

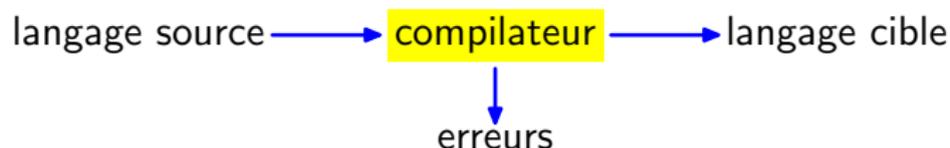
École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 2 / 2 octobre 2008

schématiquement, un compilateur est un programme qui traduit un « programme » d'un langage **source** vers un langage **cible**, en signalant d'éventuelles erreurs



Compilation vers le langage machine

quand on parle de compilation, on pense typiquement à la traduction d'un langage de haut niveau (C, Java, Caml, ...) vers le langage machine d'un processeur (Intel Pentium, PowerPC, ...)

```
% gcc -o sum sum.c
```

source `sum.c` → **compilateur C (gcc)** → exécutable `sum`

```
int main(int argc, char **argv) {  
    int i, s = 0;  
    for (i = 0; i <= 100; i++) s += i*i;  
    printf("0*0+...+100*100 = %d\n", s);  
}
```



```
0010011111011111011111111111111100000  
10101111110111111100000000000010100  
1010111111010010000000000000100000  
1010111111010010100000000000100100  
101011111101000000000000000011000  
101011111101000000000000000011100  
100011111101011110000000000011100  
...
```

dans ce cours, nous allons effectivement nous intéresser à la compilation vers de **l'assembleur**, mais ce n'est qu'un aspect de la compilation

un certain nombre de techniques mises en œuvre dans la compilation ne sont pas liées à la production de code assembleur

certains langages sont d'ailleurs

- interprétés (Basic, COBOL, Ruby, etc.)
- compilés dans un langage intermédiaire qui est ensuite interprété (Java, Caml, etc.)
- compilés vers un autre langage de haut niveau
- compilés à la volée

Différence entre compilateur et interprète

un **compilateur** traduit un programme P en un programme Q tel que pour toute entrée x , la sortie de $Q(x)$ soit la même que celle de $P(x)$

$$\forall P \exists Q \forall x \dots$$

un **interprète** est un programme qui, étant donné un programme P et une entrée x , calcule la sortie s de $P(x)$

$$\forall P \forall x \exists s \dots$$

dit autrement,

le compilateur fait un travail complexe une seule fois, pour produire un code fonctionnant pour n'importe quelle entrée

l'interprète effectue un travail plus simple, mais le refait sur chaque entrée

autre différence : le code compilé est généralement bien plus efficace que le code interprété

Exemple de compilation et d'interprétation

source → lilypond → fichier PostScript → gs → image

```
<<  
  \chords { c2 c f2 c }  
  \new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }  
  \new Lyrics \lyricmode { twin4 kle twin kle lit tle star2 }  
>>
```

C C F C

twin kle twin kle lit tle star

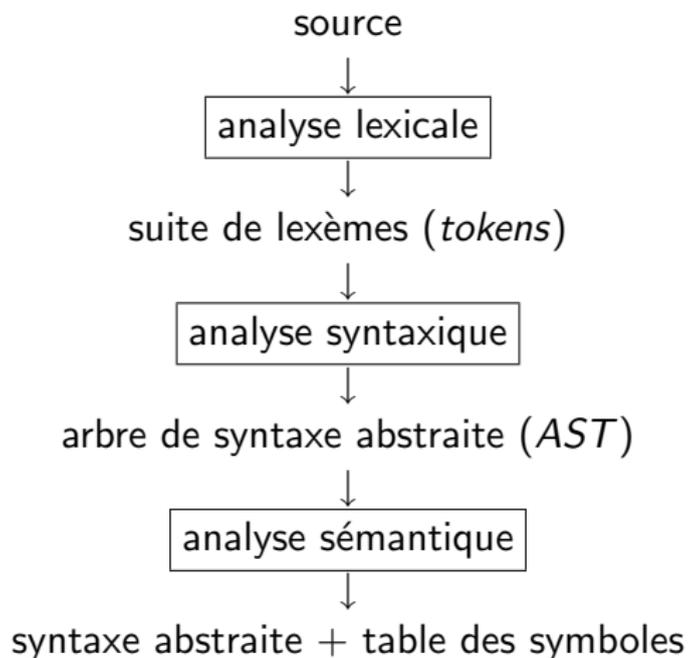
À quoi juge-t-on la qualité d'un compilateur ?

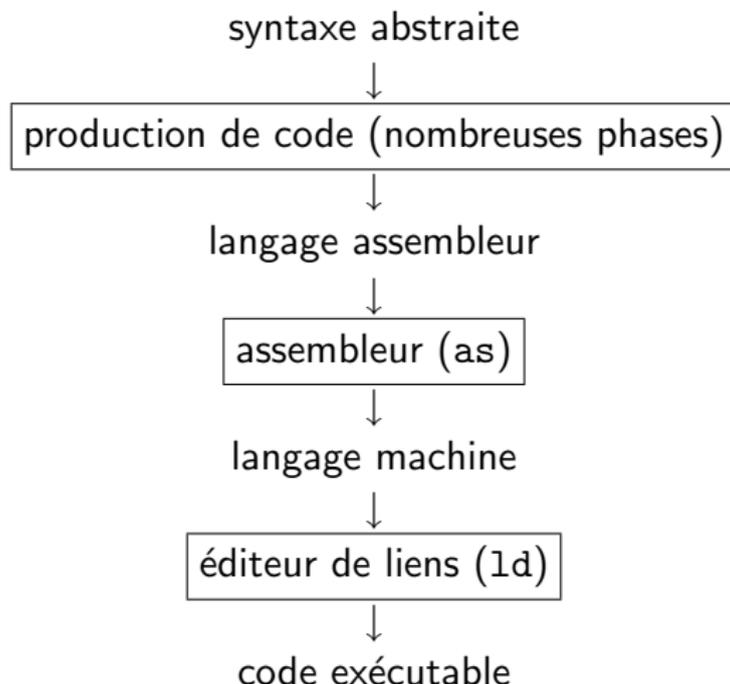
- à sa correction
- à l'efficacité du code qu'il produit
- à sa propre efficacité

typiquement, le travail d'un compilateur se compose

- d'une phase d'**analyse**
 - reconnaît le programme à traduire et sa signification
 - signale les erreurs et peut donc échouer (erreurs de syntaxe, de portée, de typage, etc.)

- puis d'une phase de **synthèse**
 - production du langage cible
 - utilise de nombreux langages intermédiaires
 - n'échoue pas





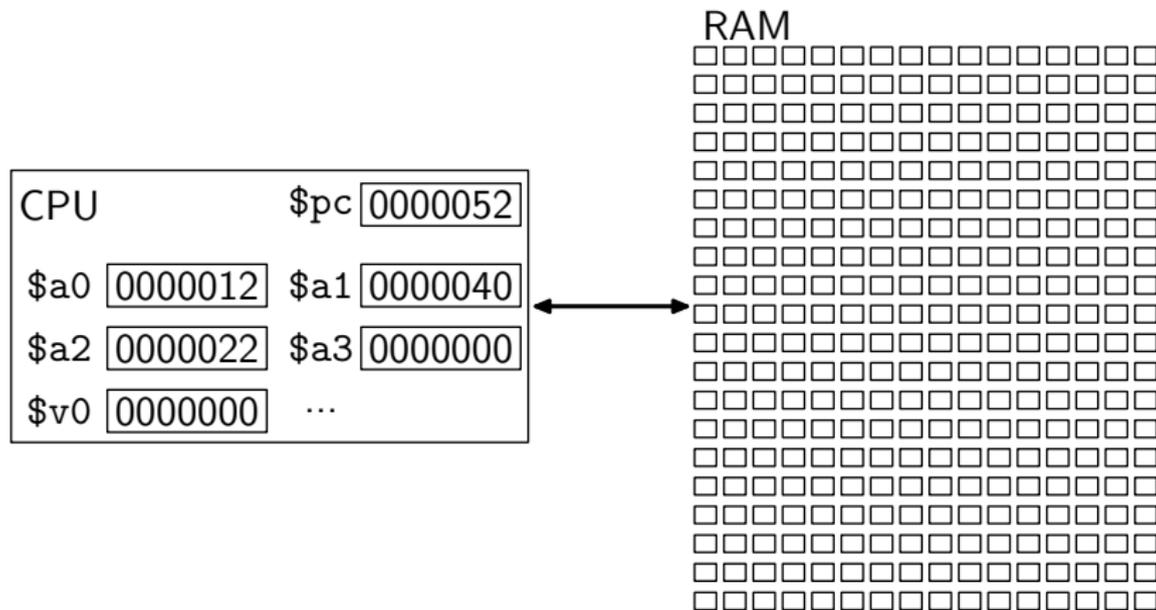
l'assembleur MIPS

(voir *SPIM*, a *MIPS32 Simulator* sur la page du cours)

très schématiquement, un ordinateur est composé

- d'une unité de calcul (CPU), contenant
 - un petit nombre de registres entiers ou flottants
 - des capacités de calcul
- d'une mémoire vive (RAM)
 - composée d'un très grand nombre d'octets (8 bits)
par exemple, 1 Go = 2^{30} octets = 2^{33} bits, soit $2^{2^{33}}$ états possibles
 - contient des données et des instructions

Un peu d'architecture



l'accès à la mémoire coûte cher (à un milliard d'instructions par seconde, la lumière ne parcourt que 30 centimètres entre 2 instructions !)

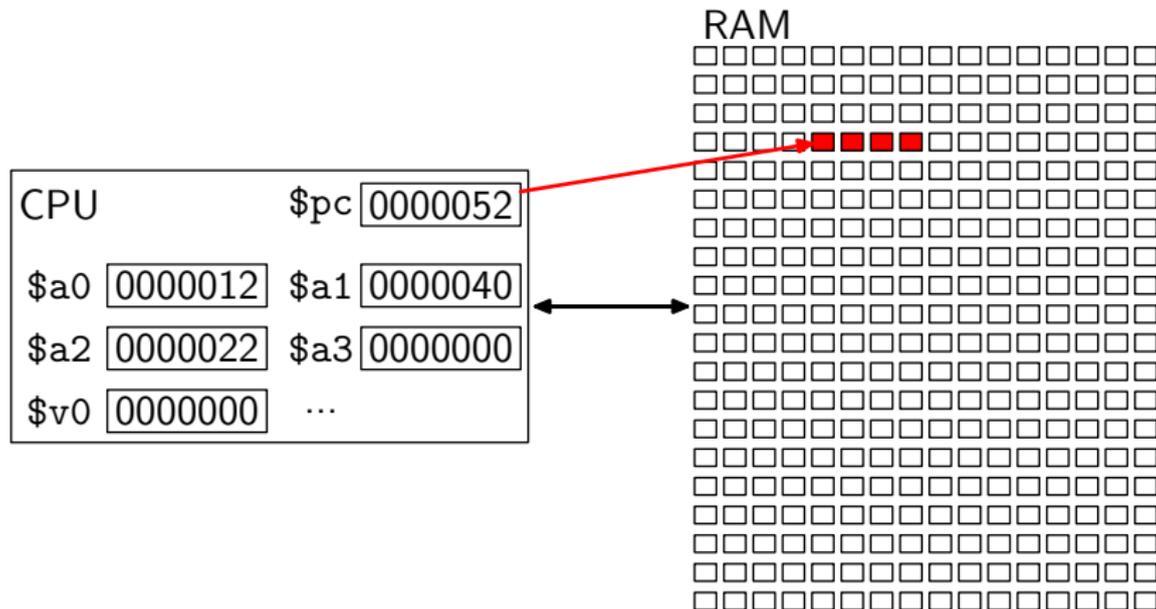
la réalité est bien plus complexe

- plusieurs (co)processeurs, dont certains dédiés aux flottants
- un ou plusieurs caches
- une virtualisation de la mémoire (MMU)
- etc.

schématiquement, l'exécution d'un programme se déroule ainsi

- un registre ($\$pc$) contient l'adresse de l'instruction à exécuter
- on lit les 4 (ou 8) octets à cette adresse (*fetch*)
- on interprète ces bits comme une instruction (*decode*)
- on exécute l'instruction (*execute*)
- on modifie le registre $\$pc$ pour passer à l'instruction suivante (typiquement celle se trouvant juste après, sauf en cas de saut)

Principe d'exécution



instruction :

000000	00001	00010	0000000000001010
--------	-------	-------	------------------

décodage : add \$a1 \$a2 10

i.e. ajouter 10 au registre \$a1 et stocker le résultat dans le registre \$a2

là encore la réalité est bien plus complexe

- pipelines
- prédiction de branchement

Quelle architecture pour ce cours ?

deux grandes familles de microprocesseurs

- CISC (*Complex Instruction Set*)
 - beaucoup d'instructions
 - beaucoup de modes d'adressage
 - beaucoup d'instructions lisent / écrivent en mémoire
 - peu de registres
 - exemples : VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
 - peu d'instructions, régulières
 - très peu d'instructions lisent / écrivent en mémoire
 - beaucoup de registres, uniformes
 - exemples : Alpha, Sparc, MIPS, ARM

on choisit **MIPS** pour ce cours (les TD et le projet)

- 32 registres, r0 à r31
 - r0 contient toujours 0
 - utilisables sous d'autres noms, correspondant à des conventions (zero, at, v0–v1, a0–a3, t0–t9, s0–s7, k0–k1, gp, sp, fp, ra)
- trois types d'instructions
 - instructions de transfert, entre registres et mémoire
 - instructions de calcul
 - instructions de saut

documentation : sur le site du cours

en pratique, on utilisera un simulateur MIPS, **SPIM**

en ligne de commande

- `spim file.s`

en mode graphique et interactif

- `xpsim --file file.s`
- mode pas à pas, visualisation des registres, de la mémoire, etc.

documentation : sur le site du cours

- chargement d'une constante (16 bits signée) dans un registre

```
li    $a0, 42    # a0 <- 42
lui   $a0, 42    # a0 <- 42 * 2^16
```

- copie d'un registre dans un autre

```
move $a0, $a1    # copie a1 dans a0 !
```

Jeu d'instructions : arithmétique

- addition de deux registres

```
add  $a0, $a1, $a2  # a0 <- a1 + a2
add  $a2, $a2, $t5  # a2 <- a2 + t5
```

de même, sub, mul, div

- addition d'un registre et d'une constante

```
addi $a0, $a1, 42  # a0 <- a1 + 42
```

(mais pas subi, muli ou divi !)

- négation

```
neg  $a0, $a1      # a0 <- -a1
```

- valeur absolue

```
abs  $a0, $a1      # a0 <- |a1|
```

Jeu d'instructions : opérations sur les bits

- NON logique ($\text{not}(100111_2) = 011000_2$)

```
not  $a0, $a1      # a0 <- not(a1)
```

- ET logique ($\text{and}(100111_2, 101001_2) = 100001_2$)

```
and  $a0, $a1, $a2 # a0 <- and(a1, a2)
andi $a0, $a1, 0x3f # a0 <- and(a1, 0...0111111)
```

- OU logique ($\text{or}(100111_2, 101001_2) = 101111_2$)

```
or   $a0, $a1, $a2 # a0 <- or(a1, a2)
ori  $a0, $a1, 42  # a0 <- or(a1, 0...0101010)
```

Jeu d'instructions : décalages

- décalage à gauche (insertion de zéros)

```
sll  $a0, $a1, 2    # a0 <- a1 * 4
sllv $a1, $a2, $a3  # a1 <- a2 * 2^a3
```

- décalage à droite arithmétique (copie du bit de signe)

```
sra  $a0, $a1, 2    # a0 <- a1 / 4
```

- décalage à droite logique (insertion de zéros)

```
srl  $a0, $a1, 2
```

- rotation

```
rol  $a0, $a1, 2
ror  $a0, $a1, 3
```

- comparaison de deux registres

```
slt  $a0, $a1, $a2    # a0 <- 1 si a1 < a2  
                          #          0 sinon
```

ou d'un registre et d'une constante

```
slti $a0, $a1, 42
```

- variantes : sltu (comparaison non signée), sltiu
- de même : sle, sleu / sgt, sgtu / sge, sgeu
- égalité : seq, sne

Jeu d'instructions : transfert (lecture)

- lire un mot (32 bits) en mémoire

```
lw    $a0, 42($a1)    # a0 <- mem[a1 + 42]
```

l'adresse est donnée par un registre et un décalage sur 16 bits signés

- variantes pour lire 8 ou 16 bits, signés ou non (lb, lh, lbu, lhu)

Jeu d'instructions : transfert (écriture)

- écrire un mot (32 bits) en mémoire

```
sw    $a0, 42($a1)    # mem[a1 + 42] <- a0
                                # attention au sens !
```

l'adresse est donnée par un registre et un décalage sur 16 bits signés

- variantes pour écrire 8 ou 16 bits (sb, sh)

on distingue

- **branchement** : typiquement un saut conditionnel, dont le déplacement est stocké sur 16 bits signés (-32768 à 32767 instructions)
- **saut** : saut inconditionnel, dont l'adresse de destination est stockée sur 26 bits

- branchement conditionnel

```
beq  $a0, $a1, label # si a0 = a1 saute à label  
                        # ne fait rien sinon
```

- variantes : bne, blt, ble, bgt, bge (et comparaisons non signées)
- variantes : beqz, bnez, bgez, bgtz, bltz, blez

Jeu d'instructions : sauts

saut inconditionnel

- à une adresse (*jump*)

```
j    label
```

- avec sauvegarde de l'adresse de l'instruction suivante dans \$ra

```
jal  label    # jump and link
```

- à une adresse contenue dans un registre

```
jr   $a0
```

- avec l'adresse contenue dans \$a0 et sauvegarde dans \$a1

```
jalr $a0, $a1
```

Jeu d'instructions : appel système

quelques appels système fournis par une instruction spéciale

```
syscall
```

le code de l'instruction doit être dans \$v0, les arguments dans \$a0–\$a3 ;
le résultat éventuel sera placé dans \$v0

exemple : appel système `print_int` pour afficher un entier

```
li      $v0, 1      # code de print_int
li      $a0, 42     # valeur à afficher
syscall
```

de même `read_int`, `print_string`, etc. (voir la documentation)

on ne programme pas en langage machine mais en assembleur

l'assembleur fourni un certain nombre de facilités :

- étiquettes symboliques
- allocation de données globales
- pseudo-instructions

Assembleur MIPS

la directive

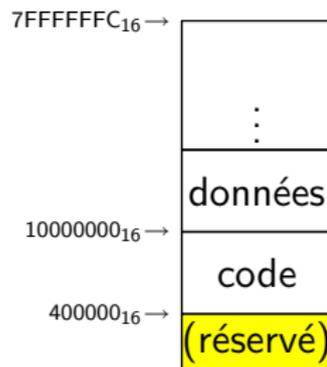
```
.text
```

indique que des instructions suivent, et la directive

```
.data
```

indique que des données suivent

le code sera chargé à partir de l'adresse $0x400000$
et les données à partir de l'adresse $0x10000000$



une étiquette symbolique est introduite par

```
label:
```

et l'adresse qu'elle représente peut être chargée dans un registre

```
la $a0, label
```

Exemple : hello world

SPIM appelle le programme à l'adresse `main`, et lui passe l'adresse où « revenir » dans `$ra`

```
.text
main:  li      $v0, 4      # code de print_string
       la      $a0, hw   # adresse de la chaîne
       syscall         # appel système
       jr      $ra      # fin du programme
       .data
hw:    .asciiz "hello world\n"
```

(`.asciiz` est une facilité pour `.byte 104, 101, ... 0`)

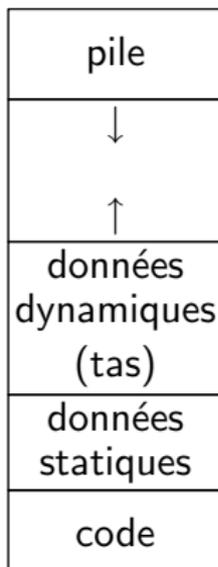
c'est de traduire un programme d'un langage de haut niveau vers ce jeu d'instructions

en particulier, il faut

- traduire les structures de contrôle (tests, boucles, exceptions, etc.)
- traduire les appels de fonctions
- traduire les structures de données complexes (tableaux, enregistrements, objets, clôtures, etc.)
- allouer de la mémoire dynamiquement

- constat** : les appels de fonctions peuvent être arbitrairement imbriqués
⇒ les registres ne peuvent suffire pour les paramètres / variables locales
⇒ il faut allouer de la mémoire pour cela

les fonctions procèdent selon un mode *last-in first-out*, c'est-à-dire de **pile**



la **pile** est stockée tout en haut, et croît dans le sens des adresses décroissantes ; `$sp` pointe sur le sommet de la pile

les données dynamiques (survivant aux appels de fonctions) sont allouées sur le **tas** (éventuellement par un GC), en bas de la zone de données, juste au dessus des données statiques

ainsi, on ne se marche pas sur les pieds

lorsqu'une fonction f (l'appelant ou *caller*) souhaite appeler une fonction g (l'appelé ou *callee*), elle exécute

```
jal g
```

et lorsque l'appelé en a terminé, il lui rend le contrôle avec

```
jr $ra
```

problème :

- si g appelle elle-même une fonction, $\$ra$ sera écrasé
- de même, tout registre utilisé par g sera perdu pour f

il existe de multiples manières de s'en sortir,
mais en général on s'accorde sur des **conventions d'appel**

utilisation des registres

- `$at`, `$k0` et `$k1` sont réservés à l'assembleur et l'OS
- `$a0–$a3` sont utilisés pour passer les quatre premiers arguments (les autres sont passés sur la pile) et `$v0–$v1` pour renvoyer le résultat
- `$t0–$t9` sont des registres **caller-saved** i.e. l'appelant doit les sauvegarder si besoin ; on y met donc typiquement des données qui n'ont pas besoin de survivre aux appels
- `$s0–$s7` sont des registres **callee-saved** i.e. l'appelé doit les sauvegarder ; on y met donc des données de durée de vie longue, ayant besoin de survivre aux appels
- `$sp` est le pointeur de pile, `$fp` le pointeur de *frame*
- `$ra` contient l'adresse de retour
- `$gp` pointe au milieu de la zone de données statiques (10008000_{16})

L'appel, en quatre temps

il y a quatre temps dans un appel de fonction

- ① pour l'appelant, juste avant l'appel
- ② pour l'appelé, au début de l'appel
- ③ pour l'appelé, à la fin de l'appel
- ④ pour l'appelant, juste après l'appel

s'organisent autour d'un segment situé au sommet de la pile appelé le **tableau d'activation**, en anglais **stack frame**, situé entre \$fp et \$sp

L'appelant, juste avant l'appel

- 1 passe les arguments dans \$a0–\$a3, les autres sur la pile s'il y en a plus de 4
- 2 sauvegarde les registres \$t0–\$t9 qu'il compte utiliser après l'appel (dans son propre tableau d'activation)
- 3 exécute

```
jal appelé
```

L'appelé, au début de l'appel

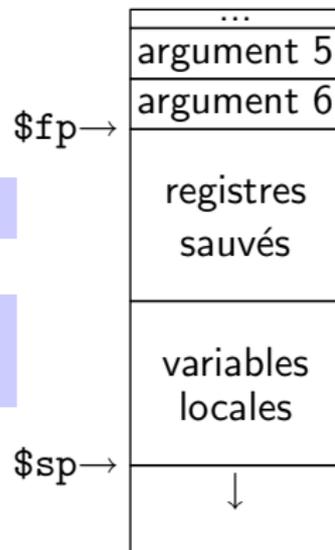
- 1 alloue son tableau d'activation, par exemple

```
addi $sp, $sp, -28
```

- 2 sauvegarde \$fp puis le positionne, par exemple

```
sw    $fp, 24($sp)
addi  $fp, $sp, 24
```

- 3 sauvegarde \$s0-\$s7 et \$ra si besoin



\$fp permet d'atteindre facilement les arguments et variables locales, avec un décalage fixe quel que soit l'état de la pile

L'appelé, à la fin de l'appel

- ① place le résultat dans \$v0 (voire \$v1)
- ② restaure les registres sauvegardés
- ③ dépile son tableau d'activation, par exemple

```
addi $sp, $sp, 28
```

- ④ exécute

```
jr $ra
```

L'appelant, juste après l'appel

- 1 dépile les éventuels arguments 5, 6, ...
- 2 restaure les registres caller-saved

exercice : programmer la fonction factorielle

- une machine fournit
 - un jeu limité d'instructions, très primitives
 - des registres efficaces, un accès coûteux à la mémoire
- la mémoire est découpée en
 - code / données statiques / tas (données dynamiques) / pile
- les appels de fonctions s'articulent autour
 - d'une notion de tableau d'activation
 - de conventions d'appel

Un exemple de compilation

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)
for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,
(c+d)/2));return f;}main(q){scanf("%d",
&q);printf("%d\n",t(~(~0<<q),0,0));}
```

Clarification

```
int t(int a, int b, int c) {
    int d=0, e=a&~b&~c, f=1;
    if (a)
        for (f=0; d=(e-=d)&-e; f+=t(a-d, (b+d)*2, (c+d)/2));
    return f;
}
```

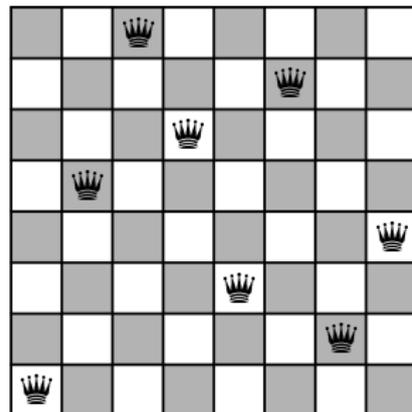
```
int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```

Clarification (suite)

```
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        int d, e=a&~b&~c;
        f = 0;
        while (d=e&-e) {
            f += t(a-d, (b+d)*2, (c+d)/2);
            e -= d;
        }
    }
    return f;
}

int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```

ce programme calcule
le nombre de solutions
du problème dit
des n reines



Comment ça marche ?

- recherche par force brute (*backtracking*)
- entiers utilisés comme des ensembles :
par ex. $13 = 0 \cdots 01101_2 = \{0, 2, 3\}$

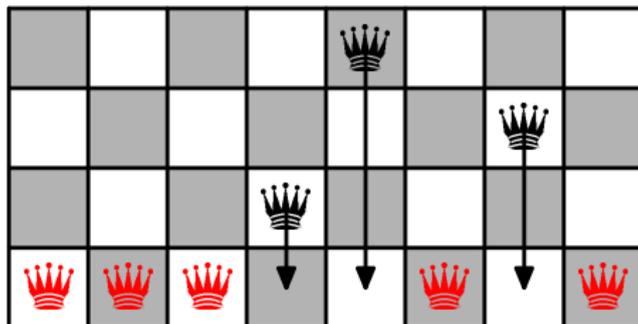
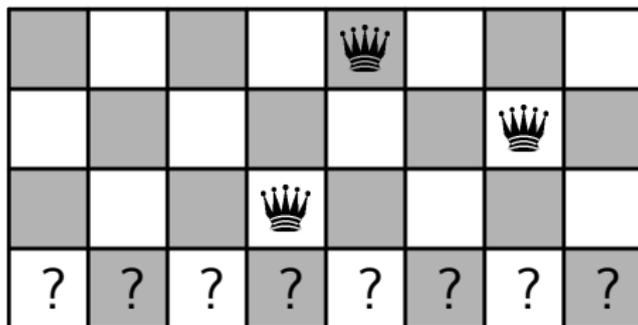
entiers	ensembles
0	\emptyset
a&b	$a \cap b$
a+b	$a \cup b$, quand $a \cap b = \emptyset$
a-b	$a \setminus b$, quand $b \subseteq a$
$\sim a$	$\complement a$
a&-a	$\min(a)$, quand $a \neq \emptyset$
$\sim(\sim 0 \ll n)$	$\{0, 1, \dots, n-1\}$
a*2	$\{i+1 \mid i \in a\}$, noté $S(a)$
a/2	$\{i-1 \mid i \in a \wedge i \neq 0\}$, noté $P(a)$

Clarification : version ensembliste

```
int t(a, b, c)
  f ← 1
  if a ≠ ∅
    e ← (a \ b) \ c
    f ← 0
    while e ≠ ∅
      d ← min(e)
      f ← f + t(a \ {d}, S(b ∪ {d}), P(c ∪ {d}))
      e ← e \ {d}
  return f

int queens(n)
  return t({0, 1, ..., n - 1}, ∅, ∅)
```

Signification de a , b et c



Intérêt de ce programme pour la compilation

```
int t(int a, int b, int c) {
    int f=1;
    if (a) {
        int d, e=a&~b&~c;
        f = 0;
        while (d=e&-e) {
            f += t(a-d, (b+d)*2, (c+d)/2);
            e -= d;
        }
    }
    return f;
}

int main() {
    int q;
    scanf("%d", &q);
    printf("%d\n", t(~(~0<<q), 0, 0));
}
```

court, mais contient

- un test (`if`)
- une boucle (`while`)
- une fonction récursive
- quelques calculs

c'est aussi une
excellente solution
au problème des n reines

commençons par la fonction récursive t ; il faut

- allouer les registres
- compiler
 - le test
 - la boucle
 - l'appel récursif
 - les différents calculs

Allocation de registres

- a, b et c sont passés dans \$a0, \$a1 et \$a2
- le résultat est renvoyé dans \$v0
- les paramètres a, b, c, tout comme les variables locales d, e, f, ont besoin d'être sauvegardés, car ils sont utilisés après l'appel récursif
⇒ on va utiliser des registres *callee-save*

a		\$s0
b		\$s1
c		\$s2
d		\$s3
e		\$s4
f		\$s5

- il faut donc allouer 7 mots sur la pile pour sauvegarder l'adresse de retour \$ra et ces 6 registres

Sauvegarde / restauration des registres

```
t:      addiu    $sp, $sp, -28      # allocation de 7 mots
        sw      $ra, 24($sp)      # sauvegarde des registres
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        sw      $s4, 4($sp)
        sw      $s5, 0($sp)
        ...
        lw      $ra, 24($sp)      # restauration des registres
        lw      $s0, 20($sp)
        lw      $s1, 16($sp)
        lw      $s2, 12($sp)
        lw      $s3, 8($sp)
        lw      $s4, 4($sp)
        lw      $s5, 0($sp)
        addiu    $sp, $sp, 28     # désallocation
        jr      $ra
```

Compilation du test

```
int t(int a, int b, int c) {  
    int f=1;  
    if (a) {  
        ...  
    }  
    return f;  
}
```

```
li    $s5, 1  
beqz  $a0, t_return  
...  
t_return:  
move  $v0, $s5
```

Cas général ($a \neq 0$)

```
if (a) {  
    int d, e=a&~b&~c;  
    f = 0;  
    while ...  
}
```

```
move    $s0, $a0 # sauvegarde  
move    $s1, $a1  
move    $s2, $a2  
move    $s4, $s0 # e=a&~b&~c  
not     $t0, $s1  
and     $s4, $s4, $t0  
not     $t0, $s2  
and     $s4, $s4, $t0  
li     $s5, 0 # f = 0
```

noter l'utilisation d'un registre temporaire \$t0 non sauvegardé

Compilation de la boucle

```
while (test) {  
    body  
}
```

```
...  
L1: ...  
    calcul de test dans $t0  
    ...  
    beqz $t0, L2  
    ...  
    body  
    ...  
    j L1  
L2: ...
```

Compilation de la boucle

il existe cependant une meilleure solution

```
while (test) {  
    body  
}
```

```
...  
j L2  
L1: ...  
    body  
    ...  
L2: ...  
    test  
    ...  
bnez $t0, L1
```

ainsi on fait seulement un seul branchement par tour de boucle
(mis à part la toute première fois)

Compilation de la boucle

```
while (d=e&-e) {  
    f += t(a-d,  
           (b+d)*2,  
           (c+d)/2);  
    e -= d;  
}
```

```
        j      test  
body:  
    sub    $a0, $s0, $s3 # a-d  
    add    $a1, $s1, $s3 # (b+d)*2  
    sll    $a1, $a1, 1  
    add    $a2, $s2, $s3 # (c+d)/2  
    srl    $a2, $a2, 1  
    jal    t  
    add    $s5, $s5, $v0  
    sub    $s4, $s4, $s3 # e -= d  
test:  
    neg    $t0, $s4      # d=e&-e  
    and    $s3, $s4, $t0  
    bnez   $s3, body  
t_return:  
    ...
```

Programme principal

```
int main() {  
    int q;  
    scanf("%d", &q);  
    printf("%d\n",  
           t(~(~0<<q), 0, 0));  
}
```

main:

```
move $s6, $ra # sauvegarde de $ra  
li   $v0, 5   # read_int  
syscall  
li   $a0, 0   # t(...)  
not  $a0, $a0  
sll  $a0, $a0, $v0  
not  $a0, $a0  
li   $a1, 0  
li   $a2, 0  
jal  t  
move $a0, $v0 # printf  
li   $v0, 1  
syscall  
li   $v0, 4  
la   $a0, newline  
syscall  
move $ra, $s6 # on restaure $ra  
jr   $ra
```

ce code n'est pas optimal

par exemple, dans le cas $a = 0$, on sauve/restaure inutilement les registres (y compris $\$ra$), alors qu'on pourrait se contenter de

```
li $v0, 1
jr $ra
```

- produire du code assembleur efficace n'est pas chose aisée (observer le code produit par gcc grâce aux options `-S` et `-fverbose-asm`)
- maintenant il va falloir automatiser tout ce processus

- TD de mercredi
 - génération de code pour un mini-langage d'expressions arithmétiques
- Cours de jeudi
 - Syntaxe abstraite
 - Sémantique
 - Interprètes