Compilation : théorie, techniques et outils

Thème 0 : Compilation

Habib ABDULRAB (Institut National des Sciences Appliquées de Rouen)
Claude MOULIN (Université de Technologie de Compiègne)
Sid TOUATI (Université de Versailles Saint-Quentin en Yvelines)

Table des matières

Objectifs *5*

I - Contenu du cours *7*

A.Thèmes du cours.....*7*

B.Pourquoi étudier la compilation ?.....*8*

C.Bibliographie.....*8*

II - Structure d'un compilateur *11*

A.Un compilateur.....*11*

B.Chaîne de transformation.....*12*

C.Bref historique.....*13*

D.Qualités d'un compilateur.....*14*

E.Portage d'un compilateur.....*14*

F.Pré-requis pour construire un bon compilateur.....*15*

G.Terminologie.....*15*

H.Phases d'un compilateur.....*15*

I.Structures de données.....*16*

J.Techniques d'optimisation.....*17*

III - Code machine *19*

A.Code cible.....*19*

B.Processeurs.....*19*

C.Jeu d'instructions (ISA).....*19*

D.CISC vs. RISC.....*20*

IV - Conclusion *21*

A.Conclusion.....21

V - Annexe **23**

A.Code machine, cible des compilateur.....23

1.Exemple d'architecture CISC: x86.....23

2.Exemple d'architecture RISC: MIPS.....26

Objectifs

Ce cours est destiné aux étudiants de deuxième cycle (Master et écoles d'ingénieurs).

Contenu du cours

1

Thèmes du cours	7
Pourquoi étudier la compilation ?	8
Bibliographie	8

A. Thèmes du cours

Thème 0 : Introduction générale

Il explique ce qu'est la compilation et ses enjeux...

Thème 1 : Analyse lexicale

Il présente les automates, les expressions régulières et les scanners (transducteurs). Il explique comment créer automatiquement des scanners permettant de retrouver les unités lexicales dans un programme source. Il présente également des rudiments de Lex.

Thème 2 : Analyse syntaxique

Se basant sur des pré-requis en théorie des langages (grammaires et automates à pile), il montre les aspects théoriques à partir desquels on peut créer automatiquement des parseurs de langages structurés. Il aborde les deux grandes familles de parseurs : LL et LR. Des rudiments de présentation des outils YACC et AntLR sont également apportés.

Thème 3 : Traduction dirigée par la syntaxe

Il présente les notions de traduction S-attribuée et L-attribuée. Il explique les mécanismes permettant d'extraire des informations sémantiques sur un programme source. Il définit la notion d'attribut et les différents types d'attributs, ainsi que des règles sémantiques et donne quelques exemples de traductions dirigées par la syntaxe.

Thème 4 : Table de symboles

Il présente la construction d'une table de symboles, et de structures de données centrales dans un compilateur. Il montre aussi des implémentations efficaces et des interfaces.

Thème 5 : Gestion de type

Il présente la notion de types dans un compilateur ainsi que les problèmes d'analyse de type et de vérification de types.Il donne l'exemple d'un système de

typage simple.

Thème 6 : Gestion mémoire

Il présente les processus d'allocationet de substitution. Il montre comment fait un compilateur pour transformer des données de haut niveau en données de bas niveau (mémoire linéaire).

Thème 7 : Représentations intermédiaires

Il définit des structures et langages intermédiaires utilisés en compilation, comme la structure en graphes (dépendances de données, dépendances de contrôle), et des formes textuelles (codes 3 adresses, codes 2 adresses, notation préfixées et postfixées, SSA, etc.).

Thème 8 : Génération de code

Il montre l'utilisation de la traduction dirigée par la syntaxe pour générer du code valide et en particulier la génération de code sur machine à pile et sur machine à registres. Il donne des exemples de génération de code pour des expression arithmétique, des structures de contrôle, des appels de fonctions, etc. Il présente la génération de code en une passe et en deux passes.

Thème 9 : Introduction à l'optimisation de code

Il présente les enjeux, les difficultés et quelques techniques de base de génération de code efficace.

Remarque

Les différents thèmes contiennent des exemples, exercices résolus, et sont suivis par des TDs qui relèvent d'un ou plusieurs thèmes.

B.Pourquoi étudier la compilation ?

- C'est une des branches les plus dynamiques de l'informatique, et une des plus anciennes à mériter ce qualificatif.
- Son domaine d'application est très large : conversion de formats ; traduction de langages ; optimisation de programmes.
- Elle permet de mettre en pratique de nombreux algorithmes généraux.

C.Bibliographie

Un livre standard, appelé « dragon book »

- Compilateurs : Principes, techniques et outils. A. Aho, R. Sethi et J. Ullman. InterEditions.

Autres livres intéressants :

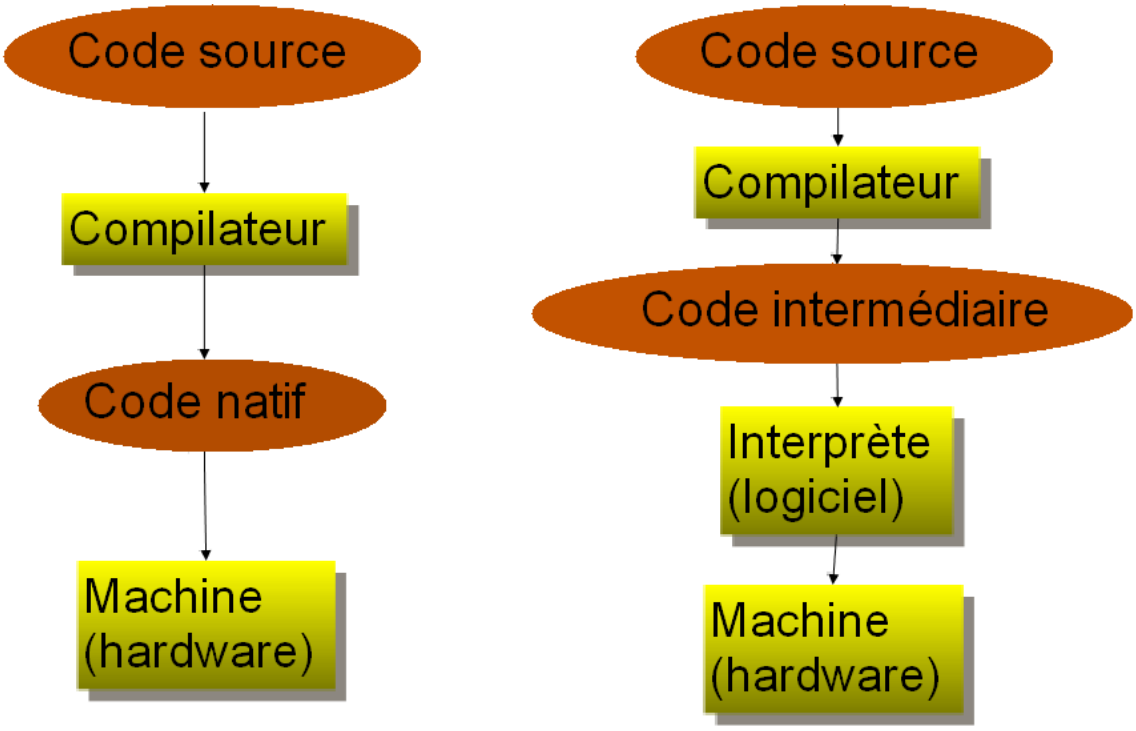
- Compilateurs. D. Grune, H.E. Bal, G. J.H. Jacobs, K.G. Langendoen. Editions Dunod.
- Lex & yacc. John R. Levine, Tony Mason et Doug Brown. Edition O'Reilly.
- The Definitive ANTLR Reference. Terence Parr. The Pragmatic Programmers, 2007.

Structure d'un compilateur

II

Un compilateur	11
Chaîne de transformation	12
Bref historique	13
Qualités d'un compilateur	14
Portage d'un compilateur	14
Pré-requis pour construire un bon compilateur	15
Terminologie	15
Phases d'un compilateur	15
Structures de données	16
Techniques d'optimisation	17

A.Un compilateur



Complément

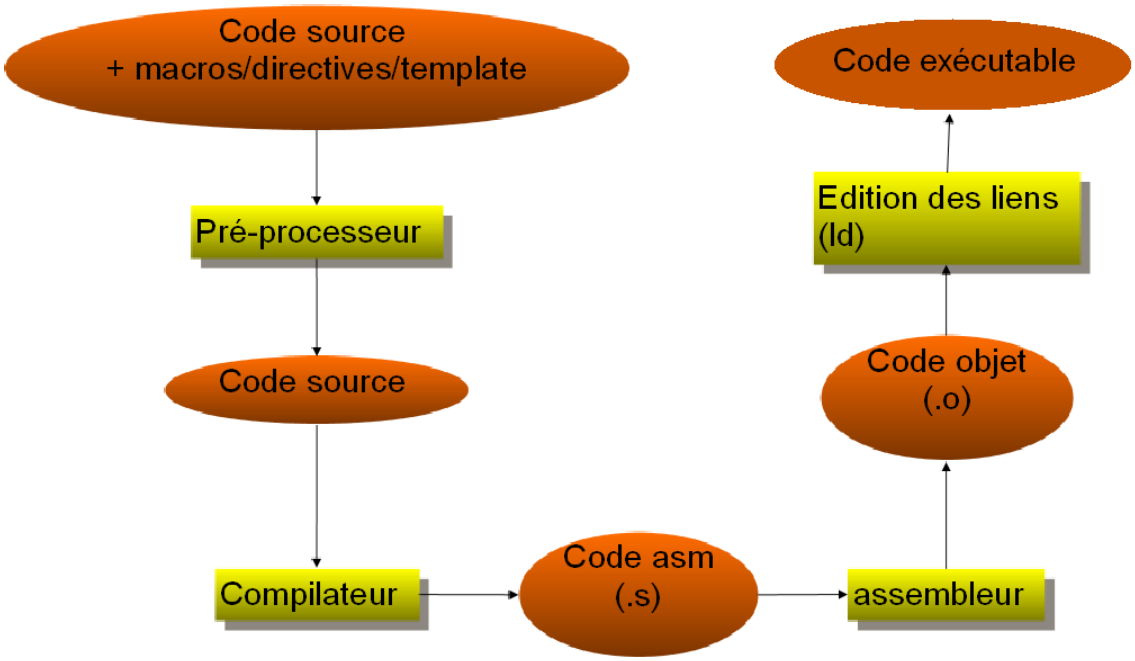
Réponse fausse : c'est la machine qui m'exécute mon programme.

Réponse correcte : c'est un programme qui transforme mon programme (fichier texte) en fichier binaire.

Meilleure réponse : c'est un traducteur de langages.

Encore mieux : c'est un traducteur de langages, et un analyseur/optimiseur.

B.Chaîne de transformation



Complément

En pratique, un compilateur fait toutes ces étapes.

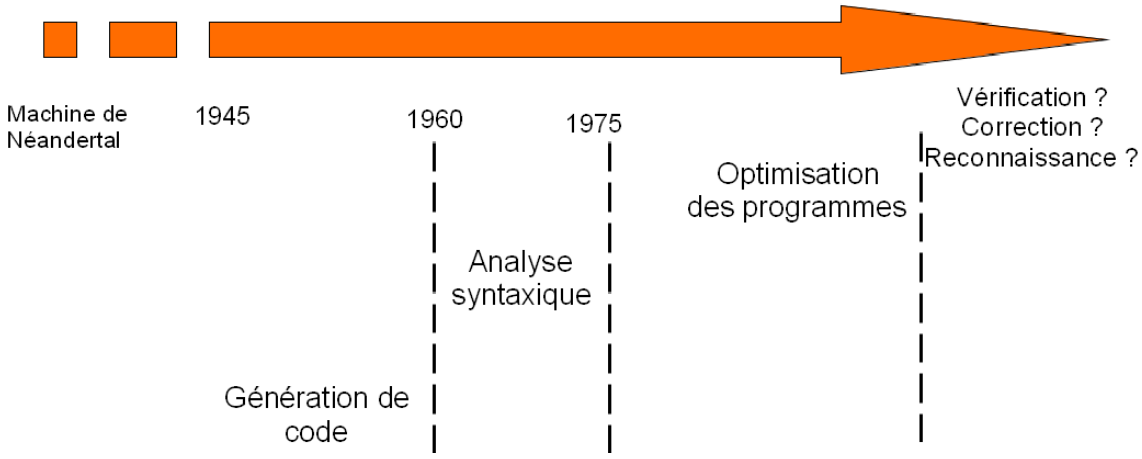
Assembleur :

- Transforme des instructions asm (texte) en binaire. C'est une correspondance presque directe.
- Résolution de certaines étiquettes.
- Suit les directives (déclarations, macros, etc.)

Édition des liens (link) :

- Remplacer les noms des variables et fonctions par des adresses.
- Fusionner les .o pour créer une application complète (cas d'un link statique)
- Générer les appels vers le chargeur dynamique (cas d'un link dynamique)

C.Bref historique



Complément

45->60 : programmation asm, très peu de langages. Pb : comment produire du code pour une machine donnée.

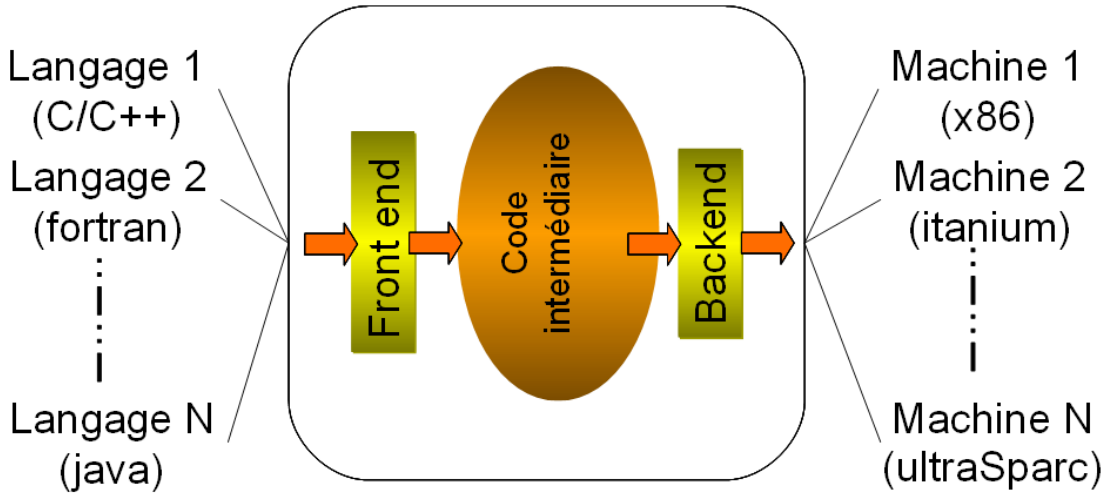
60->75 : théorie des automates et grammaires. Prolifération des nouveaux langages. Produire rapidement un compilateur.

75 -> : nouveaux langages « stabilisés ». Forte demande de compilateurs fiables et efficaces.

D.Qualités d'un compilateur

- S'approcher du « zéro bug » : confiance.
- Code généré rapide : optimisation.
- Compilateur rapide : efficacité.
- Compilateur portable : économie.
- Erreurs signalées utiles : productivité.
- Ses optimisations doivent être consistantes et prévisibles : un vœu pieux.

E.Portage d'un compilateur



Complément

A des fins d'économie, on essaye tant bien que mal de porter un compilateur existant vers différents langages et architectures.

Dans ce cours, nous nous contentons des langages impératifs, et modèles de machines von Neumann (langage impératif aussi).

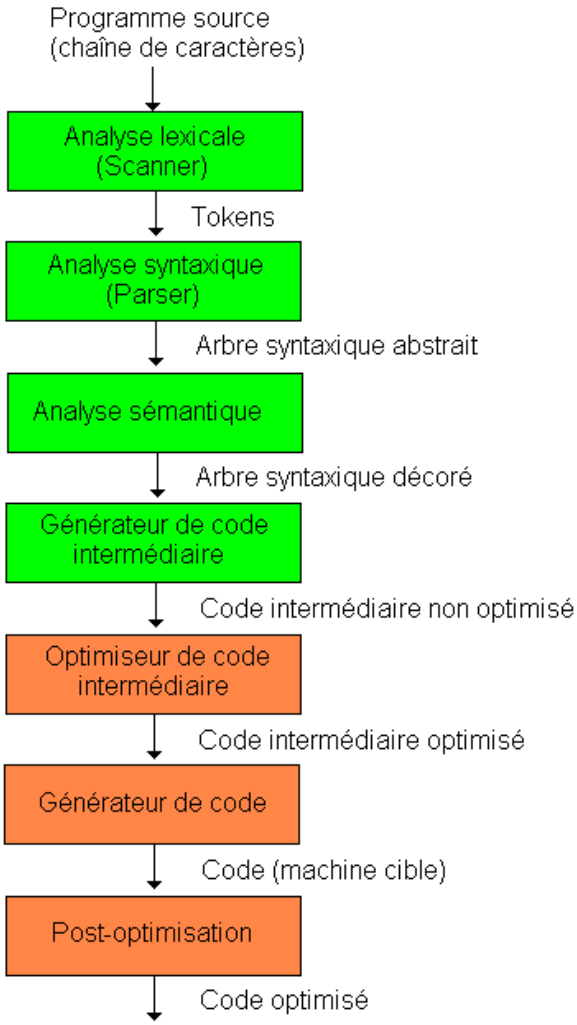
F.Pré-requis pour construire un bon compilateur

- Caractéristiques des langages de programmation (passage des paramètres, allocation mémoire, ...)
- Théorie (automates, grammaires, ...)
- Algorithmique et structures des données (table de hachage, graphes, etc.)
- Architecture des ordinateurs (programmation assembleur, bas niveau)
- Engineering logiciel.

G.Terminologie

- Lexicalement correcte ?
  - ΦϑΔOui papA
  - int i, j k void ; main(]; 54j
- Syntaxiquement correcte ?
  - Oui Moi demain partir
  - I = i j;
- Sémantiquement correcte ?
  - Le ciel éplucha l'oiseau.
  - int i; i = i + 1.5;

H.Phases d'un compilateur



Complément

Cette structure existe dans tous les compilateurs.

Le code natif est soit du asm, soit du .o, soit du binaire.

En réalité, un compilateur est plus compliqué que cela, cela devient vite une véritable machine à gaz à cause des optimisations haut/bas niveau.

I.Structures de données

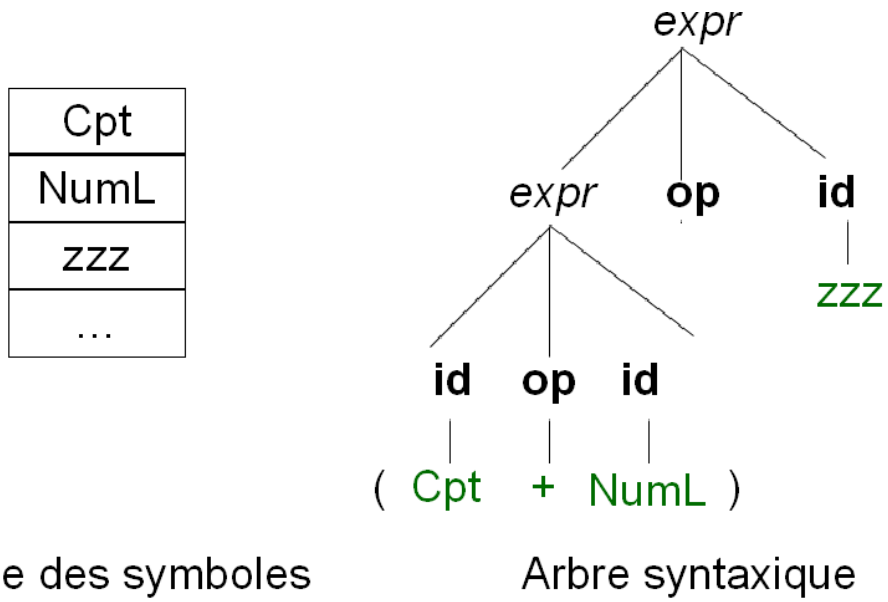


Table des symboles : dictionnaire contenant des informations sur les entités du programme à compiler (variables, fonctions, types, etc.). Elle est initialisée dès la phase d'analyse lexicale; elle est continuellement mise à jour au fur et à mesure.

Arbre syntaxique (arbre de dérivation): décrit la structure syntaxique d'un programme. Il est construit par l'analyseur syntaxique. Il est décoré par l'analyse sémantique.

J.Techniques d'optimisation

- But principal : rendre le programme plus rapide, tout en gardant des délais de compilation raisonnables.
- Autres buts : générer un code réduit, consommation d'énergie, puissance, etc.
- Ce sont des passes de compilation qui se greffent par dessus un compilateur existant.
- C'est un domaine de recherche continuel, très passionnant d'ailleurs !

Complément

Analyseur de code : une étape de compilation qui essaye de déduire des caractéristiques d'un programme. Peut servir pour l'optimisation, vérification, débogage, etc.

Optimisation de code : réécrire un programme (en préservant son résultat) pour gagner en temps d'exécution, en espace, etc.

Transformation de programme : réécrire un programme à divers buts, non nécessairement pour l'optimisation uniquement. Ex : rendre un programme plus/moins clair à l'utilisateur.

Code machine

III

Code cible	19
Processeurs	19
Jeu d'instructions (ISA)	19
CISC vs. RISC	20

Vers où allons-nous compiler ?

A.Code cible

Un compilateur peut générer:

- Un code assembleur cible
- Un code d'un autre langage de programmation haut niveau (ex: C vers fortran, C++ vers java)
- Représentation intermédiaire (ex: java vers bytecode)

B.Processeurs

- Processeurs à hautes performances (stations de travail, PC, super calculateurs, etc.)
  - Alpha (HP), UltraSparc (Sun), Pentium (Intel), Itanium (HP/Intel), Power (IBM), ...
- Processeurs embarqués : ils sont partout...
  - Mips, PowerPC, ARM, Xscale, ST 2xx
- Microcontrôleurs:
  - Électroménager, voiture, robots, etc.

On peut les classer selon d'autres critères

C.Jeu d'instructions (ISA)

- C'est l'ensemble des instructions exécutées par un processeur.
- Un des critères de classification :
  - CISC : Intel est dominant (x86). AMD.

  - RISC : majorité des autres processeurs.
- L'ISA est un langage de programmation bas niveau :
  - Non portable, difficile à vérifier/déboguer;
  - Programmes efficaces.

D.CISC vs. RISC

CISC	RISC
Opérations complexes pour un processeur complexe: opérations proches du langage haut niveau: ex MOVS	Opérations élémentaires pour un processeur simplifié : données en registres, opérations registre-registre, chargement mémoire
Peu de registres: ex 4	Plus de registres: 32, 64, 128
Opérations ≪ destructives ≫ : accumulateur	Pas d'effets de bord
Taille variable d'instructions, décodage matériel compliqué.	Taille fixe d'instructions
Optimisation de code plus compliquée	Le compilateur peut mieux optimiser le code RISC
Taille de code réduite	Taille de code plus conséquente

Inventeurs du RISC: Invention: Paterson (Pr université de Berkeley), Cocke (IBM), Hennessy

- Le débat est clos!
- Les compilateurs ne réussissent pas à générer des codes CISC très optimisés
  - C'est le processeur CISC qui accélère les programmes
- Intel a démontré que la solution CISC est vivable:
  - Le pentium pro décompose les opérations CISC en opérations RISC
  - Le parc des logiciels déployés dans les PC a maintenu le marché des processeurs CISC.

Conclusion

IV

A.Conclusion

- La compilation est une branche passionnante et ardue de l'informatique.
- Le front-end d'un compilateur permet:
  - Vérifier la validité lexico-syntaxico-sémantique du programme vis à vis de son langage.
  - Le traduire en un langage intermédiaire
- Le back-end permet de générer un code bas niveau vers une architecture donnée.
- Le cœur du compilateur analyse et optimise le programme.

Annexe

V

A.Code machine, cible des compilateur

1.Exemple d'architecture CISC: x86

Exemple : CISC (x86)

- C'est la base de l'ISA d'intel
- On l'étudie juste parce que vos machines fonctionnent sous linux avec des processeurs intel.
- Des extensions au fur et à mesure : x86, 386, MMX, SSE, etc.

Spécialités i386

- Interaction avec l'OS : gestion d'interruptions, etc...
- Gestion de la pile : pop et push
- Chaînes de caractères
  - Movs : copie
  - Cmps : comparaison
  - ...
- Boucles
  - loop label : répéter jusqu'à ECX=0
- MMX (multimédia)
- Et plus encore,...

Ces features sont en plus des capacités standard (opérations arithmétiques, logiques, branchements conditionnels et inconditionnels, etc.)

Registres (i386)

- 4 registres 32 bits généraux
  - %EAX : accumulateur
  - %EBX : Index de base
  - %ECX : compteur
  - %EDX : donnée
- Registres de pile : %EBP, %ESP
- Chaînes de caractères : %ESI, %EDI

Complément

EBP : pointeur de base de pile (base pointer)

ESP : pointeur du sommet de pile (stack pointer)

ESI : source index (pointe sur la lettre courante d'une chaîne source)

EDI : destination index (pointe sur la lettre courante d'une chaîne dest)

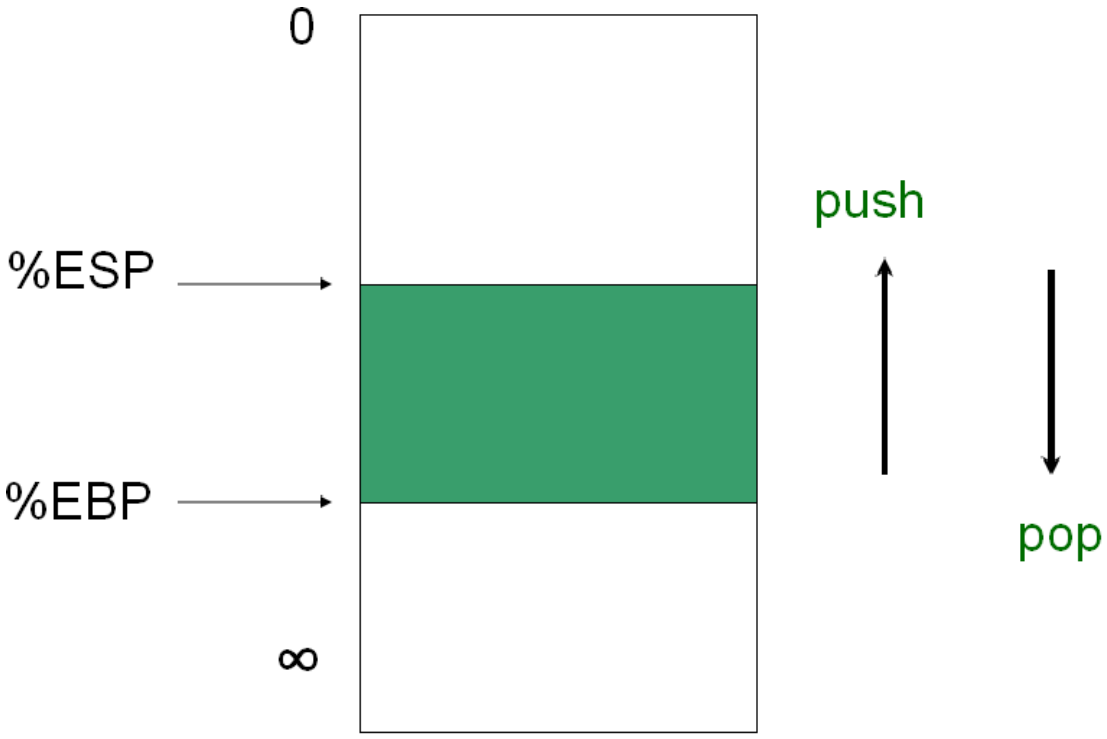
- Registres de segments : CS, DS, ES, FS, GS
  - Utilisés sous dos pour segmenter la mémoire;
  - Sous linux, utilisés par l'environnement d'exécution (libc, OS, etc.).
- Registre d'états 32 bits: EFLAG.
- Registre PC 32 bits : EIP
- Compteurs hardware : profiling, etc.
- ...

Complément

Conseil sous linux pour les registres des segments: ne pas penser à les sauvegarder et restaurer.

EFLAG : chaque bit a une signification; résultat de test, d'une opération, contrôle des interruptions

Pile



Deux assembleurs i386

- Il existe deux syntaxes i386
  - La syntaxe d'Intel : assembleur d'intel (dos, windows).
  - La syntaxe AT&T : utilisée par gas/gcc (linux, cygwin)
- Un site : www.x86.org

Syntaxe générale (gas)

- Opcode[b,w,l] src, dest
  - b : byte
  - w : word
  - l : long
- Registre : %reg
- Opérande mémoire :
  - offset(base, index, facteur)
- Constantes (opérandes immédiates)
  - $74, 0x4A, 0f-356.666e-36, `c', ''toto''

une seule opérande mémoire par instruction

Modes d'adressage (i386)

- Immédiat : opérande = exp_constante.
  - Par défaut : en décimal
  - En hex : 0x456
  - En binaire, ...
- Direct : opérande = contenu d'un registre
- Mémoire : syntaxe compliquée
  - offset(base, index, facteur)
  - Adresse = base + index * facteur + offset
  - Ex : 0(%eax, %ebx, 2) pointe sur la donnée se trouvant à l'adresse %eax + 2 * %ebx + 0

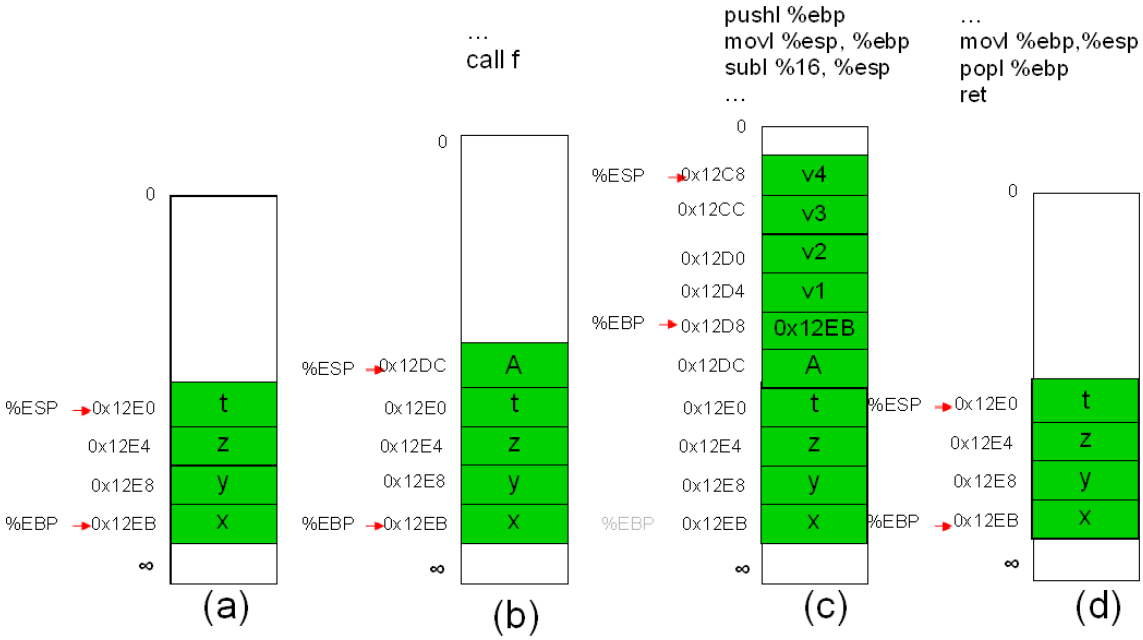
Facteur = 1, 2, 4 seulement (et peut être 8, mais je ne suis pas sûr)

Conventions d'appel (i386)

- Par défaut, les paramètres passés à la fonction appelée sont empilés.
- Résultat sur 32 bits dans %eax (par défaut)
- %ebp, %esp : callee save
- Dans le cas de plusieurs résultats:
  - Passage par adresse : résultats écrits directement en mémoire

Certaines directives de compilation peuvent forcer le passage des paramètres par registre

Conventions d'appel x86



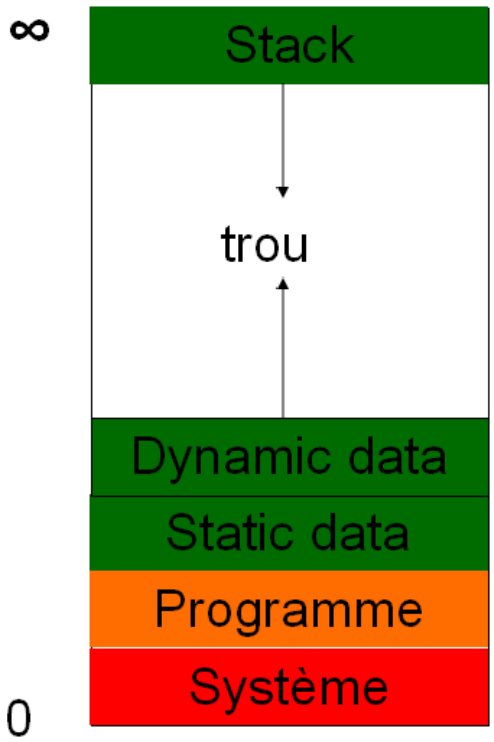
1. Programme en cours.
2. call f empile l'adresse de l'instruction de retour, i.e., (l'instruction suivant le call)
3. le callee :
   a. sauvegarde de %ebp de l'appelant sur la pile
   b. nouvelle valeur de %ebp (pile vide)
   c. allocation de 4 variables locales de 4 octets : %esp = %esp – 4
4. à la fin du callee
   a. Le résultat du calcul est supposé dans %EAX
   b. restaurer l'ancienne valeur de %esp du caller
   c. restaurer l'ancienne valeur de %ebp du caller
   d. retour : dépiler l'adresse de retour

2.Exemple d'architecture RISC: MIPS

Exemple : RISC : MIPS

- Un processeur avec une conception élégante.
- Jeu d'instructions RISC, facile à utiliser.
- Hélas, ce ne fut pas un succès commercial...

Organisation de la mémoire (MIPS)



Système : zone inaccessible au programme (ni en lecture ni en écriture).

Programme : zone où le code binaire est stocké, inaccessible en écriture (cela accélère la lecture).

Static data : zone des données statiques (variables, tableaux,, etc.).

Dynamic data : quand on fait un malloc. Cette zone peut déborder sur le trou.

Trou: zone inaccessible. Permet d'élargir la pile où la zone dynamique.

Stack : pile (appels de fonctions avec les variables locales).

Registres (MIPS)

- 32 registres généraux (numérotés).
- Pas d'opérations mémoire-mémoire ou registre-mémoire.
- Certains registres sont réservés:
  - $0 = 0 toujours
  - $2, $3 : retour de valeurs (après appel de fct)
  - $4, ...$7 : passage d'arguments
  - ...

$0 est appelé zéro

$1 est appelé at

$2 et $3 sont appelés v0 et v1

$4,...,$7 sont appelés $a0,...,a3

Mode d'adressage (MIPS)

- Immédiat : opérande = un entier.
- Direct : opérande = contenu d'un registre
- Indirect : adresse = contenu d'un pointeur = contenu d'un registre
- Indirect indexé : adresse = indirect + déplacement (offset)

Le mode d'adressage est défini par la syntaxe des arguments :

Ex : lw r1, offset(r2) : charger la donnée se trouvant à l'adresse r2 + offset

ISA (MIPS)

- Transfert de registres : move
- Arithmétique : add, sub, div, mul
- Logique : and, or, xor, sll, srl
- Chargement immédiat : li, la
- Accès mémoire : lw, sw
- Test : slt, sle, seq, sne
- Branch. Cond et incond., appels système etc.

Conventions d'appel (MIPS)

- Arguments dans les registres a0,...,a3
- Le reste dans la pile,...
- Résultats dans v0 et v1
- Registres s0 à s7 : callee save
- Registres t0 à t9 : caller save
- Registre ra : adresse de retrour
- sp : pointeur de pile
- ...

Utilisation simple de la pile

- Contient les args et variables locales des fonctions (C, pascal, java).
- Permet de ne pas se préoccuper des registres
  - construire un compilateur rapidement,
  - machines virtuelles (java):
    - Peu de registres spécialisés
    - Parfois un accumulateur

Exemple : une addition dépile deux arguments et empile un résultat