à l'adresse contenue dans `registre`. Pour exécuter cette instruction, le processeur utilise une fonction qui permet de lire/écrire un mot machine dans la mémoire à une adresse donnée.

callscanfs

Nom : callscanfs Opcode : 6F

Syntaxe : `callscanfs <registre>,<registre>`

Arguments : 2 identifiants de registre : `ax`, `bx`, `cx`, `dx`.

Taille totale de l'instruction : 3 cellules : 1 pour l'opcode, 1 pour le premier identifiant de registre, 1 pour le second identifiant de registre.

Sémantique : Réalise une saisie de chaîne de caractères au clavier. Positionne le bit `e` de `fl` si et seulement si la saisie s'est mal passée (aucun caractère n'a pû être lu). Les autres bits de `fl` sont remis à zéro. Après exécution, la chaîne de caractère saisie se trouve à l'adresse contenue dans le premier registre. Elle se termine par un caractère de fin de chaîne. Le second registre contient le nombre de caractères maximum qui peuvent être saisis, y compris le caractère de fin de chaîne. Contrairement au comportement de la fonction de la bibliothèque standard C `fgets`, l'éventuel caractère `\n` n'est pas stocké dans la chaîne résultat.

2.3.7 Instruction nulle

nop

Nom : nop Opcode : 00

Syntaxe : `nop`

Arguments : aucun

Taille totale de l'instruction : 1 cellule.

Sémantique : Ne fait rien, sauf remettre les bits de `fl` à zéro.

2.3.8 Instruction de fin de programme

end

Nom : end Opcode : FF
Syntaxe : `end`
Arguments : aucun
Taille totale de l'instruction : 1 cellule.
Sémantique : Termine le programme.

3 Le langage d'assemblage de SIPRO

Le langage d'assemblage est conçu pour permettre au programmeur de travailler directement avec les instructions du processeur. Il met à la disposition du programmeur certaines facilités, comme par exemple la gestion automatique d'une table des symboles, pour donner des noms symboliques aux adresses de sauts. Il est également conçu pour que le programme d'assemblage (l'assembleur), qui réalise la traduction d'un fichier contenant les sources d'un programme écrit dans le langage d'assemblage en suite d'octets, soit simple à écrire. L'assembleur se contente de lire les lignes du fichier une par une, dans l'ordre, de traduire la ligne en suite d'octets, et de placer ces octets dans un fichier de sortie, les uns à la suite des autres. Le fichier de sortie de l'assembleur est fourni en entrée du simulateur : il représente directement, sous forme de suite de cellules, l'état de la mémoire, cellule par cellule, avant exécution du programme. Si on se représente la mémoire comme un tableau de cellules, le contenu du fichier de sortie peut être vu comme la sauvegarde brute de ce tableau de cellules. Dans cette mémoire se trouvent à la fois les instructions constituant le programme, et une partie des données sur lesquelles il va travailler. Le fichier de sortie constitue un programme directement exécutable par le processeur. L'assembleur ayant à gérer une table des symboles qui associe un nom sybolique avec une adresse, il peut le cas échéant réaliser deux passes sur les sources du programme : une première pour donner des valeurs à chaque symbole, la seconde pour réaliser l'assemblage proprement dit.

Nous décrivons ici la syntaxe du langage d'assemblage, et le comportement de l'assembleur.
Dans le langage d'assemblage, toute instruction tient sur une ligne. Chaque ligne peut contenir :
— une directive de placement de constante dans la mémoire ;
— un commentaire ;
— un label ;
— une instruction ;
— rien du tout (ligne vide).

3.1 Les directives de placement de constante dans la mémoire

Les lignes contenant une directive de placement de constante dans la mémoire commencent par un "@". Il en existe de deux genres :
— les directives de placement de chaînes de caractères constantes dans la mémoire ;
— les directives de placement de mots machines dans la mémoire.

Une directive de placement de chaîne de caractère a la forme : `@string "Bonjour"`. L'assembleur la comprend comme : prendre le code ASCII de chaque lettre, et les mettre dans l'ordre dans le fichier de sortie, avec un caractère de fin de chaîne (de code ASCII 0).

Une directive de placement d'entier a la forme : `@int -1234`. L'assembleur la comprend comme : coder -1234 en mot machine, et écrire ce mot machine dans le fichier de sortie. En cas d'erreur, comme par exemple une constante trop grande, l'assembleur affiche un message d'erreur, le numéro de ligne et arrête son travail.

L'assembleur doit toujours s'arranger pour placer les constantes dans la mémoire à une adresse compatible avec l'alignement des entiers (même pour les chaînes de caractères).

3.2 Les commentaires

Les commentaires sont des lignes commençant par “;”. Ils n'ont aucun effet sur l'assembleur, qui se contente de les passer.

3.3 Les labels

Les lignes contenant un label commencent par “ :”, qui sont suivis par une suite de caractères alphabétiques, représentant un nom symbolique. Le nom ne peux pas être un nom de registre. L'assembleur ajoute alors ce nom à la table des symboles, dans laquelle il lui associe l'adresse courante (le nombre de cellules que l'assembleur a déjà utilisé dans la mémoire).

Les labels ne peuvent être utilisés qu'en argument de l'instruction `const`.

3.4 Les instructions

Les lignes contenant une instruction commencent par une tabulation. Une ligne ne peut contenir qu'une instruction. Après la tabulation, on trouve le nom de l'instruction, tout en minuscule. Si l'instruction a des opérandes, un espace suit le nom de l'instruction, et la première opérande suit l'espace. Dans le cas de plusieurs opérandes, on les sépare par des virgules. Dans le cas où l'opérande est un registre, on donne son nom. C'est à l'assembleur de determiner l'identifiant du registre en fonction de son nom. Dans le cas où l'instruction est une instruction de saut et où l'opérande est un nom symbolique, l'assembleur doit remplacer le nom par sa valeur. Dans le cas où l'instruction est `const`, le second argument est une valeur numérique.

4 Exemple complet

Nous développons dans cette section un exemple complet, qui comprend :

— le code d'un programme dans le langage d'assemblage, commenté et expliqué ;

— le contenu expliqué du fichier de sortie de l'assembleur.

L'exemple suivant demande un entier à l'utilisateur, calcule sa factorielle de manière récursive, puis l'affiche.

```
; Programme de calcul de la fonction factorielle, par la méthode récursive

; Permet de passer la zone de stockage des constantes
        const ax,debut
        jmp ax

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Début de la zone de stockage des constantes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:chaine1
@string "Rentrez votre nombre:"
:chaine2
@string "Factorielle "
:chaine3
@string " vaut "
:chaine4
@string "\n"
:chaine5
@string "Erreur !"
:valeurinit
@int 0
:resultat
@int 0
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fin de la zone de stockage des constantes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;
; Début réel du code
;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fonction de saisie
; La fonction ne modifie aucun registre, mis à part ax,
; qui au retour de la fonction contient la valeur saisie
; On commence par sauver tous les registres, de cette manière la fonction
; peux les modifier à loisirs, et restaurer les valeurs d'origine avant de
; retourner à la fonction appelante
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:saisie
        push sp
        push bp
        push bx
        push cx
        push dx
; On affiche chaine1
        const ax,chaine1
        callprintfs ax
; On saisie une valeur, qu'on va mettre à l'adresse spécifiée par valeurinit
        const bx,erreur
        const ax,valeurinit
        callscanfu ax
; en cas d'erreur on appelle la fonction de sortie sur erreur
        jmpe bx
; on va maintenant récuperer la valeur stockée à l'adresse valeurinit (désignée par ax)
; pour la mettre dans le registre ax
        cp bx,ax
        loadw ax,bx
; On restaure les registres avant de sortir
; On fait bien attention à faire les pop
; dans l'ordre inverse des push du début de la fonction
        pop dx
        pop cx
        pop bx
        pop bp
        pop sp
; On retourne à la fonction appelante
        ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fin de la fonction de saisie
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fonction calculant récursivement la factorielle du mot machine
; se trouvant sous le sommet de la pile
; L'argument se trouve sous le sommet de la pile
; En effet le sommet de la pile contient l'adresse de retour
; de la fonction
; Le résultat est mis dans ax
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:factorielle
```

```
; On commence par calculer dans cx l'adresse de l'argument
        cp cx,sp
        const bx,2
        sub cx,bx
; C'est fait, maintenant on met la valeur de l'argument dans ax
        loadw ax,cx
; on regarde si on est dans le cas de base
        xor bx,bx
        const dx,casdebase
        cmp ax,bx
        jmpz dx
; on est dans le cas général
        const bx,1
        sub ax,bx
        push ax
        const dx,factorielle
        call dx
        pop dx
; La valeur de retour est dans ax
; On récupère de nouveau la valeur de l'argument
        cp dx,sp
        const bx,2
        sub dx,bx
        loadw bx,dx
        mul ax,bx
        const dx,finfact
        jmp dx
:casdebase
        const ax,1
        const dx,finfact
        jmp dx
:finfact
        ret
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fin de la fonction factorielle récursive
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fonction de sortie sur erreur
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:erreur
        const ax,chaine5
        callprintfs ax
        end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fin de la fonction de sortie sur erreur
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;
; Fonction principale
;;;;;;;;;;;;;;;;;;;;
:debut
        const bp,pile
        const sp,pile
        const ax,2
        sub sp,ax
        const dx,saisie
        call dx
```

```
        const bx,valeurinit
        storew ax,bx
        const bx,chaine2
        callprintfs bx
        const bx,valeurinit
        callprintfu bx
        const bx,chaine3
        callprintfs bx
        push ax
        const bx,factorielle
        call bx
        pop bx
        const bx,resultat
        storew ax,bx
        callprintfu bx
        const bx,chaine4
        callprintfs bx
        end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Fin de la fonction principale
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Début de stockage de la zone de pile
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:pile
@int 0
```

Nous expliquons maintenant de manière détaillée le code de ce programme. Il commence par une zone de données, contenant des chaînes de caractères et des entiers. Cette zone de données est précédée par deux instructions permettant au processeur de passer cette zone de donnée afin de ne pas en interpréter le contenu comme des instructions. Le programme commence réellement à l'instruction labelisée `debut`. Cette instruction, avec les trois suivantes, initialise les deux registres de pile pour qu'ils désignent une pile vide commençant dans la mémoire à un emplacement se situant après les zones mémoires contenant des instructions. La plus petite adresse de cet emplacement est celle de l'entier labelisé `pile`. La pile progressant vers les adresses hautes (en croissant) de la mémoire. on est ainsi certain qu'on peut mettre des données dans la pile sans jamais écraser d'instruction. Le fond de pile utilisé par le programme est repéré par l'entier labelisé `pile`. Les deux premières instructions font en sorte que `sp` et `bp` contiennent tout deux l'adresse de cet entier. La pile ainsi désignée par `sp` et `bp` est de hauteur 1, puisque `sp` désigne l'entier du sommet de pile. Pour que la pile soit vide, il faut que `sp` désigne l'entier précédent celui désigné par `bp`. Il faut donc que `sp=bp-2`. Les quatre premières instructions de ce qui est appelé dans l'exemple "fonction principale" initialisent donc `bp` et `sp` pour qu'ils désignent une pile vide. Les deux instructions suivantes permettent d'appeler la fonction dont la première instruction est labelisée `saisie`. Cette fonction demande à l'utilisateur de rentrer un entier naturel au clavier. Par convention, au retour de la fonction, cet entier est placé dans le registre `ax`. Les deux instructions suivantes permettent de placer le contenu de `ax` (donc, la valeur rentrée au clavier par l'utilisateur), dans l'emplacement mémoire labelisé `valeurinit`. On appelle ensuite trois fonctions d'affichage, qui réalisent respectivement l'affichage de

— "Factorielle "
— la valeur rentrée par l'utilisateur
— " vaut "

Puis on appelle la fonction `factorielle`. Par convention, on place son argument (contenu dans le registre `ax`) dans la pile avant de l'appeler. Au retour de la fonction `factorielle`, le résultat de son calcul est par convention dans le registre `ax`. Comme on a placé une valeur dans la pile avant d'appeler la fonction `factorielle`, on prend soin de retirer cette valeur de la pile après l'appel de

la fonction, car cette dernière ne sert plus, et il convient de ne pas laisser dans la pile de valeurs inutiles. On stocke ensuite la valeur de retour dans l'emplacement mémoire labelisé par `resultat`, et on l'affiche. On affiche ensuite un caractère de retour à la ligne, et le programme est terminé.

La fonction de saisie étant suffisamment commentée, nous n'insistons pas. Nous passons à l'explication de la fonction `factorielle`. Cette fonction prend un argument entier qu'on lui fourni par l'intermédiaire de la pile, et calcule récursivement sa factorielle. Avant de terminer, elle met le résultat dans le registre `ax`. Les quatre premières instructions de la fonction servent à récupérer son argument. Ce dernier a été mis dans la pile juste avant qu'on appelle la fonction par un `call`. Le `call` place l'adresse de l'instruction à exécuter immédiatement après la fonction `factorielle` dans la pile (on appellera cette adresse *adresse de retour*), donc au dessus de l'argument de la fonction. Cette adresse d'instruction se trouvant au sommet de la pile est donc désignée par `sp`, et l'argument se trouve à l'adresse `sp-2`, 2 étant la taille d'un mot machine exprimé en cellules. Il existe deux méthodes pour récupérer l'argument : ou bien on dépile l'adresse de retour, on dépile l'argument, et on remet l'adresse de retour sur la pile, ou bien va lire directement dans la pile, en dessous de son sommet. C'est cette seconde méthode qui a été choisie. Une fois l'adresse de l'argument calculée, on récupère sa valeur, qu'on place dans le registre `ax`. On passe maintenant aux instructions suivantes, qui regardent si on est dans le cas de base (argument vaut zéro) ou pas. L'instruction `xor bx,bx` met 0 dans `bx` (faites-le à la main si vous n'en êtes pas convaincu). Ensuite on compare l'argument à `bx` (donc à 0), et si le résultat de la comparaison est positif (égalité) on saute à l'instruction labelisée `casdebase` : on met $0! = 1!$ dans `ax`, et on retourne à la fonction appelante. Si on est pas dans le cas de base, on est dans le cas général. On retire 1 à l'argument (toujours contenu dans `ax`). On note que la valeur initiale de l'argument se trouve encore dans la pile, sous l'adresse de retour : on pourra l'y retrouver si nécessaire). Une fois `ax` diminué de 1, on va faire un appel récursif à `factorielle`. On place son argument dans la pile, puis on l'appelle. Au retour, la valeur que cet appel aura calculé se trouvera dans `ax`. Comme on a placé une valeur dans la pile avant d'appeler récursivement la fonction `factorielle` et qu'on a plus besoin de cette valeur, on la retire de la pile pour laisser cette dernière dans un état propre. Il faut maintenant faire le produit de la valeur de retour de l'appel récursif (qui se trouve dans `ax`) et de l'argument de la fonction (qui lui se trouve, rappelons-le encore une fois, dans la pile sous l'adresse de retour). Comme en début de fonction, on calcule l'adresse du mot machine se situant immédiatement sous le sommet de la pile, et on récupère ce mot machine, en stockant sa valeur dans `bx`. On fait le produit de `bx` et `ax`, l'instruction `mul` met son résultat dans son premier argument, ici `ax`. Le calcul est terminé, la fonction se termine.

Voici le contenu du fichier `a.out` en sortie du programme d'assemblage correspondant au programme précédent écrit dans l'assembleur de `SIPRO`. Ce contenu a été obtenu en utilisant la commande unix `od -x a.out`. La première colonne est un numéro donné en octal. Ensuite viennent une suite de 8 mots de 16 bits, affichés en hexadécimal. Attention, le premier octet du mot est affiché en seconde position. Par exemple, la première instruction du programme est `const ax,debut`. L'instruction `const` a pour opcode `35` (base 16). Ensuite vient un octet identifiant le registre opérande de `const` : ici `ax`, l'octet vaut 4. Les deux premiers octets du fichier contenant le programme compilé sont donc dans l'ordre `35` et `4`. La commande `od` les a affichés dans l'ordre inverse. Bien entendu il en est de même pour les autres mots.

```
0000000 0435 c700 0460 6552 746e 6572 207a 6f76
0000020 7274 2065 6f6e 626d 6572 003a 6146 7463
0000040 726f 6569 6c6c 2065 0000 7620 7561 2074
0000060 0000 000a 7245 6572 7275 2120 0000 0000
0000100 0000 0240 0340 0540 0640 0740 0435 0600
0000120 046c 0535 bf00 0435 3e00 046e 0563 0530
0000140 3104 0504 0741 0641 0541 0341 0241 3066
0000160 0206 0535 0200 0621 3105 0604 0517 3505
0000200 0007 b400 0450 6105 3507 0005 0100 0421
0000220 4005 3504 0007 6f00 0765 0741 0730 3502
0000240 0005 0200 0721 3105 0705 0422 3505 0007
0000260 be00 0760 0435 0100 0735 be00 0760 3566
```

```
0000300 0004 3400 046c 35ff 0003 1401 0235 1401
0000320 0435 0200 0221 3504 0007 4200 0765 0535
0000340 3e00 0432 3505 0005 1c00 056c 0535 3e00
0000360 056b 0535 2a00 056c 0440 0535 6f00 0565
0000400 0541 0535 4000 0432 6b05 3505 0005 3200
0000420 056c 00ff 0000
```

La même commande, avec l'option `-c`, permet d'interpréter chaque octet comme étant un caractère, et donne :

```
0000000   5 004  \0   Ç   ‘ 004   R   e   n   t   r   e   z       v   o
0000020   t   r   e       n   o   m   b   r   e   :  \0   F   a   c   t
0000040   o   r   i   e   l   l   e      \0  \0       v   a   u   t
0000060  \0  \0  \n  \0   E   r   r   e   u   r       !  \0  \0  \0  \0
0000100  \0  \0   @ 002   @ 003   @ 005   @ 006   @  \a   5 004  \0 006
0000120   l 004   5 005  \0   ¿   5 004  \0   >   n 004   c 005   0 005
0000140 004   1 004 005   A  \a   A 006   A 005   A 003   A 002   f   0
0000160 006 002   5 005  \0 002   ! 006 005   1 004 006 027 005 005   5
0000200  \a  \0  \0   ´   P 004 005   a  \a   5 005  \0  \0 001   ! 004
0000220 005   @ 004   5  \a  \0  \0   o   e  \a   A  \a   0  \a 002   5
0000240 005  \0  \0 002   !  \a 005   1 005  \a   " 004 005   5  \a  \0
0000260  \0   ¾   ‘  \a   5 004  \0 001   5  \a  \0   ¾   ‘  \a   f   5
0000300 004  \0  \0   4   l 004   ÿ   5 003  \0 001 024   5 002 001 024
0000320   5 004  \0 002   ! 002 004   5  \a  \0  \0   B   e  \a   5 005
0000340  \0   >   2 004 005   5 005  \0  \0 034   l 005   5 005  \0   >
0000360   k 005   5 005  \0   *   l 005   @ 004   5 005  \0   o   e 005
0000400   A 005   5 005  \0   @   2 004 005   k 005   5 005  \0  \0   2
0000420   l 005   ÿ  \0  \0  \0
```

On voit nettement apparaître les chaînes de caractères constantes utilisées dans le programme.

C'est ce fichier `a.out` (ou d'un autre nom) que l'émulateur lira et interprétera comme un programme qu'il exécutera.