



LLVM

Option INFO — TLANG

Pierre Molinaro

Novembre 2019

LLVM : Low Level Virtual Machine

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

Les deux principaux sous-projets sont :

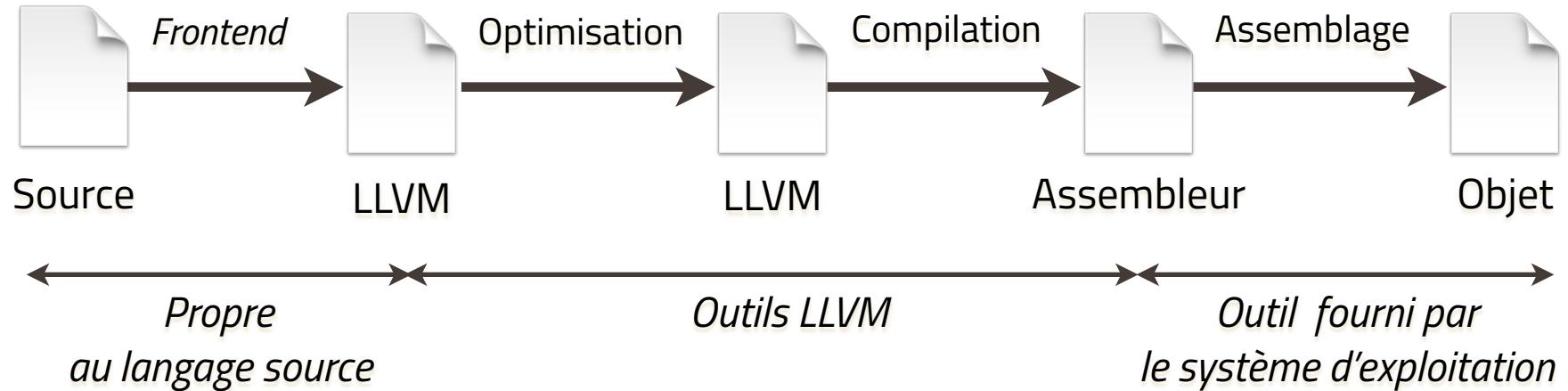
- *The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs ;*
- *Clang is an 'LLVM native' C/C++/Objective-C compiler.*

Le but de cette partie du cours TLANG est de présenter les composants de LLVM et vous faire écrire des programmes en LLVM.

Lien :

<http://llvm.org>

Position de LLVM

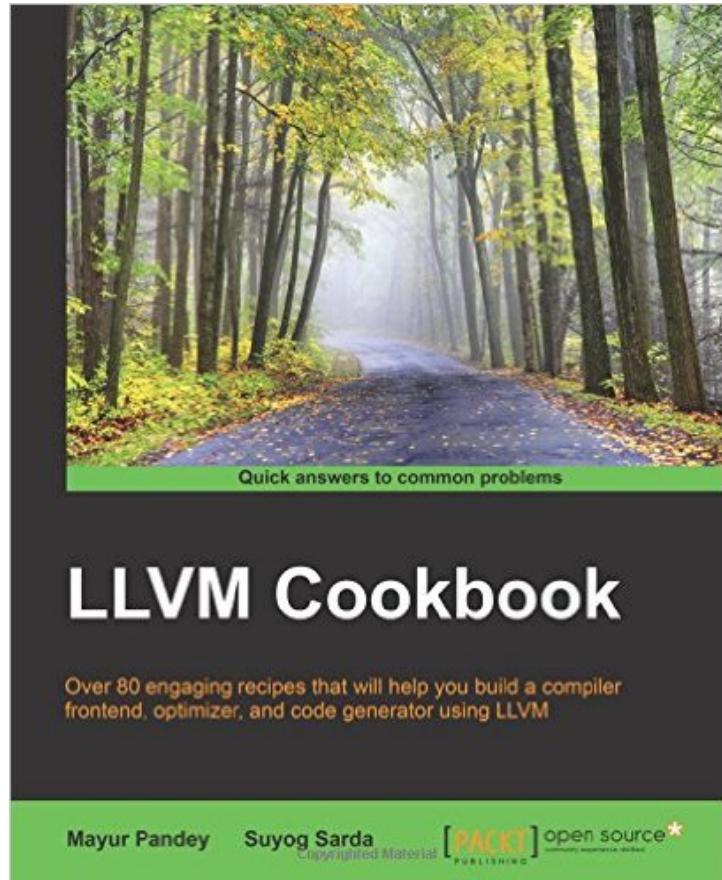


Le code LLVM peut apparaître sous trois formes :

- *en mémoire* : manipulation de structures de données en C++ ;
- fichier binaire dit *bitcode* (« *bitcode* », et non pas « *bytecode* ») ;
- fichier textuel dit *assembleur LLVM* (objet de la partie 3 de ce poly).

Ces trois formes sont équivalentes : le projet LLVM propose des outils permettant de passer de l'une à l'autre sans perte.

Des livres...



1 — Installer CLANG et LLVM

CLANG — LLVM

CLANG est un *frontend* pour les langages C, C++, Objective C.
LLVM est le *backend*.

Actuellement, un assembleur et un éditeur de liens externes sont nécessaires : on utilise habituellement `as` et `ld` de *binutils*.

L'installation dépend de l'utilisation que l'on veut faire :

- en chaîne de développement native ;
- en chaîne de développement croisée (par exemple : pour ARM).

Installation en chaîne de développement native

<http://llvm.org/releases/download.html>

Attention : la représentation textuelle LLVM dépend de la version ; ce poly est rédigé avec des exemples pour une version $\geq 3.6.x$ et $< 9.x$

Windows, Mac, Linux : la solution conseillée est d'installer une distribution binaire (voir page suivante). La chaîne de développement native a un support très limité de la cross-compilation.

Mac : si vous avez installé Xcode, vous disposez de CLANG / LLVM comme compilateur natif, mais attention, c'est une version propre à Apple. Pour être conforme à ce poly, il vaut mieux installer une distribution binaire.

Linux : sous Ubuntu, CLANG / LLVM est disponible sous la forme de paquetages à installer (vérifier leurs versions).

Installation d'une distribution binaire

<http://llvm.org/releases/download.html>

Download LLVM 3.6.2

Sources:

- [Clang source code \(.sig\)](#)
- [LLVM source code \(.sig\)](#)
- [Compiler RT source code \(.sig\)](#)
- [DragonEgg source code \(.sig\)](#)
- [LibC++ source code \(.sig\)](#)
- [LibC++ ABI source code \(.sig\)](#)
- [LLDB source code \(.sig\)](#)
- [LLD source code \(.sig\)](#)
- [Polly source code \(.sig\)](#)
- [OpenMP source code \(.sig\)](#)
- [Clang Tools Extra \(.sig\)](#)
- [LLVM Test Suite \(.sig\)](#)

Pre-built Binaries:

- [Clang for Windows \(.sig\)](#)
- [Clang for AArch64 Linux \(.sig\)](#)
- [Clang for armv7a Linux \(.sig\)](#)
- [Clang for Mac OS X \(.sig\)](#)
- [Clang for FreeBSD10 AMD64 \(.sig\)](#)
- [Clang for FreeBSD10 i386 \(.sig\)](#)
- [Clang for OpenSuSE 13.2 i586 Linux \(.sig\)](#)
- [Clang for OpenSuSE 13.2 x86_64 Linux \(.sig\)](#)
- [Clang for Fedora21 i386 Linux \(.sig\)](#)
- [Clang for Fedora21 x86_64 Linux \(.sig\)](#)
- [Clang for Ubuntu 14.04 \(.sig\)](#)
- [Clang for Ubuntu 15.04 \(.sig\)](#)
- [Clang for MIPS \(.sig\)](#)
- [Clang for MIPSel \(.sig\)](#)

Mac : Utiliser la distribution binaire (voir ci-contre), c'est une archive. Une fois celle-ci décompressée, vous pouvez déplacer le répertoire obtenu vers votre répertoire local ~ (voir [page 10](#)).

Linux : vous pouvez soit utiliser la distribution binaire (comme pour Mac), soit télécharger le paquet correspondant si il existe (voir [page 11](#)).

Windows : ?

Essai de la chaîne de développement native

Compilation, exécution

exemple.c

```
#include <stdio.h>

int main (void) {
    printf ("Hello !\n") ;
    return 0 ;
}
```

clang est conçu pour être un remplaçant de gcc : les options de clang sont pour la plupart les mêmes que celle de gcc.

Pour gagner du temps, vous pouvez écrire un shell script ou un script Python qui contient les bonnes options.

Il faut effectuer la compilation et l'exécution pour vérifier que la chaîne de développement native fonctionne correctement.

Essai de la chaîne de développement native sous Mac OS X

Sous Mac OS X, vous devez installer Xcode (à partir de l'*Apple Store*), clang a besoin du Mac OS X SDK.

Pour le compilateur appelé soit bien le 3.6.2 que vous avez téléchargé, et non pas celui de Xcode :

- soit vous lappelez avec un chemin absolu (/monchemin/bin/clang) ;
- soit vous modifiez la variable \$PATH, par exemple dans un shell script :

```
export PATH=/monchemin/bin:$PATH
```

Sur Mac, si vous appelez :

```
clang /xyz/exemple.c -o /xyz/exemple
```

Vous aurez l'erreur :

```
/Users/pierremolinaro/Desktop/appel-clang/exemple.c:1:10: fatal error: 'stdio.h' file not found
#include <stdio.h>
^
1 error generated.
```

Sous Mac OS X, il faut préciser l'emplacement du Mac OS X SDK par l'option -sysroot :

```
clang -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
Developer/SDKs/MacOSX10.11.sdk /xyz/exemple.c -o /xyz/exemple
```

Cette ligne doit s'exécuter sans erreur et produire l'exécutable /xyz/exemple

Essai de la chaîne de développement native sous Ubuntu

Sous Ubuntu (32 ou 64 bits), clang / llvm sont définis dans des paquetages que vous pouvez installer :

- **clang-3.6** (installer ce paquet installe aussi le suivant)
- **llvm-3.6**

Vérifier l'installation correcte :

```
clang-3.6 -v
```

La réponse doit être :

```
Ubuntu clang version 3.6.0-2ubuntu1~trusty ...
```

Vous pouvez ensuite effectuer la compilation de l'exemple :

```
clang-3.6 /xyz/exemple.c -o /xyz/exemple
```

Cette ligne doit s'exécuter sans erreur et produire l'exécutable /xyz/exemple

Essai de la chaîne de développement native

Obtention du code assembleur

exemple.c

```
#include <stdio.h>

int main (void) {
    printf ("Hello !\n");
    return 0 ;
}
```

clang -S exemple.c -o exemple.s

Note : le code assembleur obtenu dépend évidemment du processeur et du système d'exploitation.

exemple.s

```
.section __TEXT,__text,regular,pure_instructions
.macosx_version_min 10, 10
.globl _main
.align 4, 0x90
_main:                                ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq $16, %rsp
    leaq L_.str(%rip), %rdi
    movl $0, -4(%rbp)
    movb $0, %al
    callq _printf
    xorl %ecx, %ecx
    movl %eax, -8(%rbp)      ## 4-byte Spill
    movl %ecx, %eax
    addq $16, %rsp
    popq %rbp
    retq
    .cfi_endproc

.section __TEXT,__cstring,cstring_literals
L_.str:                                ## @.str
    .asciz "Hello !\n"

.subsections_via_symbols
```

Essai de la chaîne de développement native

Obtention de la représentation textuelle du code LLVM

exemple.c

```
#include <stdio.h>

int main (void) {
    printf ("Hello !\n");
    return 0 ;
}
```

Notes :

- le code LLVM obtenu dépend évidemment du processeur et du système d'exploitation ;
- aucun commentaire dans cette page sur le code obtenu ; la description détaillée du langage textuel LLVM est l'objet de la suite de ce poly.

clang -emit-llvm -S exemple.c -o exemple.ll

```
; ModuleID = 'exemple.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.10.0"

@.str = private unnamed_addr constant [9 x i8] c"Hello !\0A\00", align 1

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    %2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([9 x i8]* @.str, i32 0, i32 0))
    ret i32 0
}

declare i32 @printf(i8*, ...)

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

exemple.ll

L'exemple en natif (3/3)

Un exemple (avec optimisation du code LLVM engendré)

addition.c

```
int addition (int a, int b) {  
    return a+b ;  
}
```

Exemple de compilation native *optimisée* pour processeur core-avx2.

clang -emit-llvm -S **-O3** addition.c -o addition.ll

```
; ModuleID = 'addition.c'  
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"  
target triple = "x86_64-apple-macosx10.10.0"  
  
; Function Attrs: nounwind readnone ssp uwtable  
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}  
  
attributes #0 = { nounwind readnone ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"  
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  
  
!llvm.module.flags = !{!0}  
!llvm.ident = !{!1}  
  
!0 = !{i32 1, !"PIC Level", i32 2}  
!1 = !{!"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

addition.ll

2 — Utilisation en ligne de commandes de CLANG et LLVM

Les principaux utilitaires en ligne de commande

<http://llvm.org/docs/CommandGuide/>

Appel : utilitaire fichiers-d-entrée -o fichier-de-sortie

L'option « -o » désigne le fichier de sortie. Un fichier *assembleur LLVM* a pour extension « .ll ».

Un fichier *bitcode* a pour extension « .bc ».

Utilitaire	Rôle	Fichier(s) d'entrée	Fichier de sortie
llvm-as	Assemblage LLVM	<i>Assembleur LLVM</i>	<i>bitcode</i>
llvm-dis	Désassemblage LLVM	<i>bitcode</i>	<i>assembleur LLVM</i>
llvm-link	Édition de liens LLVM	Plusieurs <i>assembleur LLVM</i>	<i>bitcode</i> <i>assembleur LLVM</i> (option -S)
opt	Optimisation LLVM	<i>Assembleur LLVM</i> ou <i>bitcode</i>	<i>bitcode</i> <i>assembleur LLVM</i> (option -S)
llvm-lli	Interpréteur LLVM	Un ou plusieurs <i>assembleur LLVM</i> ou <i>bitcode</i>	—
llc	Compilation LLVM	<i>Assembleur LLVM</i> ou <i>bitcode</i>	<i>Assembleur</i> (extension « .s »)

Les fichiers sources de l'exemple

proc.c

```
#include <stdio.h>
#include "proc.h"

void routine (void) {
    printf ("Hello !\n");
}
```

proc.h

```
void routine (void) ;
```

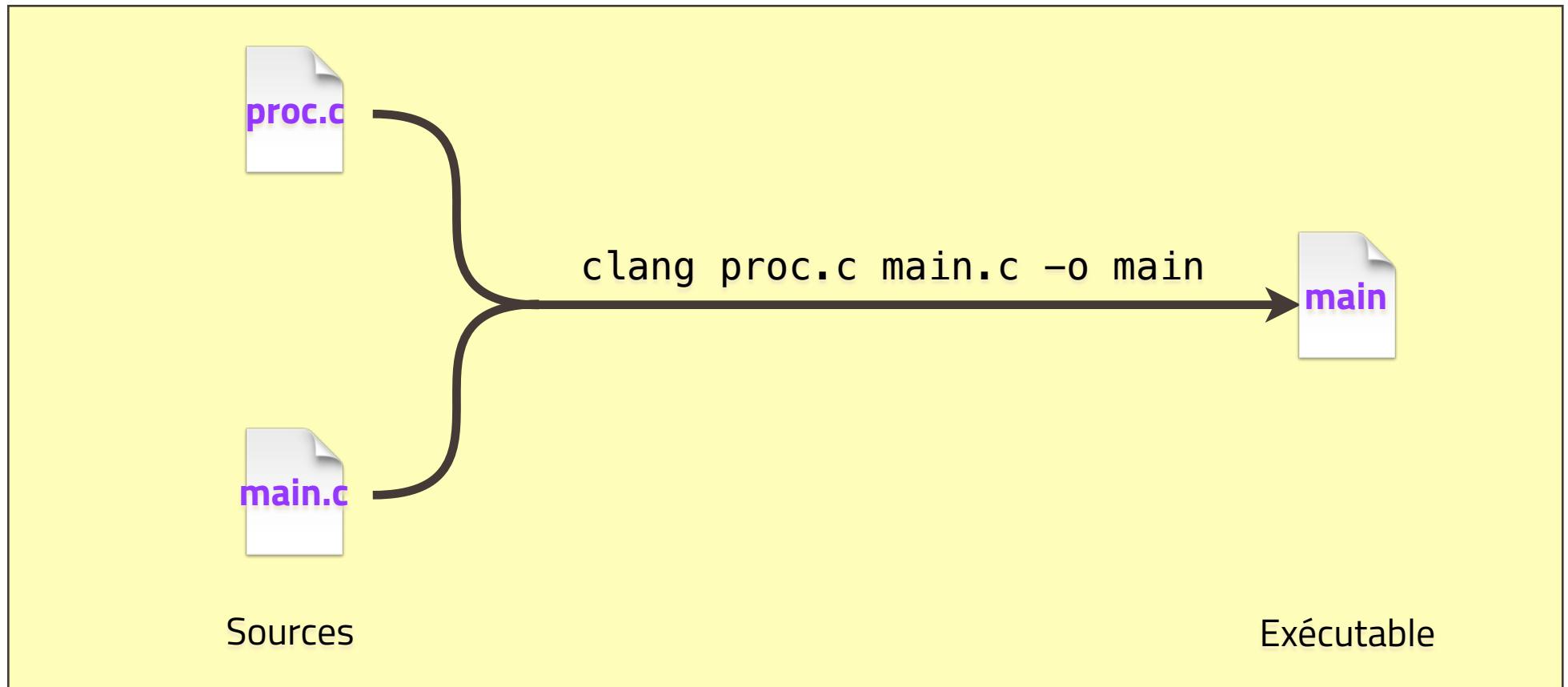
main.c

```
#include "proc.h"

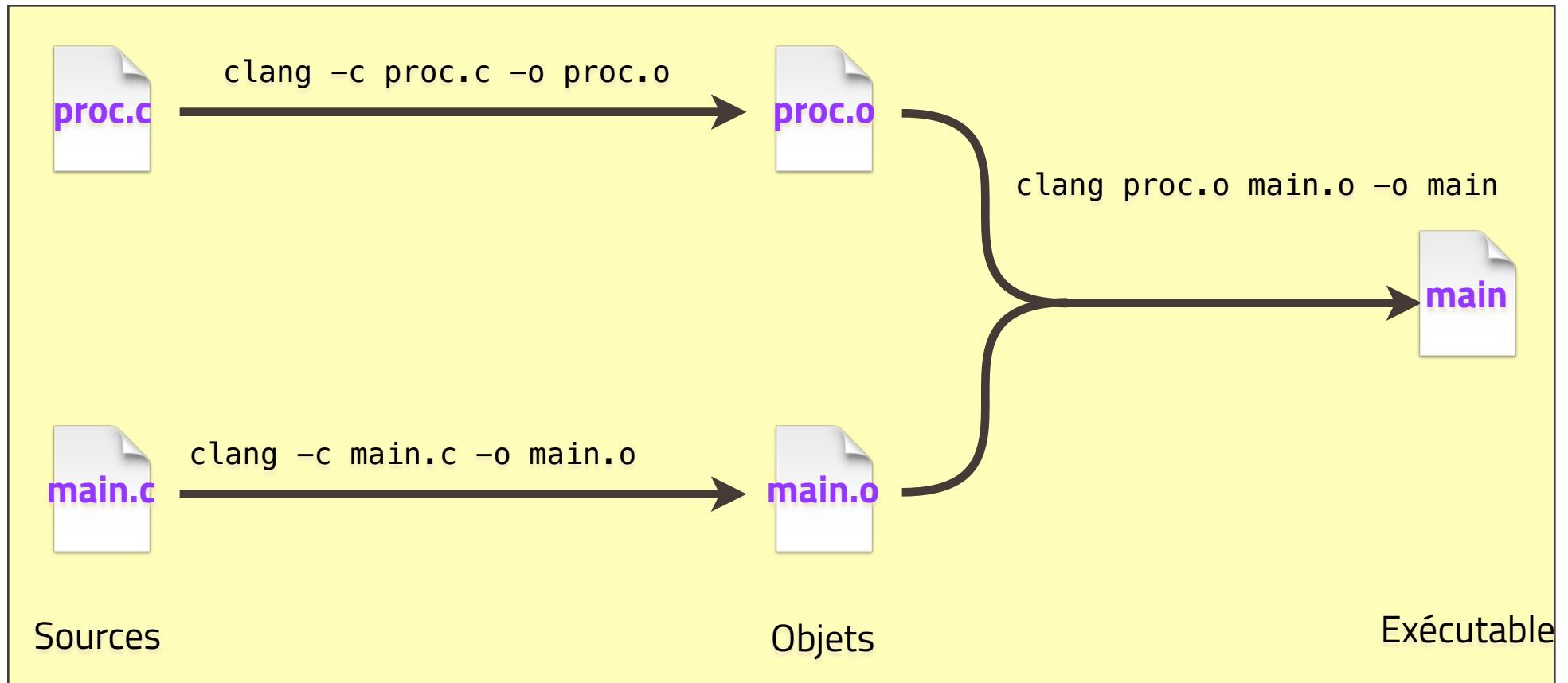
int main (void) {
    routine () ;
    return 0 ;
}
```

Nous allons voir différentes façons d'utiliser la chaîne native de compilation LLVM / CLANG.

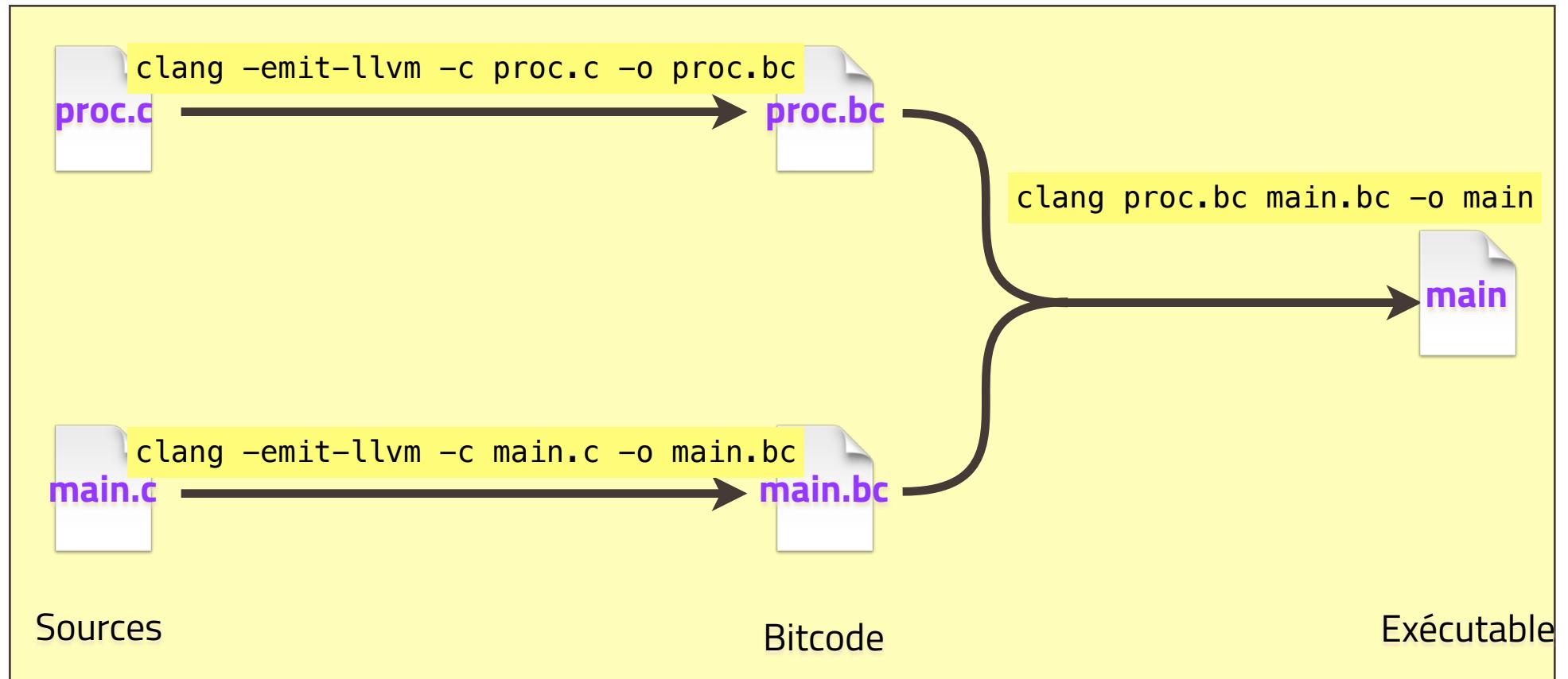
Compilation et édition de liens en une seule commande



Compilation séparée

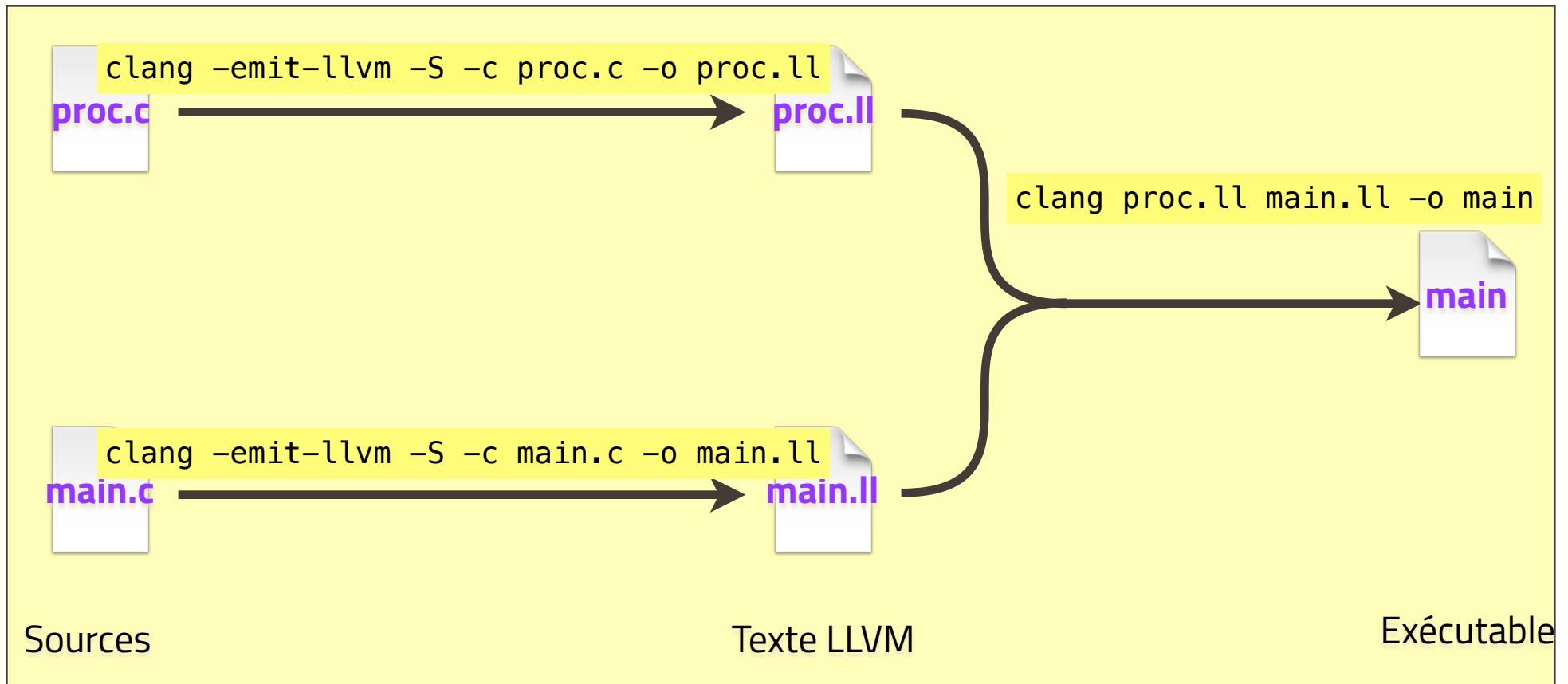


Compilation séparée, fichiers *bitcode*

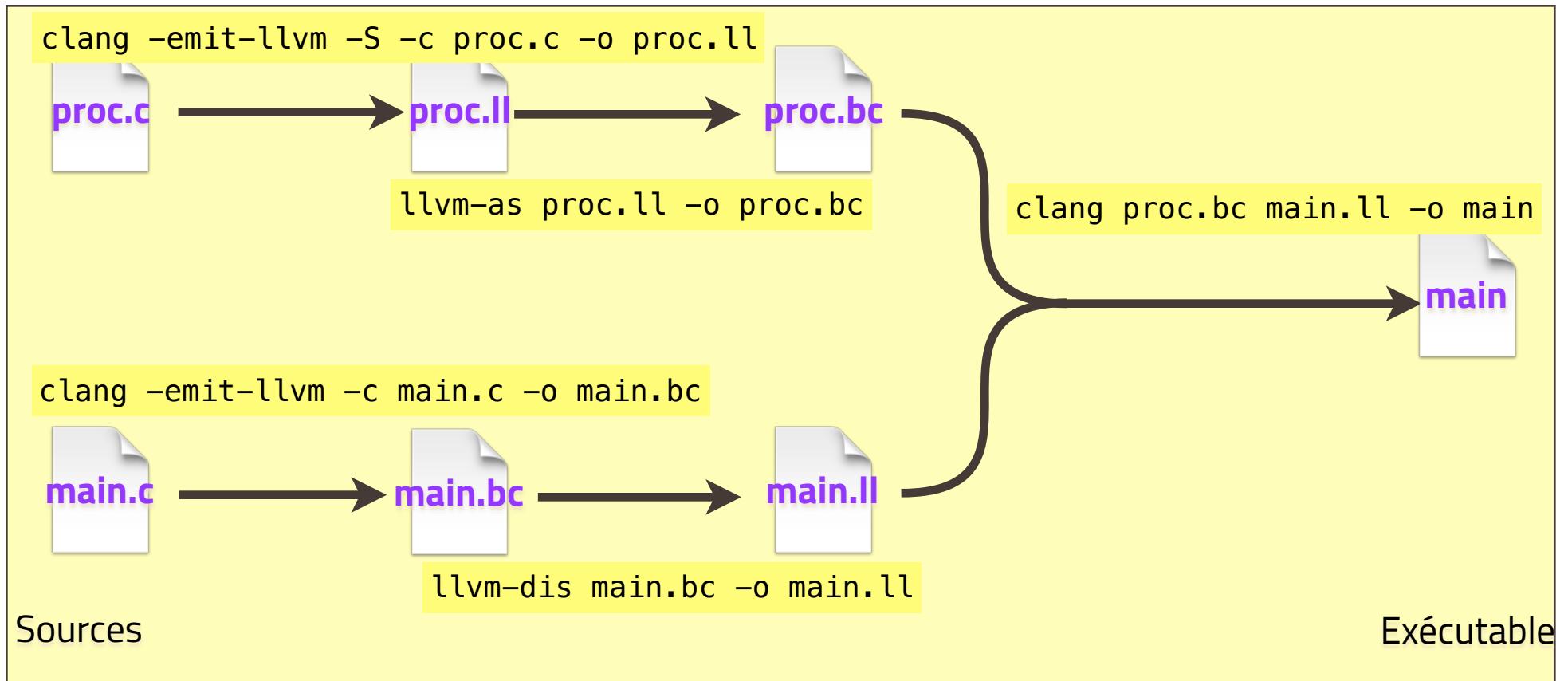


Un fichier *bitcode* (extension « .bc ») contient une représentation binaire d'un code LLVM.

Compilation séparée, fichiers *assembleur LLVM*



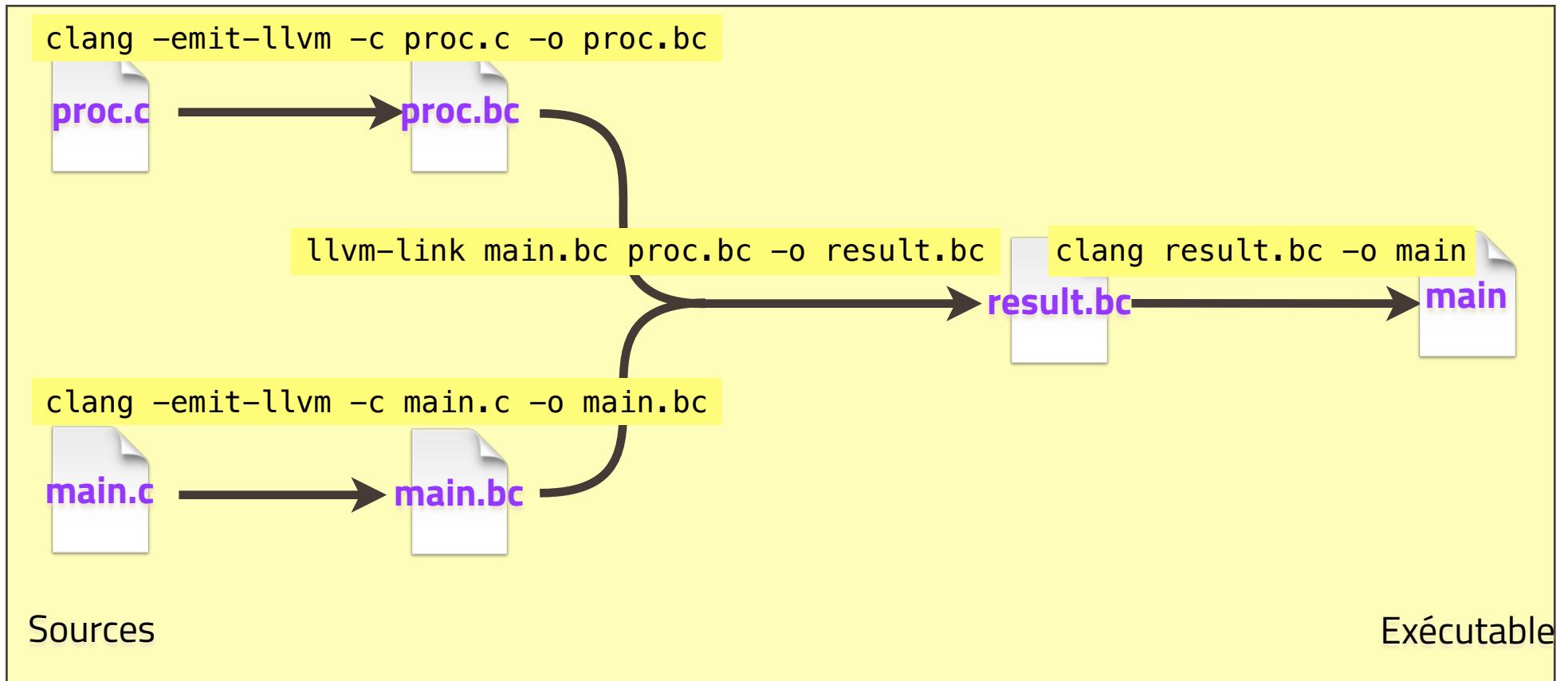
Compilation séparée, assemblage / désassemblage



Fichier *assembleur LLVM* et fichier *bitcode* sont deux représentations de la même information. On peut passer de l'une à l'autre sans perte par :

- `llvm-dis` : bitcode -> assembleur LLVM ;
- `llvm-as` : assembleur LLVM -> bitcode.

Compilation séparée, édition de liens LLVM



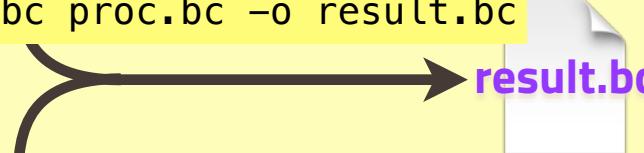
L'utilitaire `llvm-link` lie ensemble des fichiers *bitcode* et / ou des fichiers *assem-bleur LLVM*. Le fichier de sortie est un fichier *bitcode* par défaut, ou assebleur LLVM si l'option « `-S` » est présente.

Compilation séparée, édition de liens LLVM, exécution à la volée

```
clang -emit-llvm -c proc.c -o proc.bc
```



```
llvm-link main.bc proc.bc -o result.bc
```



```
clang -emit-llvm -c main.c -o main.bc
```



Sources

Exécution à la volée
llvm-lli result.bc

L'utilitaire `llvm-lli` effectue l'édition de liens à la volée, puis exécute le code obtenu.

Compilation vers fichier assembleur natif

L'utilitaire `llc` traduit un fichier *bitcode* ou *assembleur LLVM* en assembleur.

```
clang -emit-llvm -c proc.c -o proc.bc
```



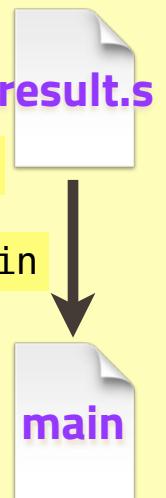
```
llvm-link main.bc proc.bc -o result.bc
```

```
clang -emit-llvm -c main.c -o main.bc
```



```
llc result.bc -o result.s
```

```
clang result.s -o main
```



Optimisations

Utilitaires acceptant des options d'optimisation

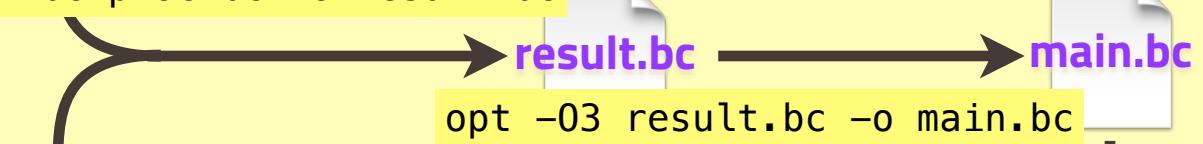
Utilitaire	Options d'optimisation	Rôle
opt	« -01 », « -02 », « -03 »	Optimisation LLVM
llc	« -01 », « -02 », « -03 »	Optimisation de la traduction en assembleur
llvm-lli	« -01 », « -02 », « -03 »	Applique les options d'optimisations aux différentes étapes de la compilation
clang	« -01 », « -02 », « -03 »	

Exemple de mise en place des optimisations

```
clang -emit-llvm -c -O3 proc.c -o proc.bc
```



```
llvm-link main.bc proc.bc -o result.bc
```



```
clang -emit-llvm -c -O3 main.c -o main.bc
```



```
opt -O3 result.bc -o main.bc
```

```
llc -O3 main.bc -o main.s
```



```
clang main.s -o main
```



3 — Assembleur LLVM

Introduction

Le but de cette partie est de vous faire écrire du code assembleur LLVM.

Comment s'y prendre :

- LA référence : <http://llvm.org/docs/LangRef.html> ;
- une façon simple d'aborder l'assembleur LLVM est de compiler des exemples simples écrits en C avec `clang` ; ne pas utiliser d'option d'optimisation pour obtenir un code issu d'une traduction élémentaire instruction par instruction ;
- pour écrire du code LLVM, SURTOUT ne pas essayer d'écrire du code optimisé ! L'optimiseur fera le travail d'optimisation bien mieux que nous.

Un premier exemple simple

addition.c

```
int addition (int a, int b) {  
    const int r = a+b ;  
    return r ;  
}
```

Exemple de compilation native optimisée.
Le résultat dépend du processeur et du système d'exploitation.

```
clang -emit-llvm -S -O3 addition.c -o addition.ll
```

```
; ModuleID = 'addition.c'  
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"  
target triple = "x86_64-apple-macosx10.10.0"  
  
; Function Attrs: nounwind readnone ssp uwtable  
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}  
  
attributes #0 = { nounwind readnone ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"  
"no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  
  
!llvm.module.flags = !{!0}  
!llvm.ident = !{!1}  
  
!0 = !{i32 1, !"PIC Level", i32 2}  
!1 = !{!"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

addition.ll

Déclarations target : spécification de la cible

Les déclarations **target** désignent le processeur et le système d'exploitation pour lequel le code est compilé.

- **target datalayout** : <http://llvm.org/docs/LangRef.html#data-layout>
- **target triple** : <http://llvm.org/docs/LangRef.html#target-triple>

```
clang -emit-llvm -S -O3 addition.c -o addition.ll
```

```
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"  
target triple = "x86_64-apple-macosx10.10.0"
```

```
clang --target=armv7-none--eabi -mcpu=cortex-m4 -emit-llvm -S -O3 addition.c -o addition.ll
```

```
target datalayout = "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64"  
target triple = "thumbv7em-none--eabi"
```

Les déclarations **target** sont optionnelles ; si elles sont absentes, alors il faut que la cible soit explicitement précisée (par exemple « `--target=armv7-none--eabi -mcpu=cortex-m4` ») parmi les options de la ligne de commande de chaque outil ; si elles sont présentes, elles permettent à chaque outil de connaître la cible.

Conseil : toujours placer ces déclarations dans vos fichiers LLVM.

Syntaxe assembleur LLVM

```
; ModuleID = 'addition.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.10.0"                                addition.ll

; Function Attrs: nounwind readnone ssp uwtable
define i32 @addition(i32 %a, i32 %b) #0 {
    %1 = add nsw i32 %b, %a
    ret i32 %1
}

attributes #0 = { nounwind readnone ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
    "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

Commentaire : commence par « ; » et s'étend jusqu'à la fin de la ligne courante.

Symboles :

- @xxx : symboles globaux ;
- %xxx : symboles internes à une fonction ;
- #xxx : attributs d'une fonction ;
- !xxx : metadatas (<http://llvm.org/docs/LangRef.html#metadata>) permettant le débogage au niveau source (dans cette présentation, les metadatas ne sont pas abordées).

Dans un symbole xxx, les lettres, chiffres, point et caractère de soulignement sont acceptés.

Cependant, un symbole ne comportant que des chiffres obéit à une règle particulière : pour que « %N » soit accepté, il faut que « %0 », « %1 », ..., « %N-1 » soient définis.

Types LLVM

<http://llvm.org/docs/LangRef.html#single-value-types>

LLVM prend en charge les types suivants :

- entiers *signés* et *non signés* : « `iN` », où N est un entier entre 1 et $2^{23}-1$;
- flottants : « `float` », « `double` », ...
- pointeur vers un type : suffixe « `type*` » ;
- structure : « `{type, type, ...}` » ;
- structure compressée : « `<{type, type, ...}>` » ;
- tableau : « `[N x type]` », où N est un entier ;
- vecteur : « `<N x type>` », où N est un entier.

Notes :

- le type booléen est « `i1` » ;
- entiers *signés* et *non signés* ne sont pas distingués ;
- les constantes entières **doivent** être écrites en décimal ;
- la notation `0x...` est réservée pour représenter les flottants.

Déclaration d'une fonction

<http://llvm.org/docs/LangRef.html#functions>

```
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}
```

```
define type-retour @nomFonction (arguments) attributs {  
liste-de-blocs  
}
```

Type de retour : si la fonction ne renvoie aucune valeur, indiquer « void ».

Arguments : séquence de « type %symbole », séparés par des virgules « , ». Vide si aucun argument.

Attributs d'une fonction

<http://llvm.org/docs/LangRef.html#fnattrs>

```
define i32 @addition(i32 %a, i32 %b) #0 {  
...  
}  
  
attributes #0 = { nounwind readnone ... }
```

La déclaration attributes déclare le symbole #xxx et lui associe une liste d'attributs de fonction. Le symbole #xxx peut être utilisé à la place d'une liste explicite d'attributs.

Principaux attributs :

- nounwind : la fonction n'engendre pas d'exception ;
- readnone : aucun accès en lecture à la mémoire ;
- readonly : aucun accès en écriture à la mémoire.

Code d'une fonction

<http://llvm.org/docs/LangRef.html#fnattrs>

```
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}
```

Le code d'une fonction est une séquence non vide de blocs. Le premier bloc qui apparaît est celui qui est exécuté quand la fonction est appelée.

Un bloc est constitué :

- d'une étiquette qui définit son point d'entrée ;
- suivi de zéro, une ou plusieurs instructions « phi » ;
- suivi de zéro, une ou plusieurs instructions séquentielles ;
- suivi d'un terminateur.

La structuration en bloc est présentée dans la partie 4.

Il n'est pas possible de définir une étiquette au milieu d'un bloc, parmi ses instructions.

Mais où apparaît l'étiquette qui définit le point d'entrée dans la fonction ci-dessus ?

Étiquettes implicites et symboles « %N »

```
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}
```

Un symbole « %N », où N est un entier, est très particulier : pour avoir le droit de l'utiliser, il faut que les symboles « %0 », « %1 », ..., « %N-1 » soient aussi utilisés.

Si un bloc n'a pas d'étiquette explicite, une étiquette « %N » lui est attribuée implicitement.

La fonction ci-dessus est donc équivalente à :

```
define i32 @addition(i32 %a, i32 %b) #0 {  
    0: ; pas de % quand on définit l'étiquette  
    %1 = add nsw i32 %b, %a ; Comme %0 est défini, on peut utiliser %1  
    ret i32 %1  
}
```

Conseil : ne pas utiliser de symbole « %N », mais un identificateur commençant par une lettre.

```
define i32 @addition(i32 %a, i32 %b) #0 {  
    entry:  
    %result = add nsw i32 %b, %a  
    ret i32 %result  
}
```

type 32

L'addition entre entiers

<http://llvm.org/docs/LangRef.html#binary-operations>

Les quatre formes de l'addition sont :

- <result> = add <ty> <op1>, <op2> ; yields ty:result
- <result> = add nuw <ty> <op1>, <op2> ; yields ty:result
- <result> = add nsw <ty> <op1>, <op2> ; yields ty:result
- <result> = add nuw nsw <ty> <op1>, <op2> ; yields ty:result

Indiquer le type « <ty> » est obligatoire. Les opérandes peuvent être :

- des symboles du type « <ty> » (pas de conversion implicite de type) ;
- des constantes entières (écrites en décimal).

Le résultat est modulo 2^N . Comme la représentation interne est le *complément à deux*, la même instruction s'applique aux entiers *signés* et *non signés*.

Note : *nuw and nsw stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the nuw and/or nsw keywords are present, the result value of the add is a poison value if unsigned and/or signed overflow, respectively, occurs.*

Le code engendré pourra engendrer une erreur d'exécution si un débordement survient.

Autres opérations arithmétiques sur les entiers

<http://llvm.org/docs/LangRef.html#binary-operations>

Opération	Signification
<result> = sub <ty> <op1>, <op2> <result> = sub nuw <ty> <op1>, <op2> <result> = sub nsw <ty> <op1>, <op2> <result> = sub nuw nsw <ty> <op1>, <op2>	Soustraction signée ou non signée <op1> - <op2>
<result> = mul <ty> <op1>, <op2> <result> = mul nuw <ty> <op1>, <op2> <result> = mul nsw <ty> <op1>, <op2> <result> = mul nuw nsw <ty> <op1>, <op2>	Multiplication signée ou non signée <op1> * <op2>
<result> = udiv <ty> <op1>, <op2> <result> = udiv exact <ty> <op1>, <op2>	Division non signée <op1> / <op2>
<result> = sdiv <ty> <op1>, <op2> <result> = sdiv exact <ty> <op1>, <op2>	Division signée <op1> / <op2>
<result> = urem <ty> <op1>, <op2>	Reste de la division non signée <op1> / <op2>
<result> = srem <ty> <op1>, <op2>	Reste de la division signée <op1> / <op2>

Note : If the exact keyword is present, the result value of the udiv is a poison value if %op1 is not a multiple of %op2 (as such, “((a udiv exact b) mul b) == a”).

Opérations logiques sur les entiers

<http://llvm.org/docs/LangRef.html#binary-operations>

Opération	Résultat
<code><result> = shl <ty> <op1>, <op2></code> <code><result> = shl nuw <ty> <op1>, <op2></code> <code><result> = shl nsw <ty> <op1>, <op2></code> <code><result> = shl nuw nsw <ty> <op1>, <op2></code>	<code><op1></code> décalé à gauche de <code><op2></code> bits (insertion de 0) ; <code><op2></code> est considéré comme un entier non signé.
<code><result> = lshr <ty> <op1>, <op2></code> <code><result> = lshr exact <ty> <op1>, <op2></code>	<code><op1></code> décalé à droite de <code><op2></code> bits (insertion de 0) ; <code><op2></code> est considéré comme un entier non signé.
<code><result> = ash <ty> <op1>, <op2></code> <code><result> = ash exact <ty> <op1>, <op2></code>	<code><op1></code> décalé à droite de <code><op2></code> bits (insertion du bit signe) ; <code><op2></code> est considéré comme un entier non signé.
<code><result> = and <ty> <op1>, <op2></code>	Et logique bit à bit
<code><result> = or <ty> <op1>, <op2></code>	Ou logique bit à bit
<code><result> = xor <ty> <op1>, <op2></code>	Ou exclusif bit à bit

Note : If the exact keyword is present, the result value of the udiv is a poison value if %op1 is not a multiple of %op2 (as such, “((a udiv exact b) mul b) == a”).

Les principaux terminateurs

<http://llvm.org/docs/LangRef.html#terminator-instructions>

Un terminator exprime toujours une rupture de l'exécution séquentielle.

Terminateur	Signification
ret <type> <valeur>	Retour à l'appelant, avec valeur associée
ret void	Retour à l'appelant, sans valeur associée
br label %<label>	Branchement inconditionnel
br i1 <valeur>, label %<iftrue>, label %<iffalse>	Branchement conditionnel

Exemples :

- br label %suite
- br i1 %condition, label %vrai, label %false

Le premier exemple, sans optimisation

addition.c

```
int addition (int a, int b) {  
    const int r = a + b ;  
    return r ;  
}
```

Exemple de compilation native sans optimisation.

Le résultat dépend du processeur et du système d'exploitation.

```
clang -emit-llvm -S addition.c -o addition.ll
```

```
; ModuleID = 'addition.c'  
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"  
target triple = "x86_64-apple-macosx10.11.0"  
  
; Function Attrs: nounwind ssp uwtable  
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    store i32 %a, i32* %1, align 4  
    store i32 %b, i32* %2, align 4  
    %3 = load i32* %1, align 4  
    %4 = load i32* %2, align 4  
    %5 = add nsw i32 %3, %4  
    store i32 %5, i32* %r, align 4  
    %6 = load i32* %r, align 4  
    ret i32 %6  
}  
  
attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  
  
!llvm.module.flags = !{!0}  
!llvm.ident = !{!1}  
  
!0 = !{i32 1, !"PIC Level", i32 2}  
!1 = !{!"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

addition.ll

Le premier exemple, sans optimisation : discussion

Le code engendré sans optimisation est fidèle à la sémantique du C : toute donnée réside en mémoire.

Chaque instruction est traduite en LLVM de façon basique.

Instruction alloca : <http://llvm.org/docs/LangRef.html#alloca-instruction>

```
; ModuleID = 'addition.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.11.0"                                addition.ll

; Function Attrs: nounwind ssp uwtable
define i32 @addition(i32 %a, i32 %b) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    %3 = load i32* %1, align 4
    %4 = load i32* %2, align 4
    %5 = add nsw i32 %3, %4
    store i32 %5, i32* %r, align 4
    %6 = load i32* %r, align 4
    ret i32 %6
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{!"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

Les arguments sont placés en mémoire

Instruction const int r = a + b ;

Instruction return r ;

Exercice

Modifier la fonction précédente pour renvoyer $a+b+1$.

plus facile avec l'optimisation
-o3

4 — La structure en blocs

Exercice

Compiler en LLVM **sans optimisation** le code suivant :

```
uint32_t somme (uint32_t n) {  
    uint32_t r = 0 ;  
    while (n > 0) {  
        r += n ;  
        n-- ;  
    }  
    return r ;  
}
```

Le code LLVM obtenu met en évidence la structuration en blocs.

Sans optimisation, parce que l'optimisation fait apparaître l'instruction phi, objet de la partie suivante.

Présentation

Un bloc est constitué de quatre parties :

- (1) le point d'entrée du bloc ;
- (2) une liste (éventuellement vide) d'instructions « phi » (voir partie 5) ;
- (3) une liste (éventuellement vide) d'instructions séquentielles ;
- (4) un *terminateur* (« *terminator* »), qui peut être :
 - soit une instruction de débranchement (« br %label ») ;
 - soit une instruction de retour de sous-programme (« ret ») ;
 - soit une condition élémentaire « br i1 %c, %vrai, %faux » ;
 - ...

Le point d'entrée d'un bloc est unique : on ne peut pas exécuter qu'une partie des instructions.

L'appel de sous-programme est considéré comme une instruction séquentielle.

Un terminateur définit **toujours** un débranchement.

X:

```
instructions_phi  
instructions  
br %Y
```

X:

```
instructions_phi  
instructions  
ret void
```

X:

```
instructions_phi  
instructions  
br i1 %c, %vrai, %faux
```

Pourquoi utiliser cette représentation ?

Simple à obtenir à partir des instructions structurées des langages.

La sémantique ne dépend pas de l'ordre des blocs.

Très facilement manipulable pour effectuer des optimisations qui conservent la sémantique du programme.

Traduction d'une instruction forever . . . end

L'instruction **forever** B **end** est une répétition infinie. On suppose que B est une liste d'instructions séquentielles.

```
...  
A  
forever  
B  
end
```

```
...  
A  
br %Y
```

```
Y:  
B  
br %Y
```

Traduction de l'instruction if (...){ ... }

L'instruction **if** (cc) { B } est classique. On suppose que cc est un booléen (type « i1 »), et que B est une liste d'instructions séquentielles.

```
...
A
if (cc) {
    B
}
C
...
```

```
...
A
br i1 %cc, %cc.true, %cc.false
```

```
cc.true:
    B
    br %cc.false
```

```
cc.false:
    C
    ...
```

Traduction de l'instruction `if (...) { ... }else{ ... }`

L'instruction `if (cc) { B }else{ C }` est aussi classique. On suppose que cc est une condition élémentaire, et que B et C sont des listes d'instructions séquentielles.

```
...
A
if (cc) {
    B
} else{
    C
}
D
...
```

X:
...
A
br i1 %cc, %cc.true, %cc.false

cc.true:
B
br %cc.end

cc.false:
C
br %cc.end

%cc.end:
D
...

Traduction de l'instruction

if (...) { ... } else if (...) { ... } else { ... }

```
...  
A  
if (cc1)  
B  
elsif (cc2)  
C  
else  
D  
end  
E  
...
```

Inutile de prendre en charge cette instruction. Il suffit que l'analyseur sémantique engendre des instructions **if then else** imbriquées.

Le seul cas à traiter est donc l'instruction **if then**
else.

```
...  
A  
if (cc1) {  
  B  
} else{  
  if (cc2) {  
    C  
} else{  
  D  
}  
}  
E  
...
```

```
...  
A  
br i1 %cc1, %cc1.true, %cc1.false
```

```
cc1.true:  
B  
br %cc1.end
```

```
cc1.false:  
br i1 %cc2, %cc2.true, %cc2.false
```

```
cc2.false:  
D  
br %cc2.end
```

```
%cc2.end:  
br %cc1.end
```

```
%cc1.end:  
E  
...
```

Condition non élémentaire : la négation « ! »

```
...  
A  
if (! cc) {  
    B  
}else{  
    C  
}  
D  
...
```

Deux possibilités :

- inverser les branches du terminateur ;
- effectuer la négation par l'instruction « xor » (LLVM n'a pas d'instruction de complémentation) ; l'optimiseur éliminera cette instruction au profit d'une inversion des branches.

```
...  
A  
br i1 %cc, %cc.f, %cc.t
```

```
cc.t:  
B  
br %cc.end
```

```
cc.f:  
C  
br %cc.end
```

```
%cc.end:  
D  
...
```

```
...  
A  
%cc.neg = xor i1 %cc, 1  
br i1 %cc.neg, %cc.t, %cc.f
```

```
cc.t:  
B  
br %cc.end
```

```
cc.f:  
C  
br %cc.end
```

```
%cc.end:  
D  
...
```

Condition non élémentaire : l'opération court-circuit « && »

Si cc1 est fausse, la condition est fausse : cc2 n'est pas évaluée et C est exécuté.

Si cc1 est vraie, cc2 est évaluée. Si cc2 est vraie, B est exécuté, sinon C est exécuté.

```
...
A
if (cc1 && cc2) {
    B
}else{
    C
}
D
...
```

```
...
A
br i1 %cc1, %cc.t1, %cc.f
```

```
cc.t1:
br i1 %cc2, %cc.t2, %cc.f
```

```
cc.t2:
B
br %cc.end
```

```
cc.f:
C
br %cc.end
```

```
%cc.end:
D
...
```

Condition non élémentaire : l'opération court-circuit << || >> ou

Si cc1 est vraie, la condition est vraie : cc2 n'est pas évaluée et B est exécuté.

Si cc1 est fausse, cc2 est évaluée. Si cc2 est vraie, B est exécuté, sinon C est exécuté.

```
...
A
if (cc1 || cc2) {
    B
} else{
    C
}
D
...
```

↔

```
...
A
if (!(!cc1 && !cc2)) {
    B
} else{
    C
}
D
...
```

Instruction while (...){ ... }

```
...  
A  
while (cc) {  
    B  
}  
D  
...
```

```
...  
A  
br %cc.test
```

```
cc.test:  
    br i1 %cc, %cc.loop, %cc.end
```

```
cc.loop:  
    B  
    br %cc.test
```

```
%cc.end:  
    D  
    ...
```

Génération de code

Quand on engendre le code assembleur, un terminateur « br %X » est traduit par une instruction de saut. Celle-ci peut être omise si c'est un branchemet vers le bloc suivant.

Une optimisation consiste donc à ré-ordonner les blocs de façon à minimiser le nombre de sauts.

Code assembleur

```
...
A
br %cc.test
```

```
cc.test:
br i1 %cc, %cc.loop, %cc.end
```

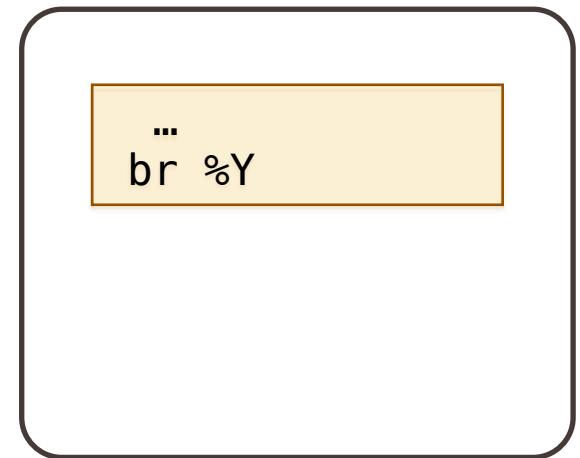
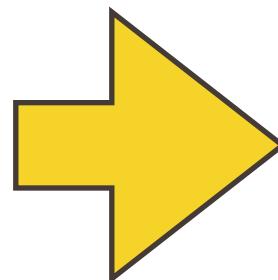
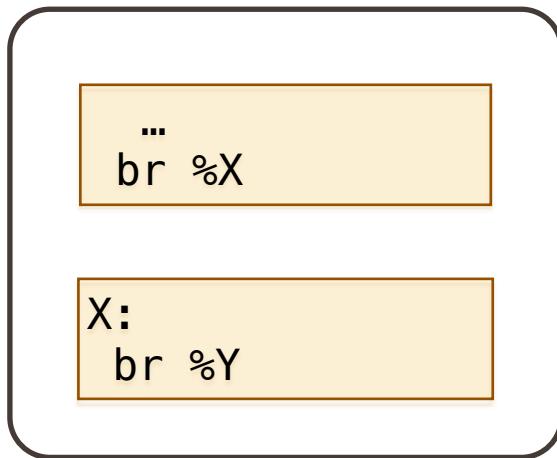
```
cc.loop:
B
br %cc.test
```

```
%cc.end:
D
...
```

```
A
goto cc.test
cc.test:
...
branch zero cc.end
goto cc.loop
cc.loop:
B
goto cc.test
cc.end:
D
...
```

Exemples d'optimisation

Une bloc sans instruction peut être supprimé, en changeant les branchements vers ce bloc.



5 — L'instruction « phi »

Exercice

Compiler en LLVM **avec optimisation** le code suivant :

```
uint32_t somme (uint32_t n) {  
    uint32_t r = 0 ;  
    while (n > 0) {  
        r += n ;  
        n-- ;  
    }  
    return r ;  
}
```

Le code LLVM obtenu met en évidence la structuration en blocs, et l'optimisation fait apparaître l'instruction phi, objet de cette partie.

Code obtenu par compilation avec option « -O3 »

```
define i32 @somme(i32 %n) #0 {  
    %1 = icmp eq i32 %n, 0  
    br i1 %1, label %12, label %.lr.ph ; Test n == 0  
  
.lr.ph:  
    %2 = add i32 %n, -1  
    %3 = mul i32 %2, %2  
    %4 = zext i32 %2 to i33  
    %5 = add i32 %n, -2  
    %6 = zext i32 %5 to i33  
    %7 = mul i33 %4, %6  
    %8 = lshr i33 %7, 1  
    %9 = trunc i33 %8 to i32  
    %10 = add i32 %3, %n  
    %11 = sub i32 %10, %9  
    br label %12  
  
; <label>:12 ; preds = %.lr.ph, %0  
%r.0.lcssa = phi i32 [ %11, %.lr.ph ], [ 0, %0 ]  
ret i32 %r.0.lcssa  
}
```

Test $n == 0$

; preds = %0

Calcul de la somme : la boucle a été dépliée

Instruction phi

L'instruction phi

<http://llvm.org/docs/LangRef.html#phi-instruction>

```
; <label>:15 ; preds = %.lr.ph, %0
%r.0.lcssa = phi i32 [ %14, %.lr.ph ], [ 0, %0 ]
ret i32 %r.0.lcssa
}
```

%r.0.lcssa prend pour valeur %14 si l'exécution vient de %.lr.ph, et 0 si l'exécution vient de %0.

Instruction phi et code engendré par un back-end

Engendrer les instructions phi dans le cas général est très compliqué.

Conseil : engendrer un code LLVM non optimisé, avec tous les objets valués en mémoire, ce qui rend inutile la génération d'instructions phi ; effectuer une traduction instruction par instruction, sans se soucier du fait que l'on peut relire une valeur en mémoire que l'on vient juster de stocker en mémoire.

L'optimiseur fera tout le travail d'optimisation, en engendrant au besoin les instruction phi.

6 — Cross compiler LLVM - CLANG

Options et librairie pour les différents processeurs ARM

ARM Core	Command Line Options	multilib
Cortex-M0+ Cortex-M0 Cortex-M1	-mthumb -mcpu=cortex-m0plus	armv6-m
	-mthumb -mcpu=cortex-m0	
	-mthumb -mcpu=cortex-m1	
	-mthumb -march=armv6-m	
Cortex-M3	-mthumb -mcpu=cortex-m3	armv7-m
	-mthumb -march=armv7-m	
Cortex-M4 (No FP)	-mthumb -mcpu=cortex-m4	armv7e-m
	-mthumb -march=armv7e-m	
Cortex-M4 (Soft FP)	-mthumb -mcpu=cortex-m4	armv7e-m /softfp
	-mfloat-abi=softfp -mfpu=fpv4-sp-d16	
	-mthumb -march=armv7e-m	
	-mfloat-abi=softfp -mfpu=fpv4-sp-d16	
Cortex-M4 (Hard FP)	-mthumb -mcpu=cortex-m4	armv7e-m /fpu
	-mfloat-abi=hard -mfpu=fpv4-sp-d16	
	-mthumb -march=armv7e-m	
	-mfloat-abi=hard -mfpu=fpv4-sp-d16	

Options et librairie pour les différents processeurs ARM

ARM Core	Command Line Options	multilib
Cortex-M7 (No FP)	-mthumb -mcpu=cortex-m7	cortex-m7
Cortex-M7 (Soft FP)	-mthumb -mcpu=cortex-m7 -mfloating-point-abi=softfp -mfpu=fpv5-sp-d16	cortex-m7 /softfp /fpv5-sp-d16
	-mthumb -mcpu=cortex-m7 -mfloating-point-abi=softfp -mfpu=fpv5-d16	cortex-m7 /softfp /fpv5-d16
Cortex-M7 (Hard FP)	-mthumb -mcpu=cortex-m7 -mfloating-point-abi=hard -mfpu=fpv5-sp-d16	cortex-m7 /fpu /fpv5-sp-d16
	-mthumb -mcpu=cortex-m7 -mfloating-point-abi=hard -mfpu=fpv5-d16	cortex-m7 /fpu /fpv5-d16
Cortex-R* (No FP)	[<code>-mthumb</code>] -march=armv7-r	armv7-ar /thumb
Cortex-R* (Soft FP)	[<code>-mthumb</code>] -march=armv7-r -mfloating-point-abi=softfp -mfpu=vfpv3-d16	armv7-ar /thumb /softfp
Cortex-R* (Hard FP)	[<code>-mthumb</code>] -march=armv7-r -mfloating-point-abi=hard -mfpu=vfpv3-d16	armv7-ar /thumb /fpu
Cortex-A* (No FP)	[<code>-mthumb</code>] -march=armv7-a	armv7-ar /thumb
Cortex-A* (Soft FP)	[<code>-mthumb</code>] -march=armv7-a -mfloating-point-abi=softfp -mfpu=vfpv3-d16	armv7-ar /thumb /softfp
Cortex-A* (Hard FP)	[<code>-mthumb</code>] -march=armv7-a -mfloating-point-abi=hard -mfpu=vfpv3-d16	armv7-ar /thumb /fpu

Compilateur croisé via la distribution binaire (1/3)

La distribution binaire présente un support très limité de la compilation croisée :

- ne pas faire appel à des fichiers d'en-tête système (pas de `#include <...>`);
- uniquement émission de code LLVM ou assembleur.

Cela est suffisant pour des exemples simples.

Pour obtenir la liste des processeurs cibles :

```
llc --version
```

```
llc --version
LLVM (http://llvm.org/):
  LLVM version 3.6.2
  Optimized build.
  Default target: x86_64-apple-darwin14.5.0
  Host CPU: core-avx2

  Registered Targets:
    aarch64      - AArch64 (little endian)
    aarch64_be   - AArch64 (big endian)
    amdgcn      - AMD GCN GPUs
    arm          - ARM
    arm64        - ARM64 (little endian)
    armeb        - ARM (big endian)
    cpp          - C++ backend
    hexagon      - Hexagon
    mips         - Mips
    mips64       - Mips64 [experimental]
    mips64el     - Mips64el [experimental]
    mipsel       - Mipsel
    msp430        - MSP430 [experimental]
    nvptx        - NVIDIA PTX 32-bit
    nvptx64      - NVIDIA PTX 64-bit
    ppc32        - PowerPC 32
    ppc64        - PowerPC 64
    ppc64le      - PowerPC 64 LE
    r600          - AMD GPUs HD2XXX-HD6XXX
    sparc         - Sparc
    sparcv9      - Sparc V9
    systemz      - SystemZ
    thumb         - Thumb
    thumbeb      - Thumb (big endian)
    x86          - 32-bit X86: Pentium-Pro and above
    x86-64       - 64-bit X86: EM64T and AMD64
    xcore        - XCore
```

Compilateur croisé via la distribution binaire (2/3)

Un exemple (sans optimisation du code LLVM engendré)

addition.c

```
int addition (int a, int b) {  
    return a+b ;  
}
```

Exemple de compilation croisée *non optimisée* pour processeur ARM Cortex-M4.

```
clang --target=armv7-none--eabi -mcpu=cortex-m4 -emit-llvm -S addition.c -o addition.ll
```

```
; ModuleID = 'addition.c'  
target datalayout = "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64"  
target triple = "thumbv7em-none--eabi"  
  
; Function Attrs: nounwind  
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    store i32 %a, i32* %1, align 4  
    store i32 %b, i32* %2, align 4  
    %3 = load i32* %1, align 4  
    %4 = load i32* %2, align 4  
    %5 = add nsw i32 %3, %4  
    ret i32 %5  
}  
  
attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  
  
!llvm.module.flags = !{!0, !1}  
!llvm.ident = !{!2}  
  
!0 = !{i32 1, !"wchar_size", i32 4}  
!1 = !{i32 1, !"min_enum_size", i32 4}  
!2 = !{"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

addition.ll

Compilateur croisé via la distribution binaire (3/3)

Un exemple (avec optimisation du code LLVM engendré)

addition.c

```
int addition (int a, int b) {  
    return a+b ;  
}
```

Exemple de compilation croisée *optimisée* pour processeur ARM Cortex-M4.

```
clang --target=armv7-none--eabi -mcpu=cortex-m4 -emit-llvm -S -O3 addition.c -o addition.ll
```

```
; ModuleID = 'addition.c'  
target datalayout = "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64"  
target triple = "thumbv7em-none--eabi"  
  
; Function Attrs: nounwind readnone  
define i32 @addition(i32 %a, i32 %b) #0 {  
    %1 = add nsw i32 %b, %a  
    ret i32 %1  
}  
  
attributes #0 = { nounwind readnone "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false" "use-soft-float"="false" }  
  
!llvm.module.flags = !{!0, !1}  
!llvm.ident = !{!2}  
  
!0 = !{i32 1, !"wchar_size", i32 4}  
!1 = !{i32 1, !"min_enum_size", i32 4}  
!2 = !{"clang version 3.6.2 (tags/RELEASE_362/final)"}
```

addition.ll

7 — Travaux pratiques

Travaux pratiques

Écrire un source LLVM comprenant des routines effectuant addition, soustraction, multiplication, ... sur N bits, N étant une valeur constante que vous choisirez entre 2 et 31.

Par exemple, pour l'addition non signée :

```
define zeroext i23 @myUnsignedAddition (i23 zeroext %a, i23 zeroext %b) nounwind {  
    ...  
}
```

Par exemple, pour l'addition signée :

```
define signext i23 @mySignedAddition (i23 signext %a, i23 signext %b) nounwind {  
    ...  
}
```

Pour que ces routines soient connues d'un source C, les déclarer dans un fichier d'en-tête :

```
uint32_t myUnsignedAddition (uint32_t a, uint32_t b) ;  
int32_t mySignedAddition (int32_t a, int32_t b) ;
```

Infos sur LLVM : consulter le lien <http://llvm.org/docs/LangRef.html>

Introduction

Le but de cette partie est de vous faire écrire du code assembleur LLVM.

Comment s'y prendre :

- LA référence : <http://llvm.org/docs/LangRef.html> ;
- une façon simple d'aborder l'assembleur LLVM est de compiler des exemples simples écrits en C avec clang ; ne pas utiliser d'option d'optimisation pour obtenir un code issu d'une traduction élémentaire instruction par instruction ;
- pour écrire du code LLVM, SURTOUT ne pas essayer d'écrire du code optimisé ! l'optimiseur fera le travail d'optimisation bien mieux que nous.