université de BORDEAUX

Architecture des ordinateurs : Programmation des processeurs avec l'environnement « *y86* »

(4TIN408U)

F. Pellegrini
Université de Bordeaux

y86 (1)

- Environnement pédagogique d'apprentissage :
  - De la programmation en langage machine
  - De l'impact de l'architecture des processeurs sur l'efficacité d'un programme
- Créé par R. E. Bryant et D. R. O'Hallaron dans le cadre de leur livre CS:APP
  - Librement téléchargeable (mais pas libre !)

http://csapp.cs.cmu.edu/

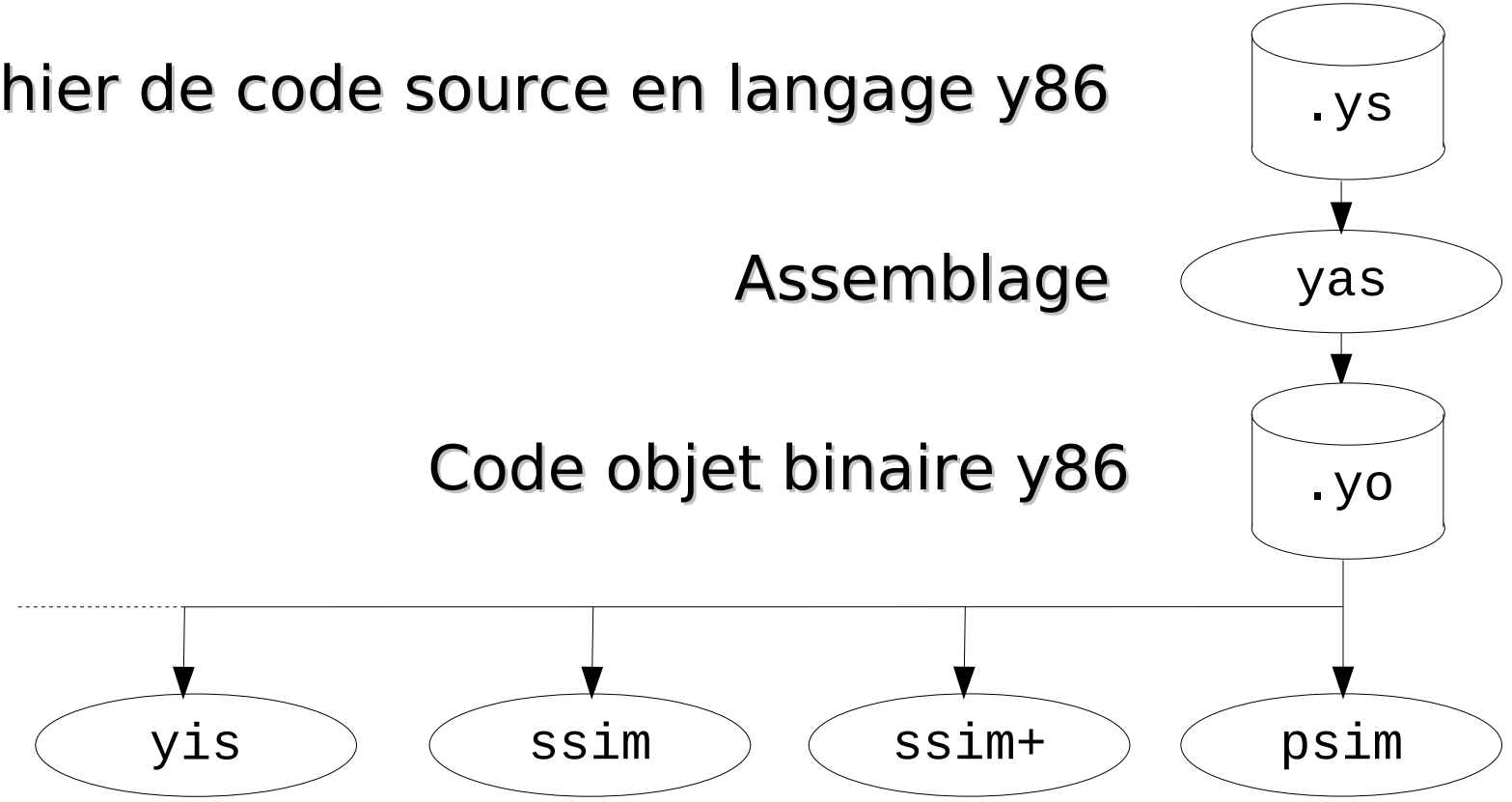
université de BORDEAUX

y86 (2)

- Architecture inspirée des processeurs Intel « x86 » (architecture IA32)
  - Jeu d’instructions simplifié
  - Syntaxe quelque peu modifiée
- Architecture évolutive

Outils disponibles (1)

- Chaîne logicielle « historique » :



université de BORDEAUX

Outils disponibles (2)

- De nombreux outils web ont également été créés par des tiers
- Outil web développé à l'université de Bordeaux
  - Permet la compilation de code source, l'exécution de code objet et aussi la modification de l'architecture du processeur

Jeu d'instructions y86

- Le jeu d'instructions du processeur y86 permet d'effectuer :
  - Des copies de données entre mémoire et registres
  - Des opérations arithmétiques et logiques
  - Des branchements
    - Exécution conditionnelle ou répétitive
  - Des appels et retours de sous-programmes
  - Des manipulations de pile

université de BORDEAUX

Adressage mémoire (1)

- Les mémoires informatiques sont organisées comme un ensemble de cellules pouvant chacune stocker une valeur numérique
- Toutes les cellules d'une mémoire contiennent le même nombre de bits
  - Par exemple, les « *octets* » sont des cellules à 8 bits
  - Le terme anglais « *byte* » est plus généraliste
  - Une cellule de n bits peut stocker 2^n valeurs numériques différentes

Adressage mémoire (2)

- La cellule est la plus petite unité mémoire pouvant être adressée
  - Chaque cellule est identifiée par un numéro unique, appelé adresse, auquel les programmes peuvent se référer
- Afin d'être plus efficaces, les unités de traitement ne manipulent plus des octets individuels mais des mots de plusieurs octets
  - 4 octets par mot pour une architecture 32 bits
  - 8 octets par mot pour une architecture 64 bits

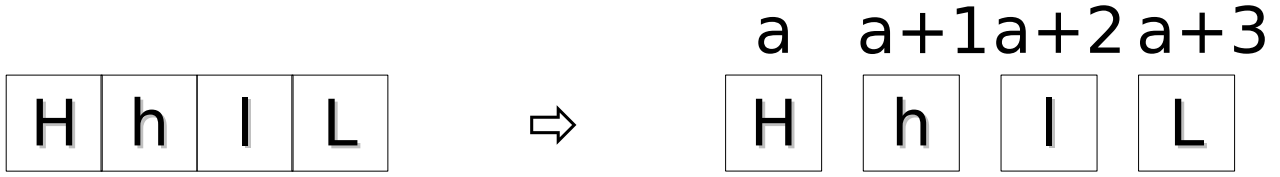
université de BORDEAUX

Adressage mémoire (3)

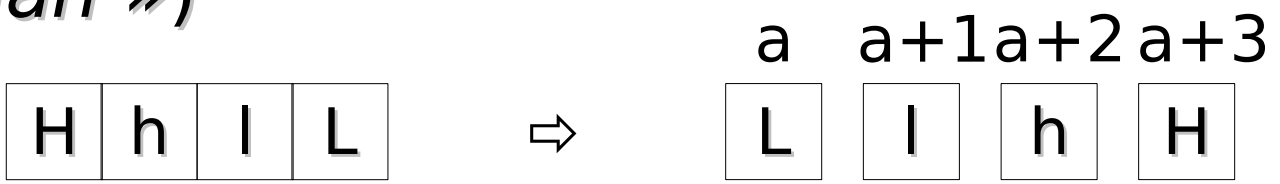
- Les ordinateurs, basés sur le système binaire, représentent également les adresses sous forme binaire
- Une adresse sur m bits peut adresser 2^m cellules distinctes, indépendamment du nombre de bits n contenus dans chaque cellule
  - Capacité totale de la mémoire : 2^{m+n} bits

Adressage mémoire (4)

- Pour stocker un mot machine en mémoire, il faut savoir dans quel ordre stocker les octets du mot dans les cases consécutives de la mémoire :
  - Octet de poids fort en premier (« *big endian* »)



  - Octet de poids faible en premier (« *little endian* »)



université de BORDEAUX

Adressage mémoire y86

- L'architecture y86 considère la mémoire comme un espace linéaire et continu d'octets
  - Cet espace commence à partir de l'adresse 0 et s'arrête à la plus grande adresse non signée représentable sur un mot machine
- Les mots machine, sur 32 bits, y sont stockés avec leur poids faible en premier
  - Stockage « *little endian* », ou « *petit-boutiste* »
- Par convention, les programmes débutent leur exécution à l'adresse 0

Registres

- Les registres sont des circuits du processeur capables de mémoriser des mots machine
- Ils sont localisés au plus près de l'UAL, afin que les calculs puissent être effectués le plus rapidement possible
- Les registres sont identifiés par un nom et/ou un numéro
- L'ensemble des registres et de la mémoire représente l'état du processeur

Registres y86 (1)

- L’architecture y86 possède 8 registres directement manipulables par l’utilisateur
  - 7 registres banalisés : eax, ebx, ecx, edx, esi, edi, ainsi que ebp
  - 1 registre spécial : esp
- Ces registres ont une taille de 32 bits (4 octets)

Registres y86 (2)

- Deux autres registres ne sont qu'indirectement accessibles et modifiables :
  - Le compteur ordinal (« *Program Counter* »)
    - Stocke l'adresse de l'instruction courante
  - PSW (« *Program Status Word* »)
    - Stocke les codes de condition : drapeaux à un bit positionnés par les instructions arithmétiques et logiques :
      - Z : à 1 si le résultat est nul (tous bits à zéro)
      - O (« *overflow* ») : à 1 si débordement arithmétique
      - S : à 1 si le résultat est strictement négatif

Registres y86 (3)

- Les 8 registres généraux sont identifiés dans le code machine par un numéro sur 3 bits
- Ces numéros sont intégrés au code binaire des instructions manipulant les registres
  - Représentation effective sur 4 bits

eax	000	edx	010	esp	100	esi	110
ecx	001	ebx	011	ebp	101	edi	111
rien	1000						

  - « addl rA,rB » ⇨

6	0	rA	rB

  - « addl %eax,%ebp » ⇨

6	0	0	5

Instructions y86

- Les instructions y86 occupent de 1 à 6 octets en mémoire
  - Leur taille dépend de leurs opérandes
  - La taille de l'instruction peut être déterminée en fonction des valeurs du premier octet de l'instruction
- Chacune d'elles modifie l'état du processeur
- Elles sont bien plus simples à décoder que les instructions IA32 originales

université de BORDEAUX

Langage d'assemblage (1)

- Un langage d'assemblage offre une représentation symbolique d'un langage machine
  - Utilisation de noms symboliques pour représenter les instructions et leurs adresses
- L'assembleur convertit cette représentation symbolique en langage machine

Langage d'assemblage (2)

- Avantages du langage d'assemblage par rapport au langage machine
  - Facilité de lecture et de codage
    - Les noms symboliques des instructions (« codes mnémoniques ») sont plus simples à utiliser que les valeurs binaires dans lesquelles elles seront converties
  - Facilité de modification
    - Pas besoin de recalculer à la main les adresses des destinations de branchements ou des données lorsqu'on modifie le programme

Traductions successives

- Code C :

```
int c = a + b;
```

- Code assembleur :

```
        mrmovl   4(%esp),%eax
        mrmovl   8(%esp),%ebx
        addl     %ebx,%eax
```

  - Correspond à :

```
register int eax = a;
register int ebx = b;
```

  - Opérandes :
    - a : mémoire, à l'adresse (esp + 4)
    - b : mémoire, à l'adresse (esp + 8)
    - c : registre eax (variable temporaire ?)
  - Résultat dans eax

```
eax += ebx;
```

- Code binaire :

```
50 04 04 00 00 00
50 34 08 00 00 00
60 30
```

université de BORDEAUX

Instructions des langages d'assemblage

- Les instructions des langages d'assemblage sont habituellement constituées de trois champs :
  - Un champ d'étiquette, optionnel
  - Un champ d'instruction
  - Un champ de commentaires, optionnel

Champ étiquette

- Champ optionnel
- Associe un nom symbolique à l'adresse à laquelle il se trouve
  - Destination de branchement ou d'appel de sous-routine pour les étiquettes situées en zone de code
  - Adresses de variables ou constantes mémoire pour les étiquettes situées en zone de données

université de BORDEAUX

Champ instruction

- Contient soit une instruction soit une directive destinée à l'assembleur lui-même
- Une instruction est constituée :
  - Du code mnémonique du type de l'opération
  - De représentations symboliques des opérandes
    - Noms symboliques de registres
    - Représentations de constantes numériques sous différents formats :
      - Décimal, binaire, octal, hexadécimal
    - Expressions parenthésées ou à base de crochets permettant d'exprimer les différents modes d'adressage

Directives

- Influent sur le comportement de l'assembleur
- Ont de multiples fonctions :
  - Placement et alignement en mémoire du code et des données
  - Réservation et initialisation d'espace mémoire
  - ...

université de BORDEAUX

Exemple : programme somme

- Calcul de « c = a + b »

```
0x000:                  |            .pos 0
0x000: 500864000000     | prog:      mrmovl a,%eax
0x006: 503868000000     |            mrmovl b,%ebx
0x00c: 6030             |            addl   %ebx,%eax
0x00e: 40086c000000     |            rmmovl %eax,c
0x014: 10               |            halt
                        |
                        | # Donnees
0x064:                  |            .pos 100
0x064: 02000000         | a:         .long 2
0x068: 03000000         | b:         .long 3
0x06c: 00000000         | c:         .long 0
```

Directives y86 (1)

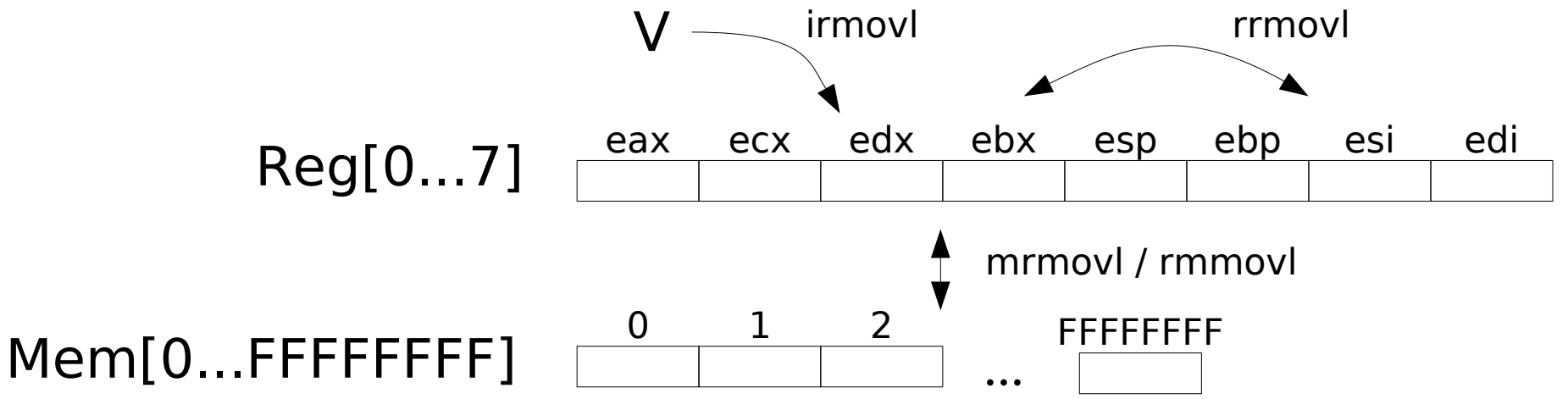
- Placement à une adresse donnée : « .pos *x* »
  - Démarre la génération de code binaire à l'adresse fournie
- Écriture d'un mot de données à la position courante : « .long x »
  - Le mot est écrit en codage « *little endian* » : octet de poids le plus faible en premier
  - Pour que les données soient bien lisibles par le simulateur, il faut les aligner sur un multiple de 4 avec « .pos » ou « .align »

Directives y86 (2)

- Alignement de la génération sur un multiple d'une puissance de 2 : « .align x »
  - Facilite les calculs d'adresses
  - Optimise les accès mémoire sur les architectures modernes
  - En pratique, on n'utilisera que la valeur 4
    - Alignement mot, pour affichage par le simulateur

Copie de données en y86 (1)

- Copie de données autorisés :
  - Valeur constante immédiate vers un registre
  - Valeur d'un registre vers un autre registre
  - Valeur d'un registre vers un mot mémoire
  - Valeur d'un mot mémoire vers un registre



université de BORDEAUX

Copie de données en y86 (2)

- Pas de placement d'une valeur immédiate en mémoire
  - Nécessité de passer par un registre intermédiaire
- Pas de copie de mémoire à mémoire
  - Il est impossible de faire deux accès mémoire au cours de la même instruction
    - Complexifierait la circuiterie
    - Augmenterait la taille des instructions
  - Nécessité de passer par un registre intermédiaire

université de BORDEAUX

Copie de données en y86 (3)

- La copie de données se fait au moyen de l'instruction : « *sd*movl src,dst »
  - Manipule uniquement des mots de 32 bits (« long »)
- Les deux premières lettres codent le type de la source et de la destination :
  - « i » : Donnée immédiate (uniquement source)
  - « r » : Registre
  - « m » : Mémoire
- Pas de « im », « mm » et « *i »

université de BORDEAUX

Copie de données en y86 (4)

- Les « modes d'adressage » définissent la façon dont les données sont rendues accessibles aux instructions qui les manipulent
- Adressage registre : on copie la valeur d'un registre source vers un registre destination
  - « rrmovl rA,rB »

    Reg[rB] = Reg[rA]

```
eax = ebx;
```

```
        rrmovl   %ebx,%eax
```

Copie de données en y86 (5)

- Adressage immédiat : la valeur à placer dans le registre est contenue dans l'instruction elle-même
  - « irmovl V,rB »

    Reg[rB] = V

```
eax = 42;
```

```
irmovl   42,%eax
```

Copie de données en y86 (6)

- Adressage direct : on fournit la valeur de l'adresse mémoire à laquelle on veut accéder
  - « mrmovl D,rA »

    Reg[rA] = Mem[D]
  - Permet de manipuler des variables globales
    - L'adresse est contenue dans l'instruction

```
int a = 12;
eax = a;
```

```
            mrmovl   a,%eax

            .pos 0x100
a:          .long 12
```

Copie de données en y86 (7)

- Adressage indirect par registre : on fournit le numéro d'un registre pointeur qui contient l'adresse mémoire à accéder
  - « mrmovl (rB),rA »

    Reg[rA] = Mem[Reg[rB]]
  - Ne pas confondre « irmovl a,%esi » et « mrmovl a,%esi » !

```
int a = 12;
esi = &a;
eax = *esi;
```

```
            irmovl   a,%esi
            mrmovl   (%esi),%eax

            .pos 0x100
a:          .long 12
```

Copie de données en y86 (8)

- Adressage indexé : l'adresse mémoire est calculée par ajout d'une valeur au contenu d'un registre pointeur
  - « mrmovl D(rB),rA »
    - Reg[rA] = Mem[Reg[rB] + D]
  - Permet d'accéder aux champs d'une structure pointée, aux éléments d'un tableau, etc.

```
int t[2] = { 12, 34 };
esi = t;
eax = esi[1];
```

```
            irmovl   t,%esi
            mrmovl   4(%esi),%eax

t:          .long 12
            .long 34
```

Copie de données en y86 (9)

- Codage des instructions de copie :
  - « rrmovl rA,rB »

2	0	rA	rB

    Reg[rB] = Reg[rA]
  - « irmovl V,rB »

3	0	8	rB	V

    Reg[rB] = V
  - « rmmovl rA,D(rB) »

4	0	rA	rB	D

    Mem[Reg[rB] + D] = Reg[rA]
  - « mrmovl D(rB),rA »

5	0	rA	rB	D

    Reg[rA] = Mem[Reg[rB] + D]
  - On peut avoir D = 0 ou rB = 8 si pas nécessaire

université de BORDEAUX

Opérations arithmétiques en y86 (1)

- Les instructions arithmétiques du jeu d'instructions y86 original n'opèrent qu'entre deux registres
  - Le deuxième registre opérande reçoit le résultat de l'opération
- Codage des instructions arithmétiques :
  - « *op*l rA,rB »

6	op	rA	rB

Reg[rB] op= Reg[rA]

add	0	and	2
sub	1	xor	3

Opérations arithmétiques en y86 (2)

- Sur le simulateur que vous allez utiliser, le jeu d'instructions a été étendu pour permettre les opérations avec des valeurs immédiates
  - « i*oper*l V,rB »

    Reg[rB] op= V

- Évite de devoir charger une valeur dans un deuxième registre pour effectuer l'opération
  - Gain en registres
  - Gain en nombre d'instructions et en place

Opérations arithmétiques en y86 (3)

- Les opérations arithmétiques et logiques modifient l'état d'un registre spécifique appelé « mot d'état programme »
- Les bits concernés sont :
  - Z : à 1 si le résultat est nul (tous bits à zéro)
  - O (« *overflow* ») : à 1 si débordement arithmétique
  - S : à 1 si le résultat est strictement négatif
- Les valeurs de ces bits peuvent être exploitées pour effectuer des branchements conditionnels

Astuces de programmation

- « andl rA,rA » permet de positionner les bits S et Z en fonction de la valeur de rA sans modifier celle-ci (O vaudra toujours 0)
  - x and x = x
  - Économise 4 octets par rapport à « isubl 0,rA »
- « xorl rA,rA » met à zéro la valeur de rA, quelle que soit sa valeur antérieure
  - x xor x = 0
  - Économise 4 octets par rapport à « irmovl 0,rA »

Branchements en y86 (1)

- Les instructions de branchement (« *jump* ») permettent de modifier la valeur du compteur ordinal
- Le branchement peut n'être effectué que si certaines conditions du mot d'état programme sont vérifiées : branchements conditionnels
- Les branchements conditionnels permettent de mettre en œuvre :
  - Les tests : if ... then ... else ...
  - Les boucles : while ..., for ..., do ... while ...

Branchements en y86 (2)

- On évalue les codes de condition en comparant par soustraction deux valeurs contenues dans les registres rA et rB
  - jle : (rA ⩽ rB) : (S^O = 1) ou (Z=1)
  - jl : (rA < rB) : (S^O = 1)
  - je : (rA = rB) : (Z=1)
  - jne : (rA ≠ rB) : (Z=0)
  - jge : (rA ⩾ rB) : (S^O = 0) ou (Z=1)
  - jg : (rA > rB) : (S^O = 0)

Branchements en y86 (3)

- Réalisation de tests :
  - if... then... :

```
if (eax != 3) {
  ...1
}
...2
```

```
            rrmovl   %eax,%ecx
            isubl    3,%ecx
            je       apres
            ...1
apres:      ...2
```

- Là où le test en C exprime quand exécuter le bloc conditionnel, le test en y86 exprime quand le contourner
  - On doit prendre l'inverse de la proposition logique écrite en langage C

Branchements en y86 (4)

- Cas particulier : comparaison avec zéro
  - if... then... :

```
if (eax != 0) {
  ...1
}
...2
```

```
            isubl    0,%eax
            je       apres
            ...1
apres:      ...2
```

```
            andl     %eax,%eax
            je       apres
            ...1
apres:      ...2
```

- Pas besoin d'un registre de travail
  - « isubl 0,%eax » ne modifie pas eax
- « andl rA,rA » économise 4 octets

Branchements en y86 (5)

- Réalisation de tests :
  - If... then... else... :

```
if (eax != 3) {
  ...1
}
else {
  ...2
}
...3
```

```
            rrmovl   %eax,%ecx
            isubl    3,%ecx
            je       else
            ...1
            jmp apres
else:       ...2
apres:      ...3
```

- Branchement inconditionnel
  - jmp : Condition toujours vraie

Branchements en y86 (6)

- Réalisation de boucles :

```
while (eax -- > 0) {
  ...1
}
...2
```

```
boucle:   isubl    1,%eax
          jl       fin
          ...1
          jmp      boucle
fin:      ...2
```

  - Transformation du test (eax -- > 0) en (-- eax >= 0)
  - La décrémentation met à jour les drapeaux de condition Z, S et O sans qu'on ait besoin d'une instruction supplémentaire ensuite pour comparer le résultat à 0

Branchements en y86 (7)

- Codage des instructions de branchement :
  - jmp D

7	0	D

    PC = D
  - jop D

7	op	D

    Si (condition vérifiée) PC = D
- Les valeurs des codes de conditions sont :

mp	0	l	2	ne	4	g	6
le	1	e	3	ge	5		

Autres instructions y86

- « halt » : arrête l'exécution du programme
- « nop » : instruction sur un octet qui ne fait rien
  - Octet de valeur 0x00
  - Insérée autant que nécessaire par la directive .align
  - Permet de réserver de la place libre dans un programme pour y placer dynamiquement des instructions à l'exécution
    - Programmes auto-modifiables

Manipulation de tableaux (1)

- Pour utiliser un tableau, il faut être capable de faire des accès mémoire aux adresses des différentes cases

```
for (i = 4; i < n; i ++)
  t[i] = t[i % 4];
```

- Les adresses des accès sont donc variables
  - Elles doivent être contenues dans des registres
  - Utilisation des modes d'adressage mémoire :
    - Indirect par registre
    - Indexé

Manipulation de tableaux (2)

- Lors d'un parcours de tableau, on a deux façons de considérer l'adresse de la case mémoire courante t[i] :
  - Comme la $i^{ème}$ case du tableau t

```
for (i = 0; i < 10; i ++)
  t[i] = i;
```

  - Comme la valeur courante d'un pointeur initialisé à l'adresse de début du tableau puis incrémenté

```
for (p = t, i = 0; i < 10; p ++, i ++)
  *p = i;
```

université de BORDEAUX

Manipulation de tableaux (3)

- Avec un pointeur, on utilise l'adressage indirect par registre pour accéder à la case courante
  - Un registre joue le rôle de variable pointeur
  - Le déplacement du pointeur se fait 4 octets par 4 octets

```
for (eax = 0, ecx = 10, ebx = t;
     -- ecx >= 0;
     eax ++, (byte *) ebx += 4)
  *ebx = eax;
```

```
            irmovl   t,%ebx
            irmovl   10,%ecx
            xorl     %eax,%eax
boucle:     isubl    1,%ecx
            jl       fin
            rmmovl   %eax,(%ebx)
            iaddl    4,%ebx
            iaddl    1,%eax
            jmp      boucle
fin:        halt
            .align 4
t:          .long 0
```

université de BORDEAUX

Manipulation de tableaux (4)

- Si l'adresse de début du tableau est constante, &t[i] peut se calculer comme : « t + (i * sizeof(int)) »
  - L'adresse constante de début du tableau sert de déplacement à un accès par adressage indexé
  - Évite une addition

```
for (eax = 0, ecx = 10, ebx = 0;
     -- ecx >= 0;
     eax ++, ebx += 4 /* octets */)
  t[ebx] = eax;
```

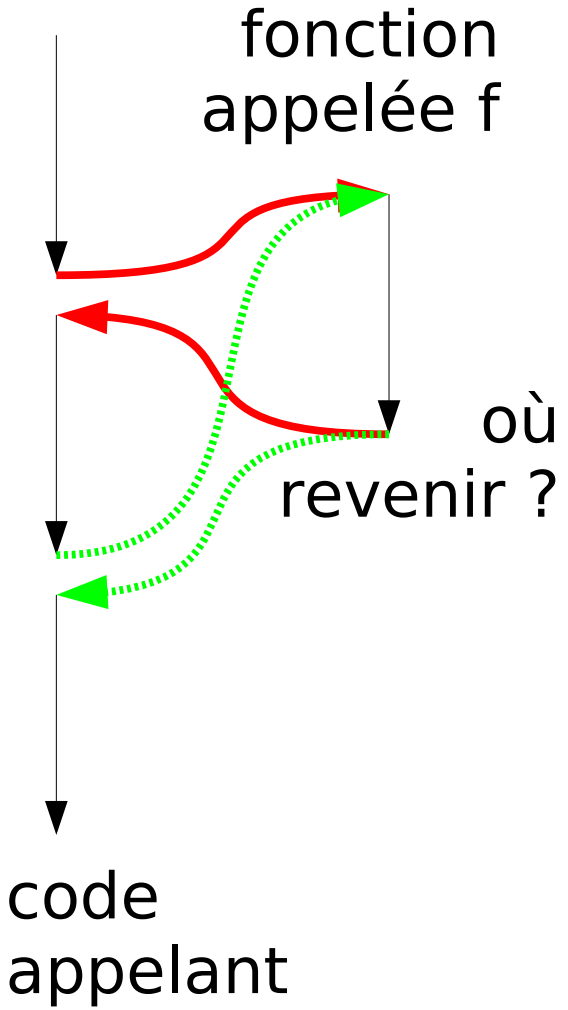
```
            irmovl   10,%ecx
            xorl     %eax,%eax
            xorl     %ebx,%ebx
boucle:     isubl    1,%ecx
            jl       fin
            rmmovl   %eax,t(%ebx)
            iaddl    4,%ebx
            iaddl    1,%eax
            jmp      boucle
fin:        halt
            .align 4
t:          .long 0
```

Appel de procédures (1)

- Lorsqu'on réalise plusieurs fois la même tâche à différents endroits d'un programme, il est dommage de dupliquer le code
  - Gaspillage de place en mémoire
  - Perte de temps et risque d'erreur accru en maintenance, à modifier plusieurs fois des fragments identiques
- Il faut « factoriser » le code considéré

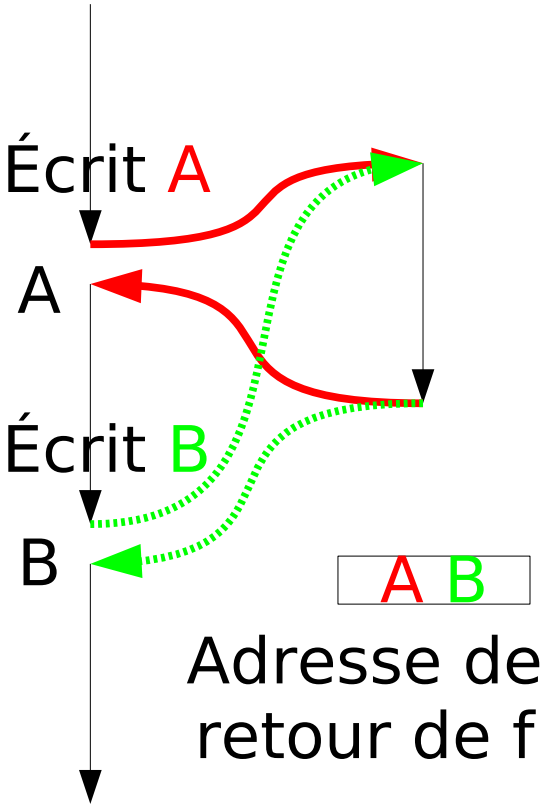
Appel de procédures (2)

- Se dérouter pour aller exécuter un fragment de code est simple
  - Instruction jmp
- Mais comment savoir à quel endroit du code revenir ?
  - L'« adresse de retour » ne peut être écrite « en dur » dans le code appelé
  - L'adresse de retour est variable
  - Comment et où la mémoriser ?



Appel de procédures (3)

- On peut imaginer que, pour chaque fonction appelable, il existe une zone mémoire où l'appelant mettra l'adresse où revenir à la fin de la fonction
- Ça ne marche pas pour les appels récursifs !
  - On écrase la valeur de retour quand on se ré-appelle
  - On boucle infiniment au retour...

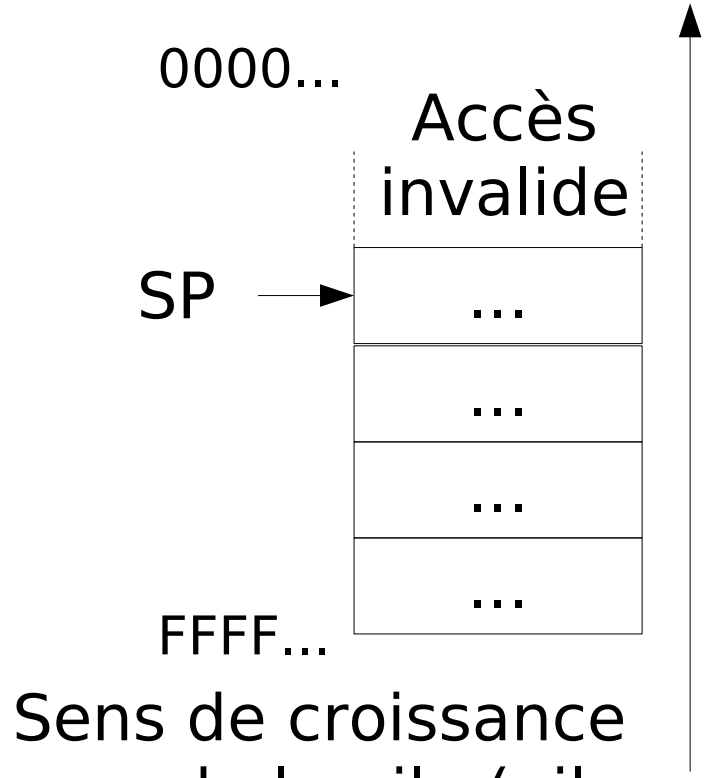


Appel de procédures (4)

- Il faut sauvegarder autant d'adresses de retour qu'il y a d'appels en cours
  - Les zones mémoire de stockage des adresses de retour sont donc multiples, et variables
  - Elles doivent être indexées par un registre
  - Registre spécial « esp » (« *stack pointer* ») mettant en œuvre une structure de pile
- Appel et retour de fonctions au moyen des instructions « call » et « ret »

Appel de procédures (5)

- La pile est une zone mémoire réservée à l'appel de fonctions
- Le pointeur de pile marque la position du « sommet » de la pile
  - Dernière case utilisée
  - Tout ce qui se trouve au delà du sommet de pile ne doit pas être accédé
    - C'est une erreur de le faire !



Appel de procédures (6)

- On ne peut manipuler la pile que si l'on a défini son emplacement au préalable
  - Réservation d'une zone mémoire dédiée
  - Doit être de taille suffisante pour éviter d'écraser les zones de code et de données
- Initialisation du pointeur de pile au début du programme
  - Première instruction du programme, pour être sûr

```
        .pos 0
        irmovl  pile,%esp
        ...
```

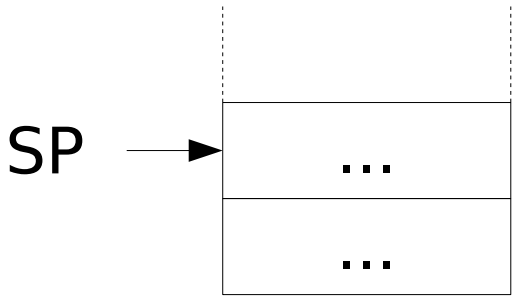
```
        .pos    0x200
pile:   .long   0
```

Appel de procédures (7)

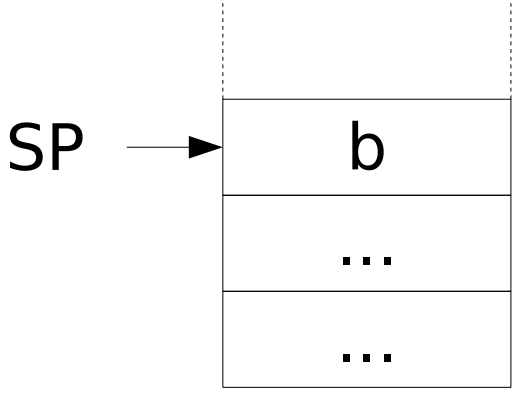
- Instruction « call *addr* » :
  - Empile l'adresse située après l'instruction
    - Le pointeur de pile est décrémenté
    - Le compteur ordinal est écrit à cette adresse
  - Place dans le compteur ordinal l'adresse d'appel
  - Effectue un jmp à cette adresse

PC = a

PC = func



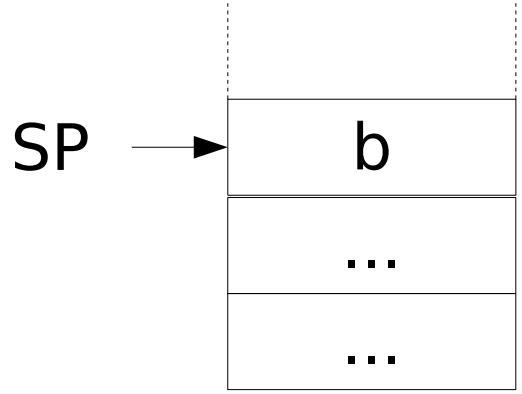

```
a:    call   func
b:    ...

func: ...
```



Appel de procédures (8)

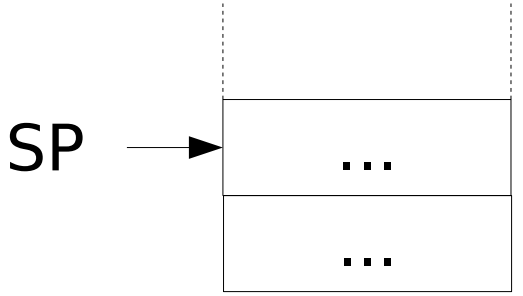
- Instruction « ret » :
  - Dépile l'adresse située en sommet de pile
    - Lit la valeur contenue à l'adresse du sommet de pile
    - Incrémente le pointeur de pile
  - Place dans le compteur ordinal l'adresse obtenue
    - Effectue un jmp à cette adresse

PC = c




```
a:    call   func
b:    ...

func: ...
      ...
c:    ret
```

PC = b



Appel de procédures (9)

- On peut passer des paramètres à la fonction appelée en positionnant les valeurs des registres utilisés par cette fonction

```
            .pos 0
            irmovl  pile,%esp
            mrmovl  a,%ebx
            mrmovl  b,%ecx
            call    mult
            rmmovl  %eax,c
            halt

mult:       xorl    %eax,%eax
multbcl:    andl    %ebx,%ebx
            je      multfin
            addl    %ecx,%eax
            isubl   1,%ebx
            jmp     multbcl
multfin:    ret
```

```
            .pos    0x100
a:          .long 3
b:          .long 5
c:          .long 0
```

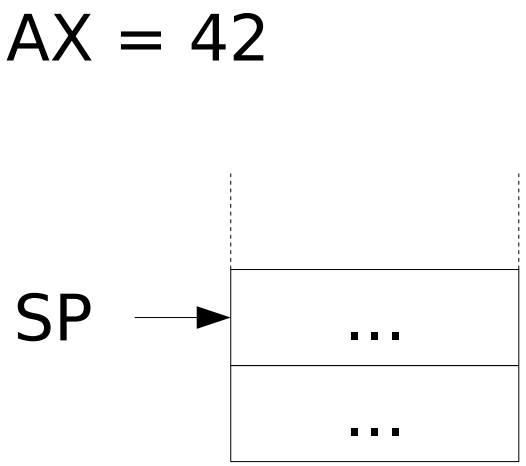
```
            .pos    0x200
pile:       .long   0
```

Sauvegarde de registres (1)

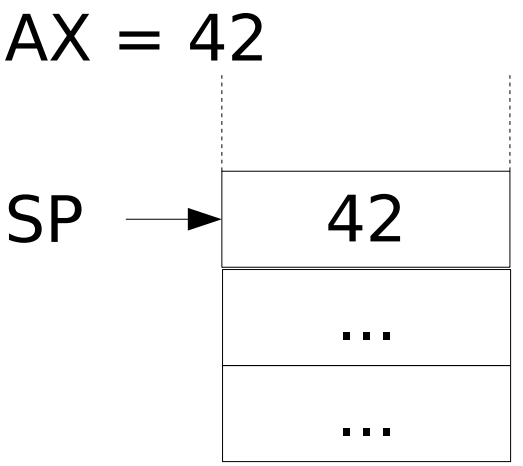
- Le nombre de registres étant limité, il est souvent nécessaire de stocker temporairement des valeurs en mémoire
- Pour que les fonctions utilisant ce principe puissent être appelées récursivement, ces zones de sauvegarde ne peuvent être situées à des adresses fixes en mémoire
  - Écrasement des valeurs stockées lors de l'appel précédent
- Utilisation de la pile pour résoudre le problème

Sauvegarde de registres (2)

- Instruction « pushl *reg* » :
  - Empile la valeur contenue dans le registre
    - Le pointeur de pile est décrémenté



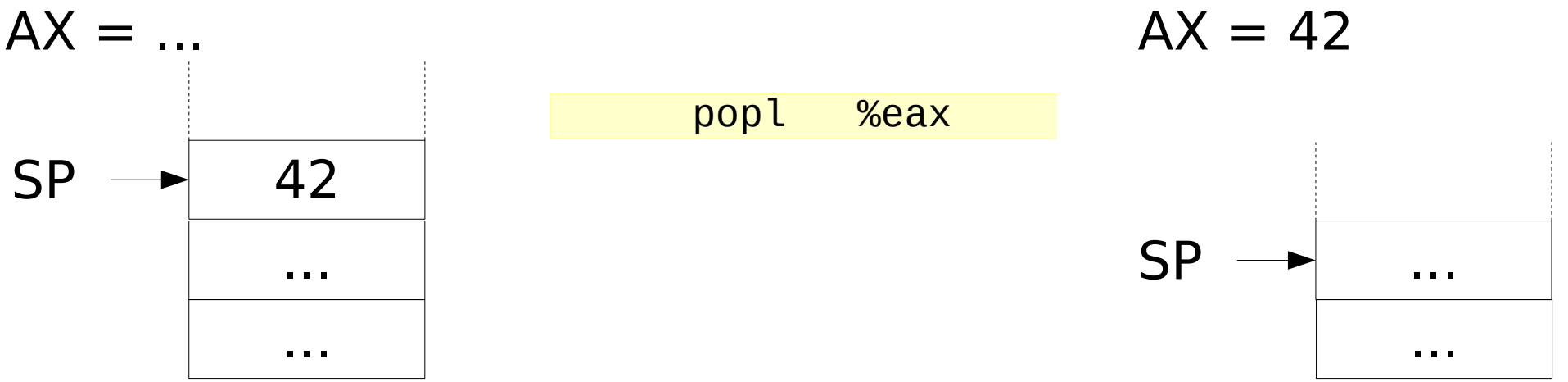

```
        irmovl 42,%eax
        pushl  %eax
```



université de BORDEAUX

Sauvegarde de registres (3)

- Instruction « popl *reg* » :
  - Dépile et place dans le registre la valeur située en sommet de pile
    - Lit la valeur contenue à l'adresse du sommet de pile
    - Incrémente le pointeur de pile



Sauvegarde de registres (4)

- On dépile habituellement dans l'ordre inverse de celui dans lequel on a empilé

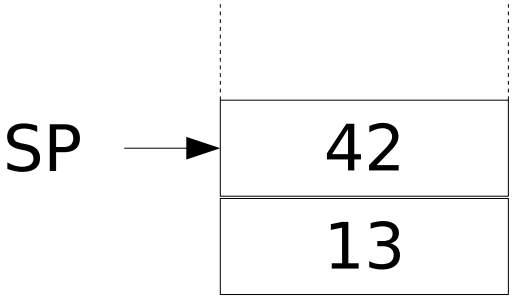
```
        pushl   %ebx
        pushl   %esi
        pushl   %edi
        ...
        popl    %edi
        popl    %esi
        popl    %ebx
```

- Inverser les dépilages permet de réaliser des échanges de valeurs entre registres (« swap »)

AX = 13

BX = 42

```
        pushl   %eax
        pushl   %ebx
        popl    %eax
        popl    %ebx
```



AX = 42

BX = 13

Sauvegarde de registres (5)

- Rien n'empêche une fonction appelée de modifier les valeurs de tous les registres, y compris ceux contenant ses paramètres
- La fonction appelante pourrait cependant vouloir conserver leurs valeurs, pour des calculs ultérieurs
- Alors : qui se charge de les sauvegarder ?
  - L'appelant, avant l'appel (« *caller save* ») ?
  - L'appelé, après l'appel (« *callee save* ») ?

Sauvegarde de registres (6)

- Si c'est l'appelant qui sauvegarde ce qu'il veut garder, il peut sauvegarder pour rien des registres que l'appelé ne modifiera pas
- Si c'est l'appelé qui sauvegarde ce qu'il modifie, il peut également sauvegarder des registres sans importance pour l'appelant
- Dans tous les cas, il faut s'organiser :
  - Si appelant et appelé sauvegardent les mêmes registres, on fait du travail pour rien
  - Si aucun ne les sauve, c'est bien plus gênant !

Sauvegarde de registres (7)

- Les registres servant traditionnellement au stockage des valeurs de retour ne peuvent pas être sauvegardés par l'appelé
  - L'appelé a justement pour mission de les modifier
  - C'est à l'appelant de les sauvegarder s'il le souhaite (« *caller save* »), pour les restaurer une fois qu'il aura traité les valeurs de retour s'il y en a
  - Ce sont : AX, CX, DX
  - Registres « de travail » librement utilisables par chaque fonction

Sauvegarde de registres (8)

- Les registres servant traditionnellement aux calculs d'adresse sont supposés ne pas être modifiés par l'appelé
  - C'est à l'appelé de les sauvegarder au besoin (« *callee save* »)
  - Ce sont : BX, SI, DI et éventuellement BP
- Le registre spécial SP fait l'objet d'un traitement spécifique (parfois BP aussi)
  - Nécessaires à la gestion des variables locales aux fonctions

Passage de paramètres (1)

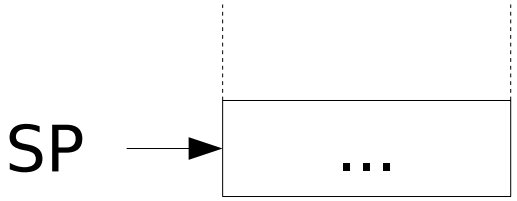
- Lorsqu'on appelle une fonction, on peut avoir à lui passer plus de paramètres qu'il n'existe de registres
- Obligation de passer les paramètres à travers une zone mémoire
  - Écrits par l'appelant, lus par l'appelé
- Pour garantir la ré-entrance, cette zone ne peut être située à une adresse fixe en mémoire
- Il faut passer les paramètres par la pile

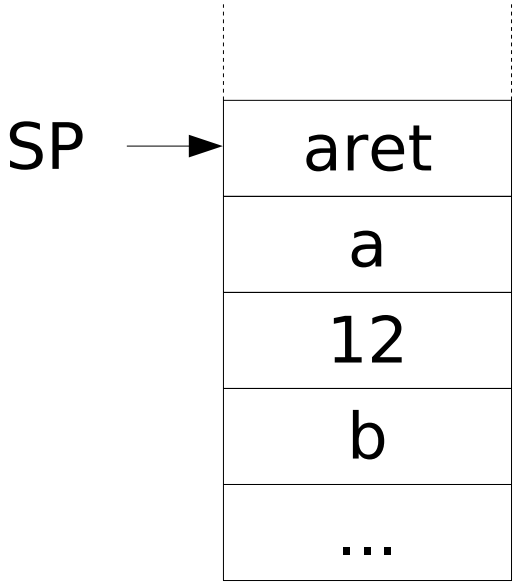
Passage de paramètres (2)

- Avant d'appeler la fonction, l'appelant empile les paramètres dans l'ordre inverse duquel ils sont nommés
  - Le dernier est empilé en premier, et ainsi de suite...

```
f (a, 12, b);
```

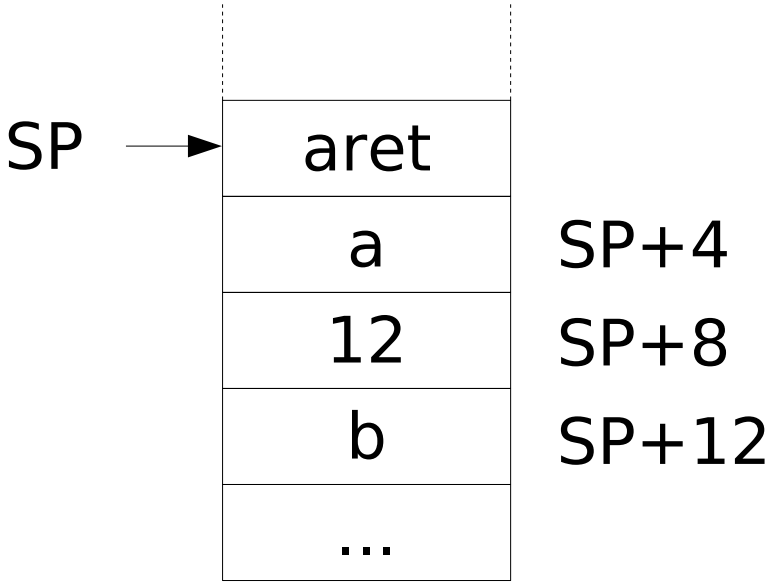
```
        mrmovl b,%eax
        pushl  %eax
        irmovl 12,%eax
        pushl  %eax
        mrmovl a,%eax
        pushl  %eax
        call   f
aret:   ...

f:      ...
```





Passage de paramètres (3)

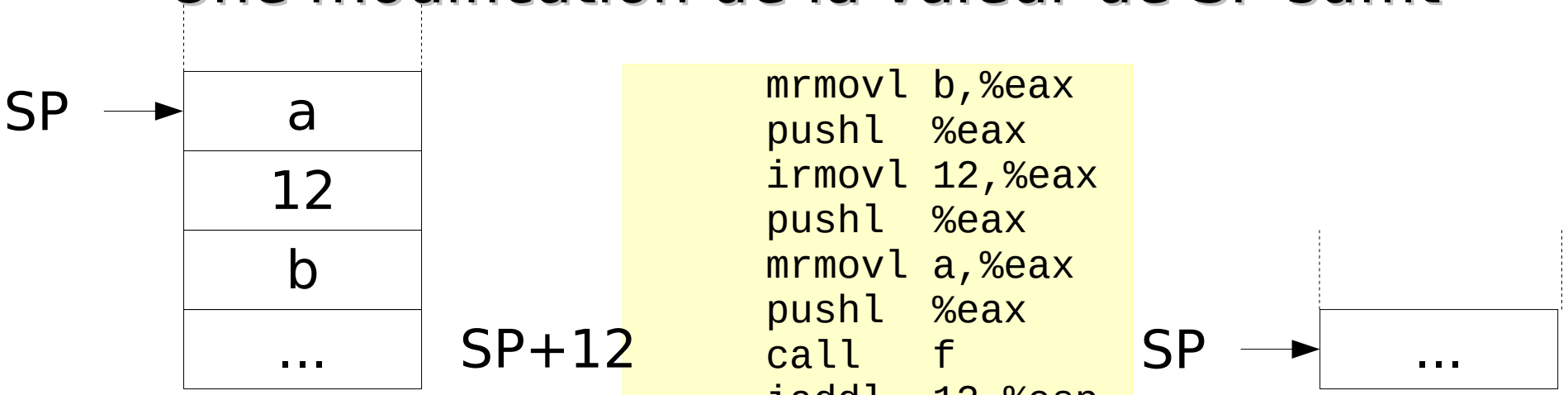
- La fonction appelée peut récupérer les paramètres en les lisant en mémoire par rapport à la position du sommet de pile
  - Elle peut les lire autant de fois que nécessaire
  - Elle peut les modifier (comme les paramètres C)




```
f:      mrmovl 4(%esp),%eax
        mrmovl 8(%esp),%edx
        mrmovl 12(%esp),%ecx
        ...
        ret
```

Passage de paramètres (4)

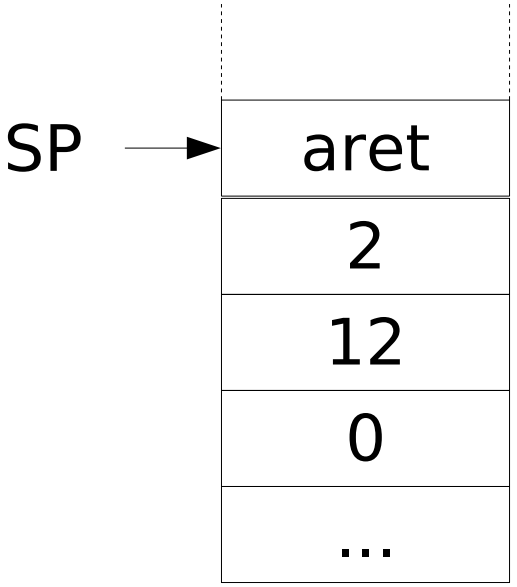
- Une fois revenu de la fonction, l'appelant doit vider la pile des paramètres empilés
  - Seul l'appelant sait combien il en a mis
- On n'a pas besoin de dépiler les valeurs, juste de remettre la pile en son ancien état
  - Une modification de la valeur de SP suffit



Passage de paramètres (5)

- Le passage des paramètres dans l'ordre inverse permet de mettre en œuvre des fonctions à nombre d'arguments variables
  - On sait toujours où se trouve le premier paramètre

```
somme (2, 12, 0);
somme (13, 4, 3, 0);
```




```
        irmovl 0,%eax
        pushl  %eax
        irmovl 12,%eax
        pushl  %eax
        irmovl 2,%eax
        pushl  %eax
        call   somme
        iaddl  12,%esp
```

```
somme:  rrmovl %esp,%edx
        xorl   %eax,%eax
boucle: iaddl  4,%edx
        mrmovl (%edx),%ecx
        andl   %ecx,%ecx
        je     fin
        addl   %ecx,%eax
        jmp    boucle
fin:    ret
```

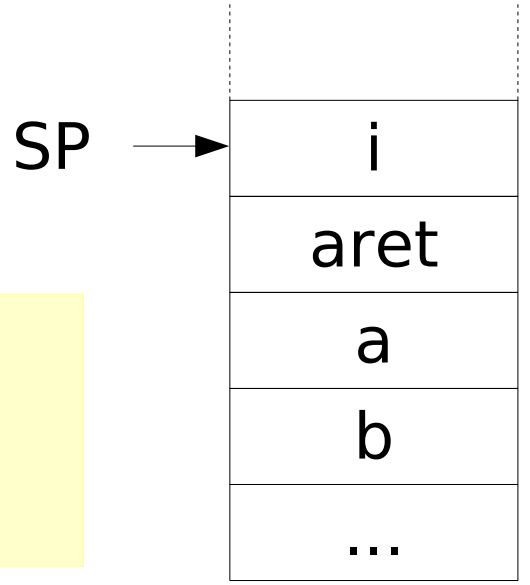
Variables locales (1)

- Toute fonction peut avoir besoin de plus de variables locales qu'il n'existe de registres
- Obligation de stocker ces variables dans une zone mémoire dédiée
- Pour garantir la ré-entrance, cette zone ne peut être située à une adresse fixe en mémoire
  - C'est encore et toujours le même argument !
- Il faut stocker ces variables dans la pile

Variables locales (2)

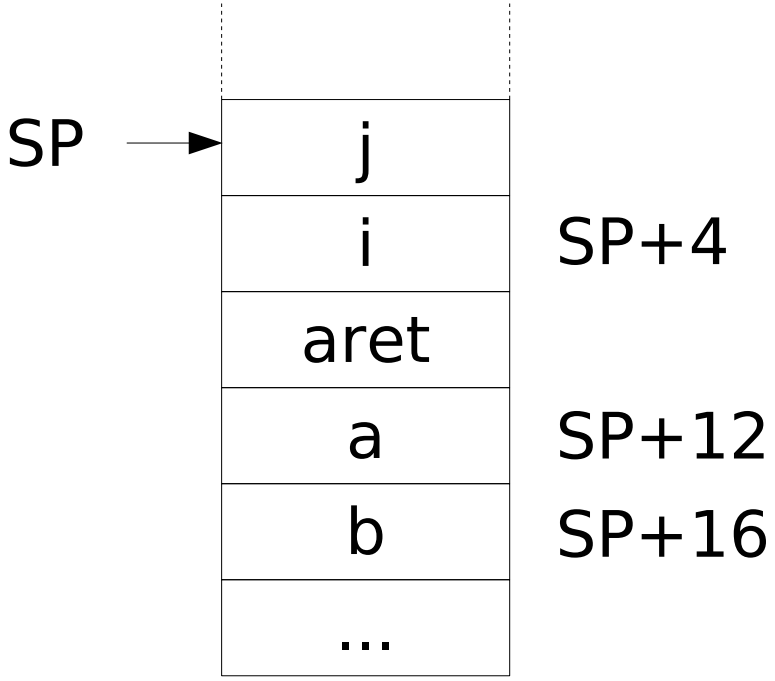
- C’est à la fonction appelée de gérer ses variables locales
- Elles doivent donc être situées dans la pile au-dessus des paramètres et de l’adresse de retour placés par la fonction appelante
- Attention : quand on déplace SP, on modifie l’accès aux paramètres !

```
int
f (int a, int b)
{
  int     i;
  ...
```



Variables locales (3)

- Mieux vaut réserver l'ensemble de l'espace nécessaire dès l'entrée dans la procédure




```
int
f (int a, int b)
{
  int     i, j;
  i = a + b;
  ...
```

```
f:      isubl  8,%esp
        mrmovl 12(%esp),%eax
        mrmovl 16(%esp),%edx
        addl   %eax,%edx
        rmmovl %edx,4(%esp)
        ...
        iaddl  8,%esp
        ret
```

Récapitulatif : usage de la pile (1)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelante :
  - Sauvegarde éventuelle des registres *caller save*
  - Empilage des paramètres, dans l'ordre inverse de celui dans lequel ils sont listés dans la procédure
    - Autorise les fonctions à nombre d'arguments variables : le premier paramètre est le plus proche de SP, les autres sont dessous !
  - Appel de la procédure
    - Sauvegarde automatique de l'adresse de retour dans la pile

université de BORDEAUX

Récapitulatif : usage de la pile (2)

- Lorsqu'on entre dans une fonction :
  - SP sert de référence au « contexte courant »
    - Lorsqu'on entre dans la fonction, le premier paramètre est accessible à l'adresse de SP plus un mot, soit (SP+4), le suivant à (SP+8), etc.
  - Des modifications ultérieures du pointeur de pile au sein de la fonction peuvent changer ces valeurs de déplacement
    - Toujours bien représenter la pile lorsqu'on code !

Récapitulatif : usage de la pile (3)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelée (début de procédure) :
  - Empilage éventuel des registres *callee save*
    - Modifie le décalage à partir duquel les paramètres et variables locales sont accessibles !
      - Les compilateurs savent faire ce travail sans se tromper
  - Soustraction éventuelle à SP de la taille des variables locales
    - Modifie aussi le décalage à partir duquel les paramètres et variables locales sont accessibles

Récapitulatif : usage de la pile (4)

- Séquence de retour d'une procédure, partie gérée par la procédure appelée (fin de procédure) :
  - Ajout à SP du nombre d'octets nécessaire pour « oublier » les variables locales
  - Dépilage des registres *callee save*
  - Appel de l'instruction de retour
    - Dépile l'adresse de retour située dans la pile

université de BORDEAUX

Récapitulatif : usage de la pile (5)

- Séquence de retour d'une procédure, partie gérée par la procédure appelante :
  - Ajout à SP de la taille de tous les paramètres empilés avant l'appel de la procédure
  - Dépilage éventuel des registres *caller save*
  - Retour complet à l'état antérieur

université de BORDEAUX

Récapitulatif : usage de la pile (6)

- Au final, pour la fonction appelante :

```
        pushl  %ecx                 # Sauvegarde caller-save
        irmovl 12,%eax
        pushl  %eax                 # Empilage deuxième paramètre
        irmovl 42,%eax
        pushl  %eax                 # Empilage premier paramètre
        call   func                 # Appel de la fonction
        iaddl  8,%esp               # Suppression paramètres
        popl   %ecx                 # Restauration caller-save
```

Récapitulatif : usage de la pile (7)

- Au final, pour la fonction appelée :

```
func:   pushl  %esi                 # Sauvegarde callee-save
        pushl  %edi                 # Sauvegarde callee-save
        isubl  4,%esp               # Réservation variable locale
        ...
        mrmovl 16(%esp),%eax        # Lecture premier paramètre
        mrmovl 20(%esp),%esi        # Lecture deuxième paramètre
        rmmovl %esi,(%esp)          # Écriture variable locale
        ...
        iaddl  4,%esp               # Suppression variable locale
        popl   %edi                 # Restauration callee-save
        popl   %esi                 # Restauration callee-save
        ret                         # Retour de la fonction
```

université de BORDEAUX

Branchement à adresse variable (1)

- Les instructions de branchement dont on dispose ne permettent que de se brancher à une adresse constante :
  « *jop* constante »
- Il existe pourtant des situations dans lesquelles on doit pouvoir effectuer un branchement à une adresse variable :
  - Pointeurs de fonction : « call registre »
  - Branchement à l'adresse d'un « case: » pour un « switch... case... »

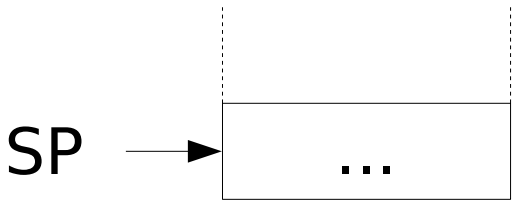
Branchement à adresse variable (2)

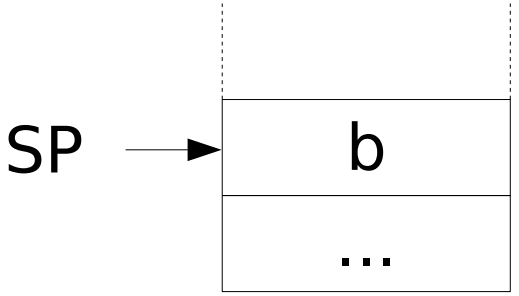
- On peut cependant émuler un branchement à l'adresse contenue dans un registre en utilisant les branchements par la pile

```
        irmovl b,%eax
a:      pushl  %eax
        ret


b:      ...
```

PC = a

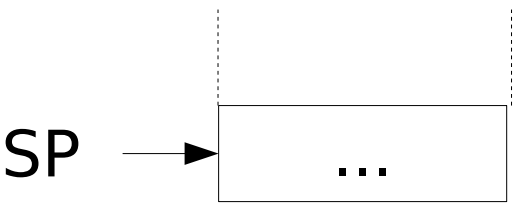
AX = b





PC = b

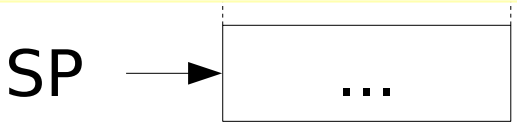
AX = b



Branchement à adresse variable (3)

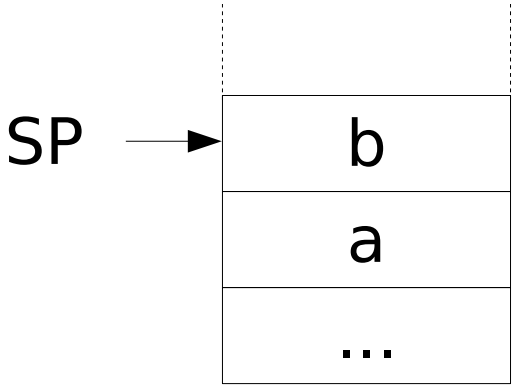
- Pour mettre en œuvre un pointeur de fonction, il faut faire un « call », pas un « jmp »
  - Il faut arriver à placer l'adresse de retour dans la pile

```
        irmovl b,%eax
        call   fcall
a:      ...

fcall:  pushl  %eax
c:      ret

b:      ...
        ret
```



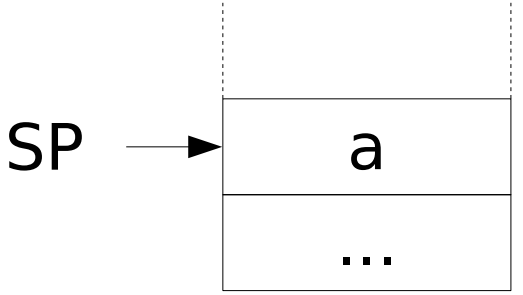
PC = c

AX = b



PC = b

AX = b



université de BORDEAUX

Commentaires (1)

- Plus le langage est rudimentaire, plus il faut de commentaires
  - Pas de noms de variables (registres)
  - Pas de typage explicite
- En langage machine, il faut au moins une ligne de commentaire par instruction
- Ne pas faire de paraphrase !
  - Il faut ajouter une information de plus haut niveau utile à la compréhension du lecteur

```
        isubl   1,%ecx          # Décrémente ecx
```

université de BORDEAUX

Commentaires (2)

- Exemples de commentaires pertinents

```
mult:     xorl    %eax,%eax   # Met à zéro le résultat
multbcl:  andl    %ebx,%ebx   # Teste le multiplicateur y
          je      multfin     # Si y vaut zéro, fin de la fonction
          addl    %ecx,%eax   # Ajoute le multiplicande x
          isubl   1,%ebx      # Décrémente le multiplicateur y
          jmp     multbcl     # Reboucle
multfin:  ret                 # Retourne x * y
```

```
somme:  rrmovl %esp,%edx    # Initialise le pointeur de travail
        xorl   %eax,%eax    # Initialise la somme courante à zéro
boucle: iaddl  4,%edx       # Avance le pointeur à la case suivante
        mrmovl (%edx),%ecx  # Lit la valeur de la case courante
        andl   %ecx,%ecx    # Teste la valeur courante
        je     fin          # Termine si la case courante vaut zéro
        addl   %ecx,%eax    # Ajoute la valeur courante à la somme
        jmp    boucle       # Passe à l'itération suivante
fin:    ret                 # Retourne la somme courante
```