

# From Source to Binary

How A Compiler Works

從原始碼到二進制

ソースからバイナリへ

Von Quelle Zu Binären

De source au binaire

Desde fuente a binario

Binarium ut a fonte

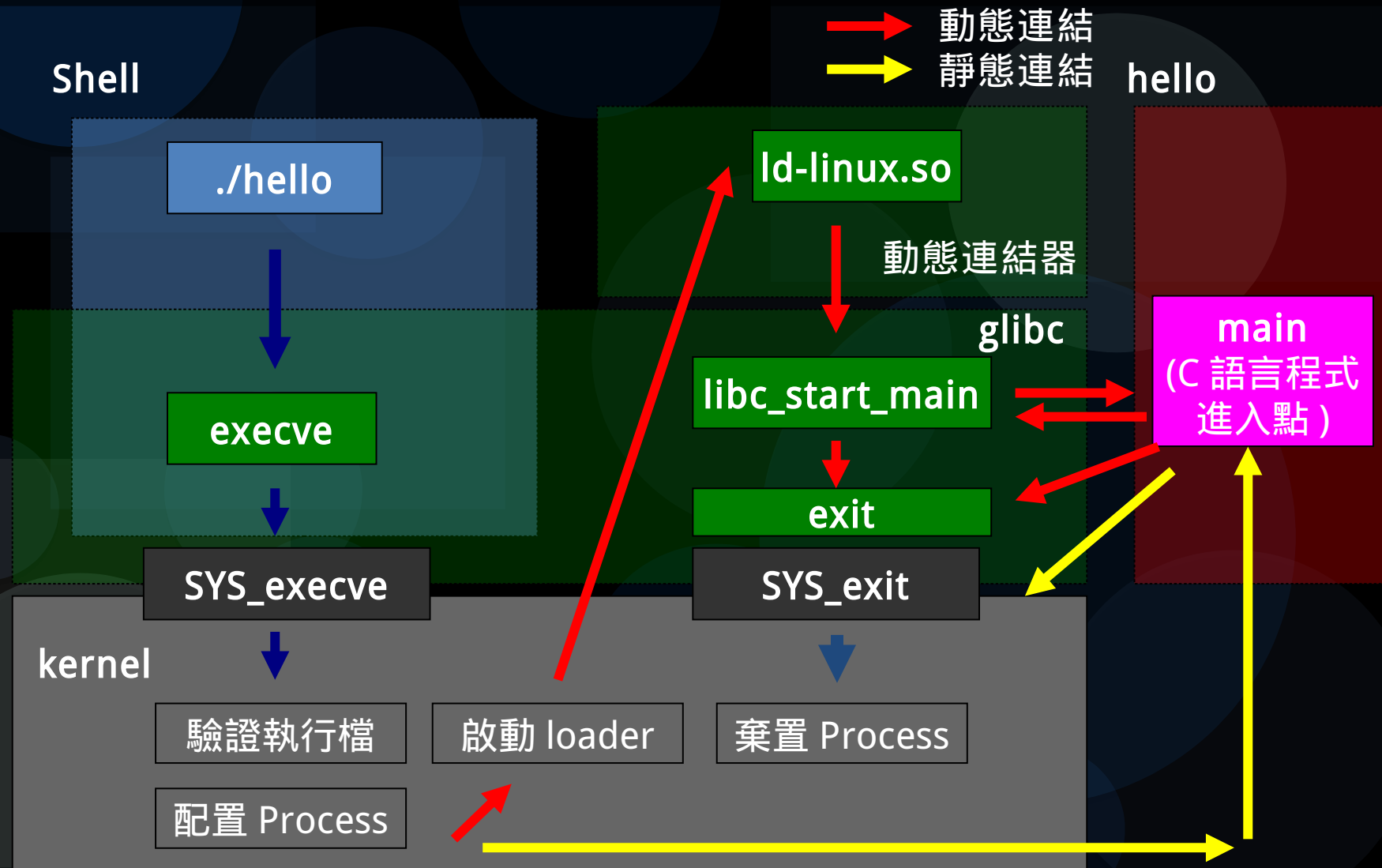
Luse Cheng, Kito Cheng, Jim Huang  
Mar 2, 2015

# 本簡報的目的

---

- 理解從 C 程式原始碼到二進制的過程，從而成為電腦的主人
- 觀察 GNU Toolchain 的運作並探討其原理
- 引導不具備編譯器背景知識的聽眾，得以對編譯器最佳化技術有概念
- 選定一小部份 ARM 和 MIPS 組合語言作為示範

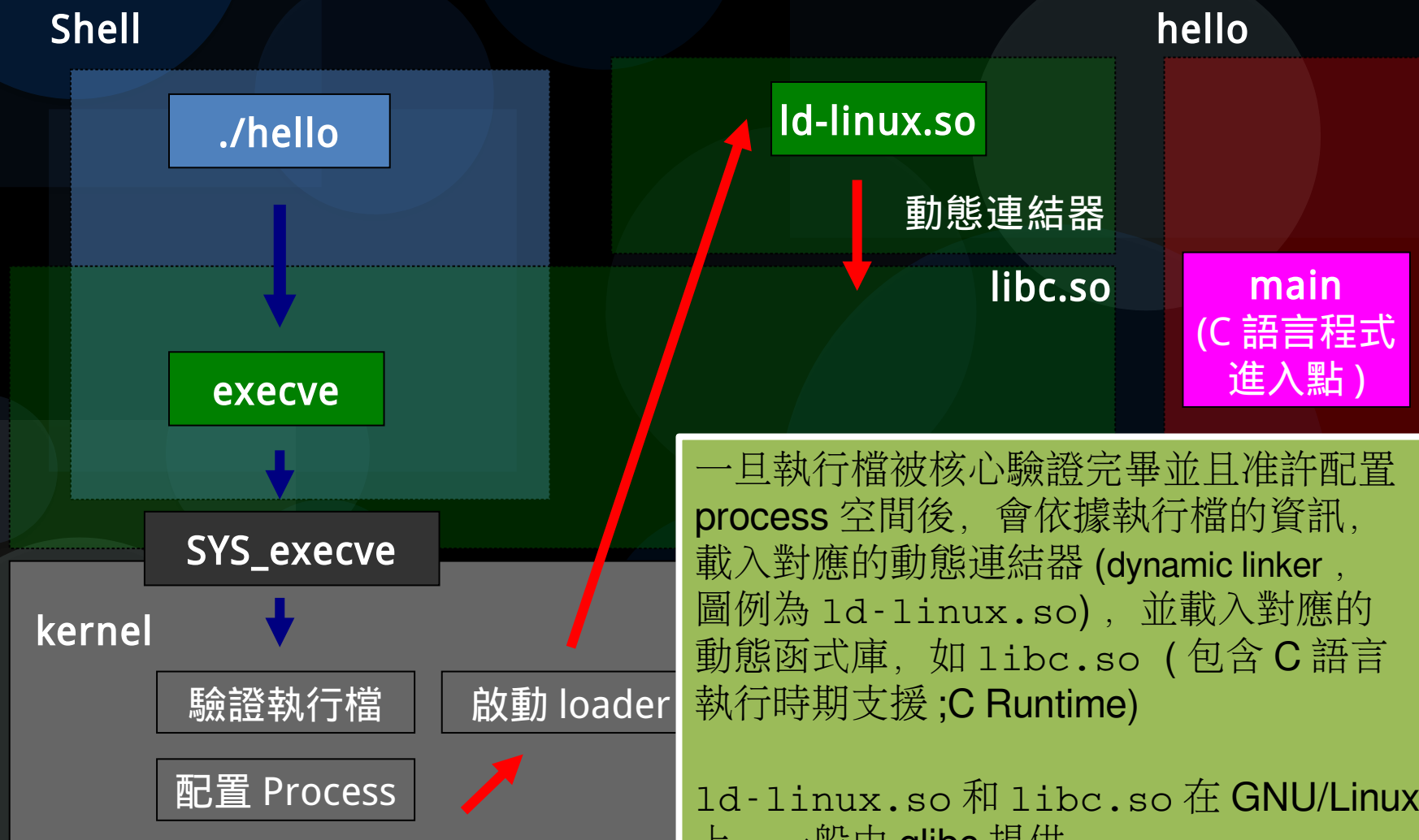
# 現代 GNU/Linux 環境，程式執行流程



# 執行 `execve` 系統呼叫



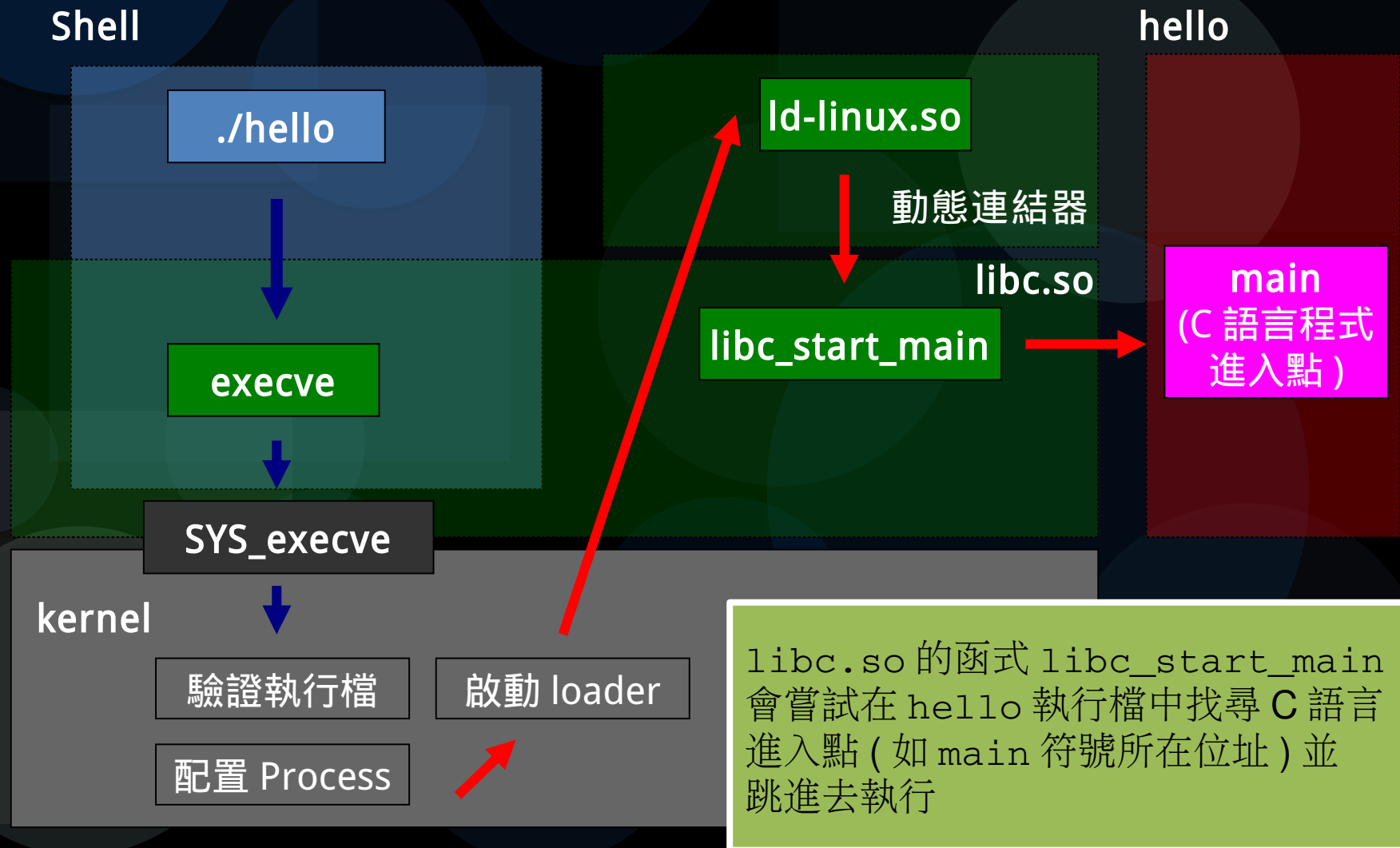
# 驗證執行檔並呼叫動態連結器



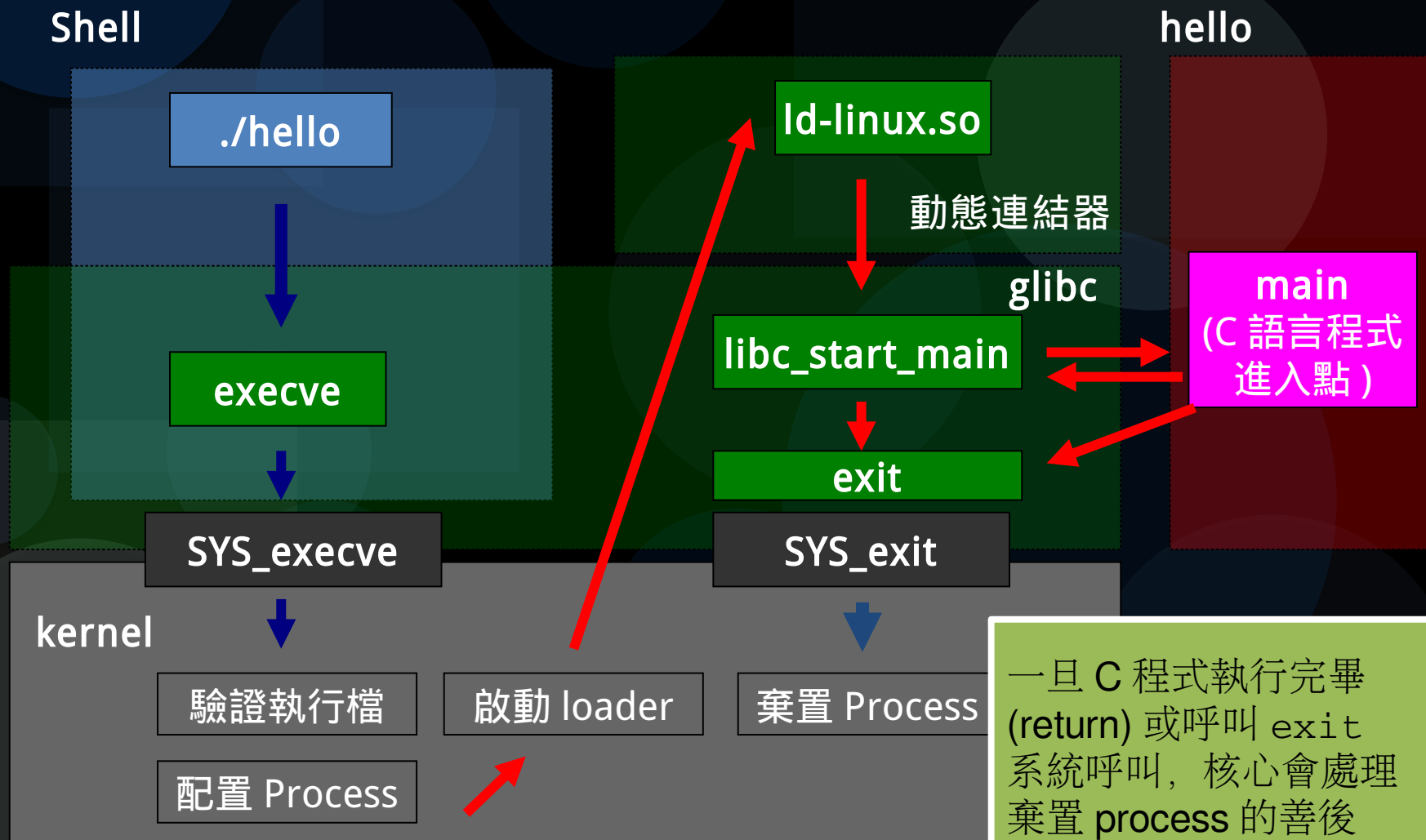
一旦執行檔被核心驗證完畢並且准許配置 process 空間後，會依據執行檔的資訊，載入對應的動態連結器 (dynamic linker, 圖例為 `ld-linux.so`)，並載入對應的動態函式庫，如 `libc.so` (包含 C 語言執行時期支援 ;C Runtime)

`ld-linux.so` 和 `libc.so` 在 GNU/Linux 上，一般由 `glibc` 提供

# 由 C 執行時期跳入真正的程式進入點



# C 語言程式結束並呼叫 `exit` 系統呼叫



# 從原始碼到二進制：常見的誤解

- 從原始碼到二進制的過程，不僅需要 Compiler
- 還要有很多工具介入才可完成：Toolchain
- 真正的 Compiler 其實只做以下：

軟體

硬體

Source Code

HDL Source

Compiler (gcc, llvm)

Design Compiler (DC)  
RTL Compiler (RC)

Selecting  
Instruction

Technology  
Mapping

Assembly

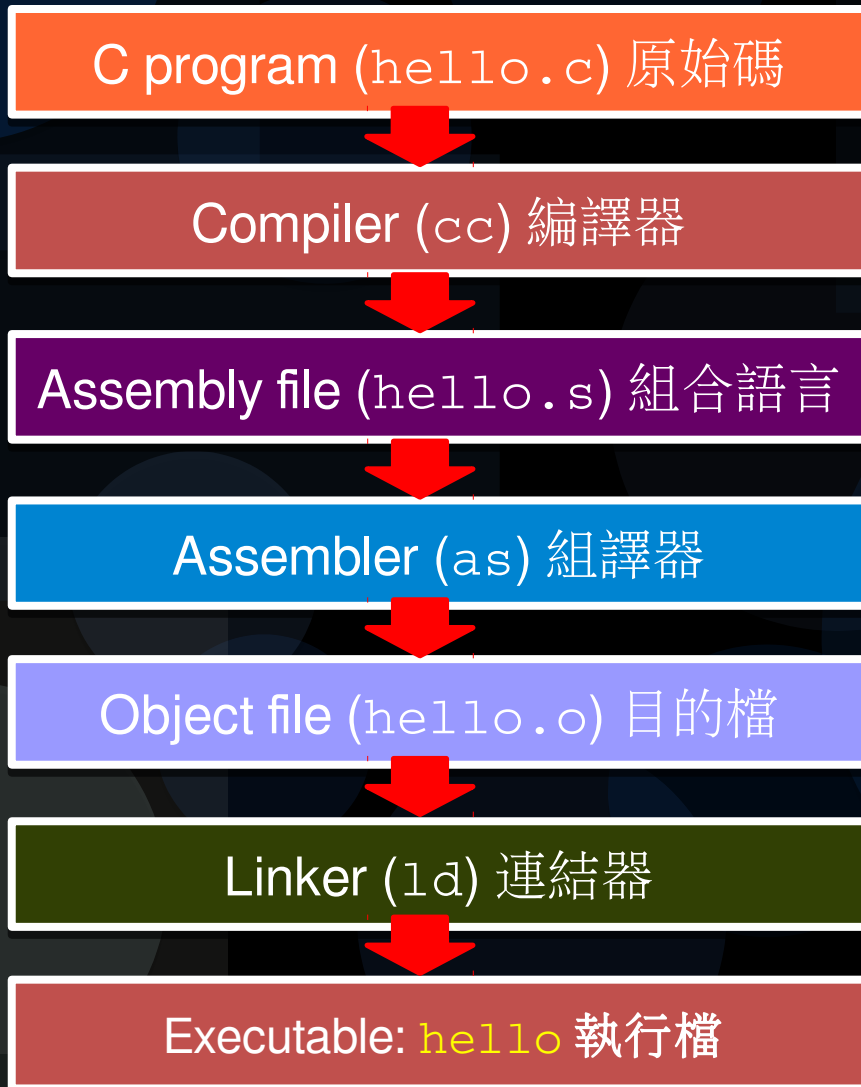
Netlist

A compiler is a **computer program (or set of programs)** that **transforms** source code written in a programming language (the **source language**) into another computer language (the **target language**, often having a binary form known as **object code**)

source : <http://en.wikipedia.org/wiki/Compiler>



# 編譯 Hello world 程式的流程



三大步驟

編譯

組譯

連結

# 圖解編譯流程 (1/5)

```
#include <stdio.h>
int main() {
    printf ("Hello World");
    return 0;
}
```

- 給定一個普通的輸出 Hello World 的程式
- 編譯方式：
  - `gcc -o hello hello.c` (或) `clang -o hello hello.c`
- 檢驗正確輸出 ELF 格式的執行檔
  - `file hello`
  - 預期輸出：  
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs)
- 執行方式：
  - `./hello`

# 圖解編譯流程 (2/5)

```
#include <stdio.h>
int main() {
    printf ("Hello World");
    return 0;
}
```

stdio.h

```
int printf (const char* fmt, ...);
```

Pre-  
process

```
int main() {
    printf("Hello World");
    return 0;
}
```

- C preprocessor 參照標頭檔 `stdio.h` 的內容，展開 `macro` 和驗證 `prototype`，因此輸出的結果就不會再見到 `#include` 字樣

# 圖解編譯流程 (3/5)

```
#include <stdio.h>
int main() {
    printf ("Hello World");
    return 0;
}
```

stdio.h

```
int printf (const char* fmt, ...);
```

Pre-  
process

```
int main() {
    printf ("Hello World");
    return 0;
}
```

Compile

```
main:
LDR R1, [R2 + 12]
BAL printf
```

- C compiler 輸出對應的組合語言，上圖針對 ARM 架構
- BAL 是 ARM 的一種跳躍指令，後方緊接著 printf 的位址
- 編譯階段不知道 printf 的位址，所以暫時填入 printf 符號名稱

# 圖解編譯流程 (4/5)

```
#include <stdio.h>
int main() {
    printf ("Hello World");
    return 0;
}
```

stdio.h

```
int printf (const char* fmt, ...);
```

Pre-  
process

```
int main() {
    printf ("Hello World");
    return 0;
}
```

Compile

```
main:
LDR R1, [R2 + 12]
BAL printf
```

Assemble

```
main:
    EC 00 00 12
    F0 ?? ?? ??
```

- ARM assembler 依據編譯器輸出的組合語言，組譯為對應的 ARM 機械碼
- 此時仍不知道 printf 的地址，所以暫時不填入

# 圖解編譯流程 (5/5)

```
#include <stdio.h>
int main() {
    printf ("Hello World");
    return 0;
}
```

stdio.h

```
int printf (const char* fmt, ...);
```

Pre-process

```
int main() {
    printf ("Hello World");
    return 0;
}
```

Compile

```
main:
LDR R1, [R2 + 12]
BAL printf
```

Assemble

```
main:
    EC 00 00 12
    F0 ?? ?? ??
```

Linking

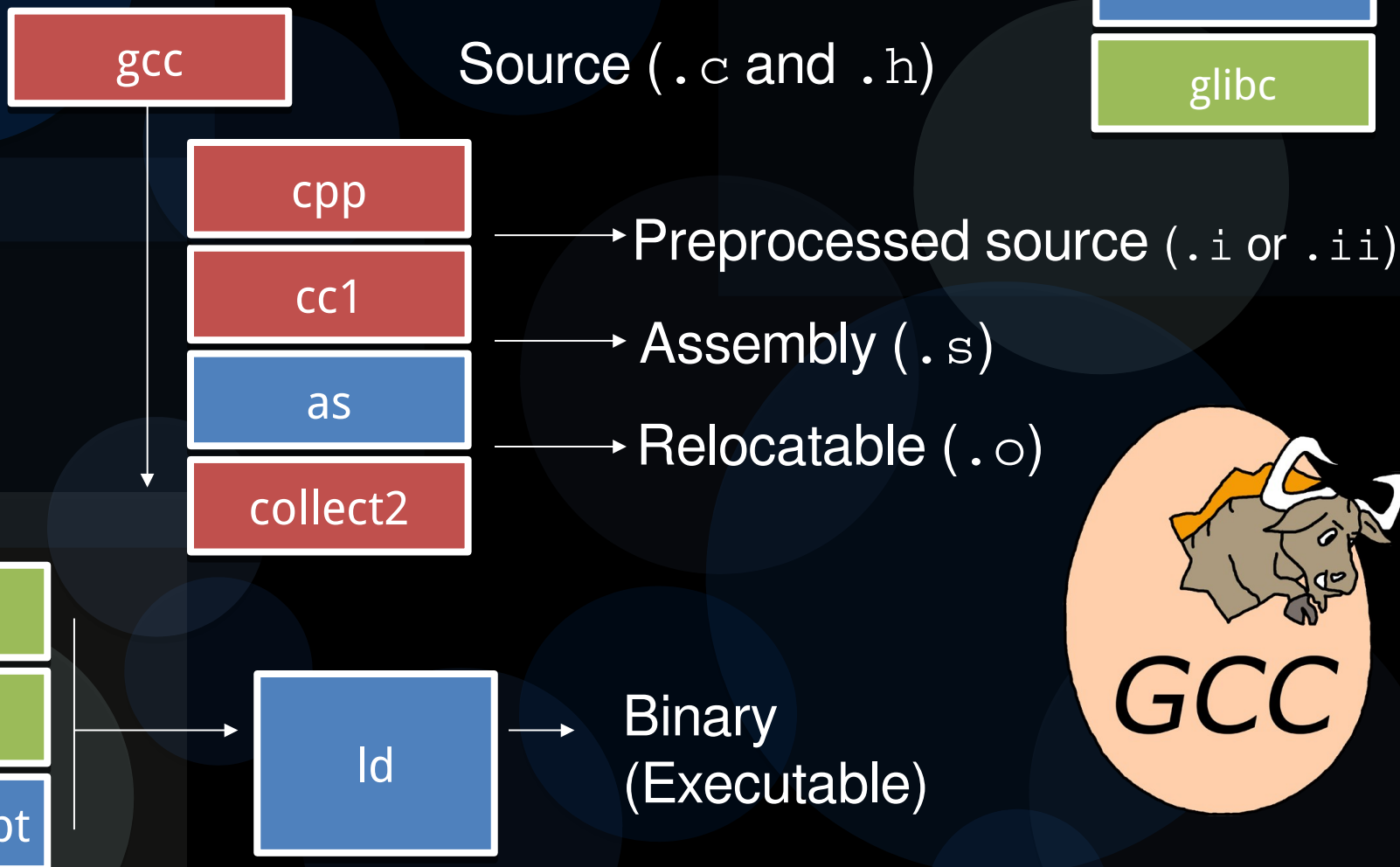
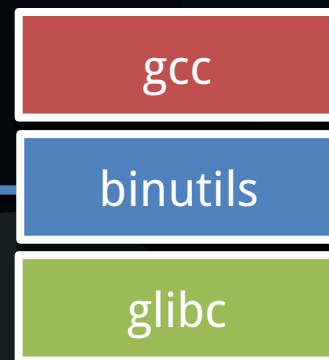
libc.a

```
0x1000 <printf>:
    EC 00 00 12
    ..
```

```
0x2000 <main>:
    EC 00 00 12
    F0 00 10 00
```

• printf 實做於 libc.a(C 語言標準函式庫的靜態版本), 其位址為 0x1000, Linker 重新配置 (relocate)

常見的誤解：GCC 是個 C 編譯器  
事實：GCC 是個 Compiler Driver



**GCC** 這個執行檔，可當 **Compiler**，也能當 **Preprocessor**，也能當 **Assembler**，更能當 **Linker**

# 編譯器簡史

(程式語言不是年代較早，就比較低階。1970 年代才出現的 C 語言遠比十幾年前就出現的 Fortran 與 COBOL 低階)

- First compiler (1952)
  - 第一個編譯器
  - A-0, **Grace Murray Hopper**
- First complete compiler (1957)
  - 第一個完整的編譯器
  - Fortran, **John Backus** (18-man years)
- First multi-arch compiler (1960)
  - 第一個多平台支援的編譯器
  - COBOL, **Grace Murray Hopper** et al.
- The first self-hosting compiler (1962)
  - 第一個能自己編譯自己的編譯器
  - LISP, Tim Hart and Mike Levin



**Grace Hopper**, inventor of A-0, COBOL, and the term “compiler.”



Image source:

<http://www-history.mcs.st-and.ac.uk/PictDisplay/Hopper.html>

[http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace\\_Hopper.jpg/300px-Grace\\_Hopper.jpg](http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg)



# 先有雞還是先有蛋？

- 先有 C 語言還是先有 C 編譯器？
- 故事： Ken Thompson – Reflections on Trusting Trust
  - (Compiling a compiler)  
在 C compiler( 事實上是 pre-processor) 中放了一個後門，若 compiler 發現它所處理的程式是 login 的原始程式，它就會在裡面加入一段程式碼，讓 Ken Thompson 可以任意 login 進去該主機。由於 UNIX 原始是碼得用 C compiler 來編譯，這意味著 Thompson 將可登入任何安裝 UNIX 的電腦
  - 那換個乾淨的 compiler 總可以吧？但新 compiler 的原始碼也得由舊 compiler 來編譯，而為了讓這個特洛伊木馬能流傳久遠，如果舊 compiler 發現自己正處理的是另一個 ( 正常的 ) C compiler，它就會把前述那段「如果是 login，就加入一段…」的程式碼，注入到這個新的 compiler 裡面

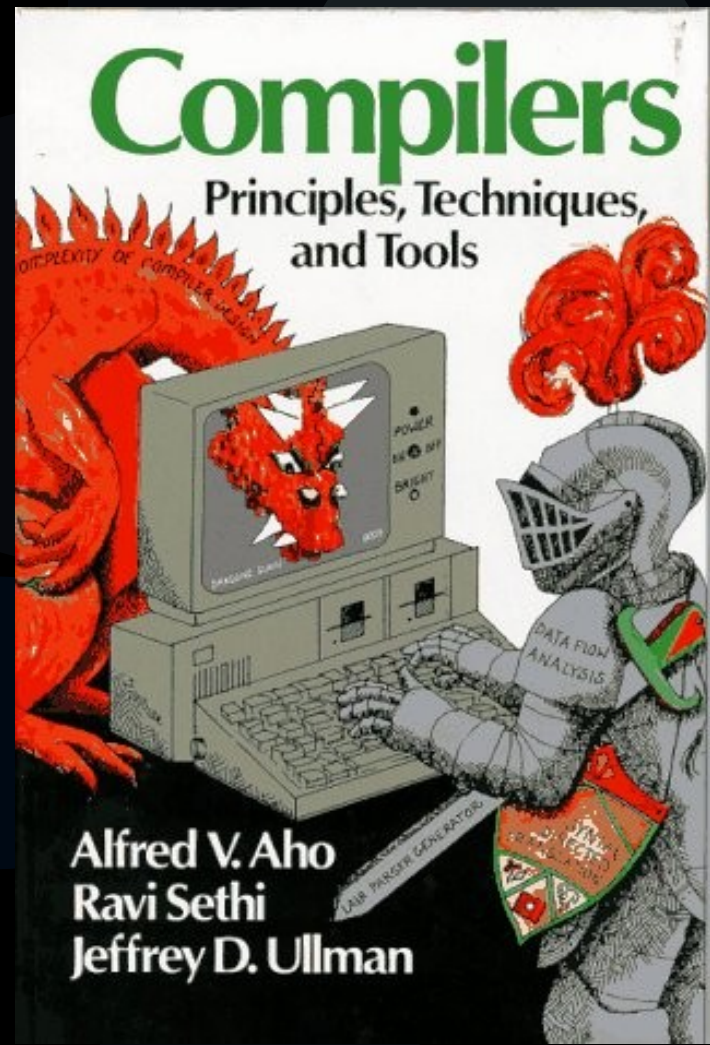
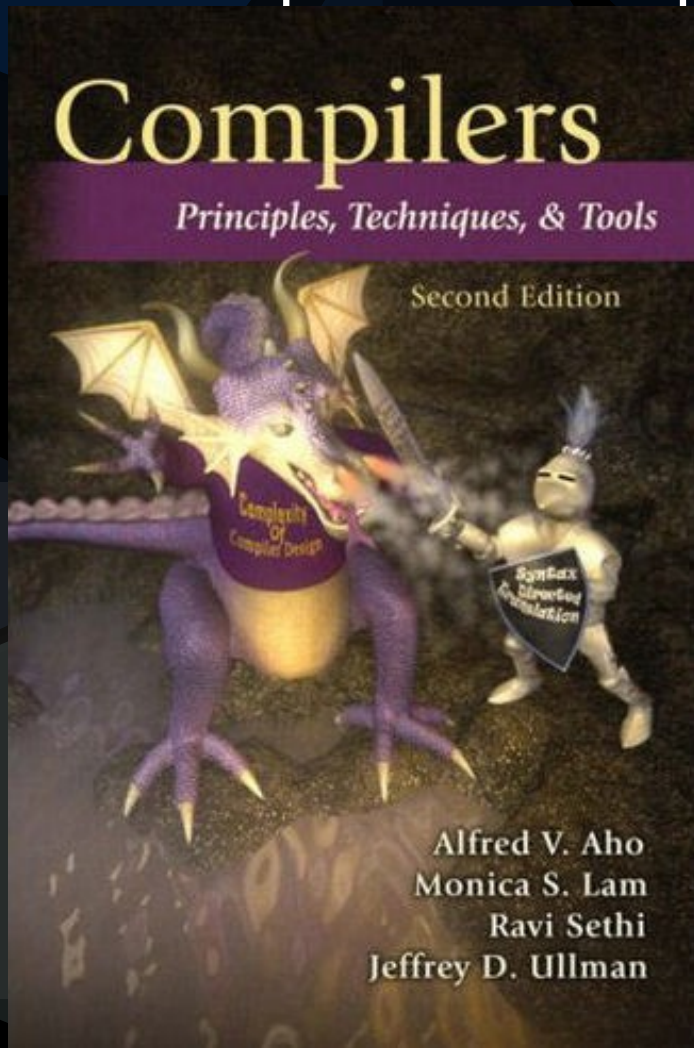
# Bootstrapping & Cross-compilation

- 原有系統的内建 C compiler 可能較簡陋或老舊，因此我們想把新的 compiler 用舊的 compiler 編譯，然後當成系統内建的 compiler 用
  - 先有雞還是先有蛋？
    - 用 X 語言（像是既有的 Fortran，或者直接用組合語言）寫一個 C-（比 C 語言更簡化的語言）的編譯器
    - 用 C- 寫一個 C 的編譯器
    - 用 C 寫一個 C 的編譯器
- Oracle JDK 中，Java 編譯器 javac 就是用 Java 語言撰寫，甚至有 Maxine Research VM 這樣主要用 Java 撰寫的 Java 虛擬機器**

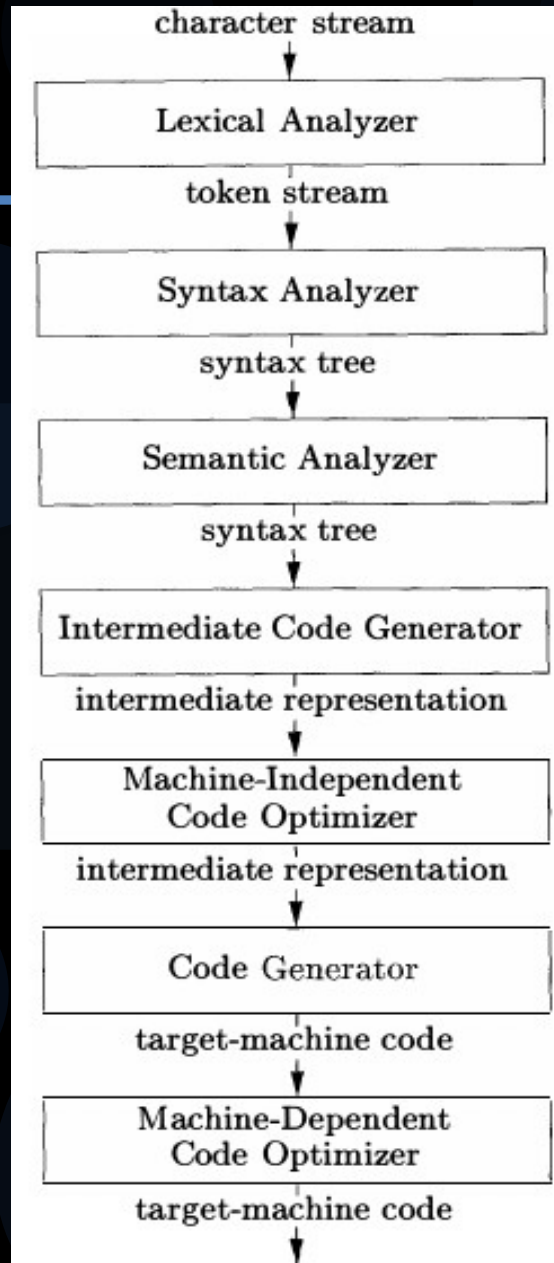
相傳在十八世紀，德國 Baron Münchhausen 男爵常誇大吹噓自己的英勇事蹟，其中一項是「拉著自己的頭髮，將自己從受陷的沼澤中提起」，此事後來收錄於《吹牛大王歷險記》，改寫為「用拔靴帶把自己從海中拉起來」，其中「拔靴帶」（bootstrap）指的是長統靴靴筒頂端後方的小環帶，是用以輔助穿長統靴。這種有違物理原理的誇大動作，卻讓不同領域的人們獲得靈感，商業上，bootstrapping 引申為一種創業模式，即初期投入少量的啓動資本，然後在創業過程中主要依靠從客戶得來的銷售收入，形成一個良好的正現金流。在資訊領域，因開機過程環環相扣，得先透過簡單的程式讀入記憶體，執行後又載入更多磁區、程式碼來執行，直到作業系統完全載入為止，因此開機過程也稱 bootstrapping，簡稱 "boot"

# 編譯器教科書

## 《 Compilers: Principles, Techniques, & Tools 》

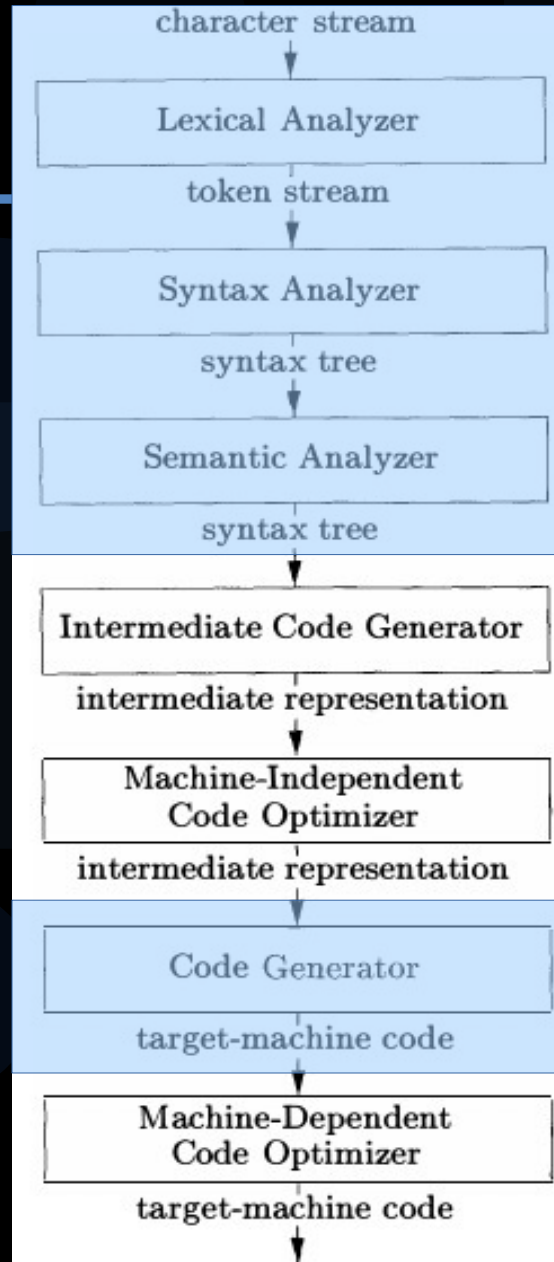


# 編譯流程





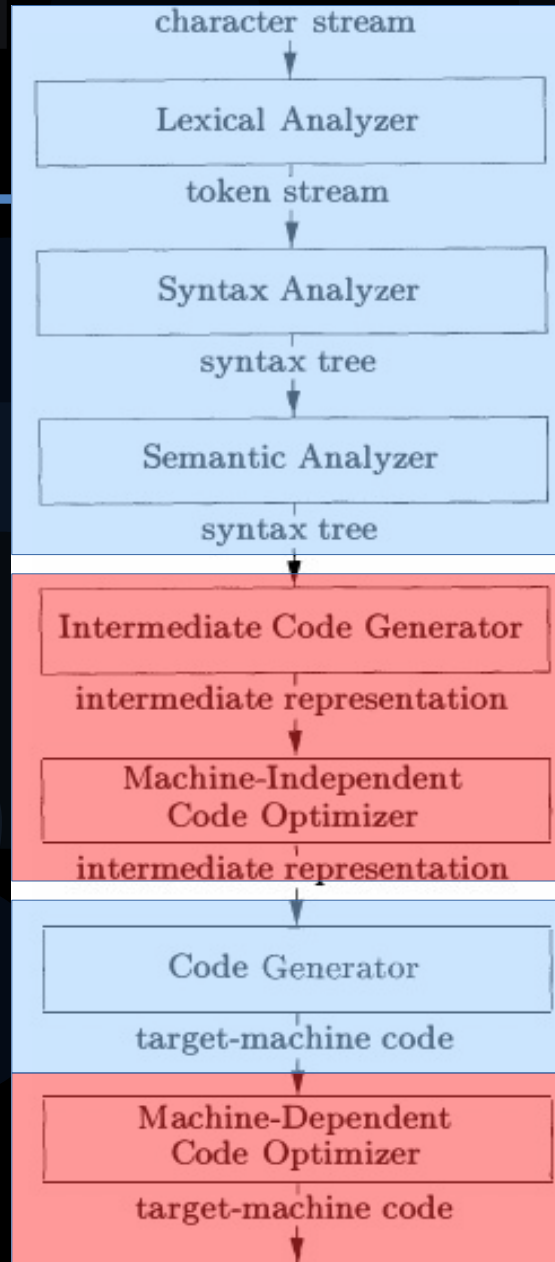
# 編譯流程



通常大學部編譯器課程  
僅涵蓋 parser 的部份

以及陽春的 code generator

# 編譯流程



但 Compiler 引人入勝之處，  
其實都在最佳化的地方

# 編譯器術語: Basic Block

- 單一進入點，單一出口點的程式區段

```
int foo (int n)
{
    int ret;
    if (n > 10)
        ret = n * 2;
    else
        ret = n + 2;
    return ret;
}
```

```
int ret;
if (n > 10)
```

```
ret = n * 2;
```

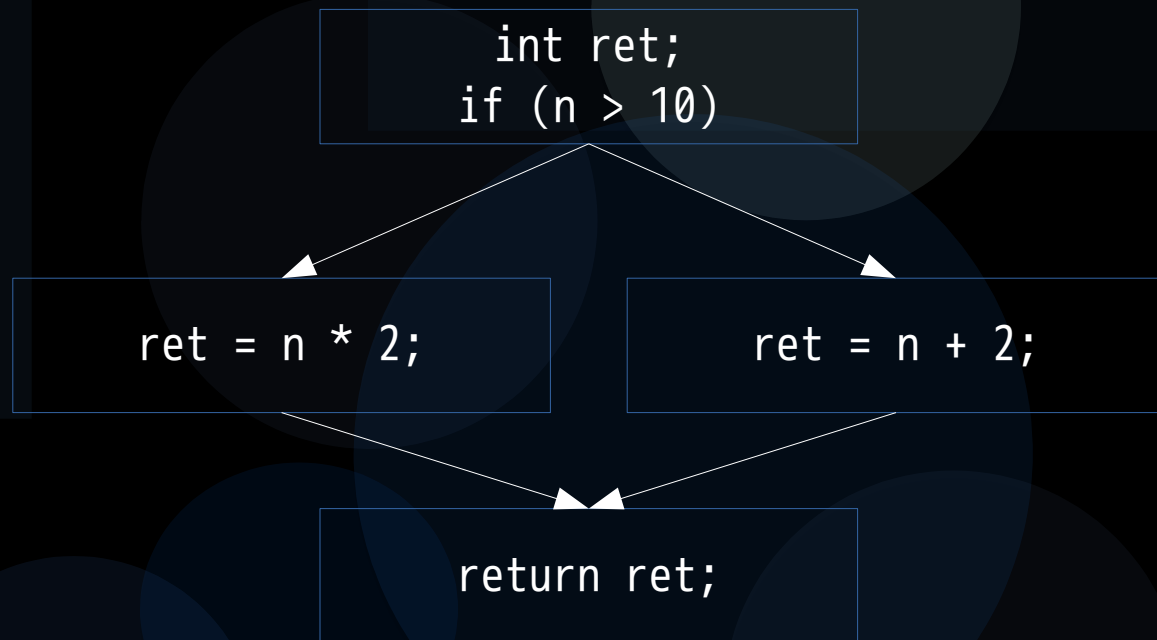
```
ret = n + 2;
```

```
return ret;
```

# 編譯器術語： CFG (Control Flow Graph)

- 簡單來說就是程式的流程圖

```
int foo (ini n)
{
    int ret;
    if (n > 10)
        ret = n * 2;
    else
        ret = n + 2;
    return ret;
}
```





# Basic Block vs. CFG

## Basic Block

```
int ret = 0;  
int i;
```

```
i = 0;
```

```
i < n;
```

```
ret += i;
```

```
++i
```

```
return ret
```

```
int sum (int n)  
{  
    int ret = 0;  
    int i;  
    for (i = 0; i < n; ++i)  
        ret += i;  
    return ret;  
}
```

## CFG

```
int ret = 0;  
int i;
```

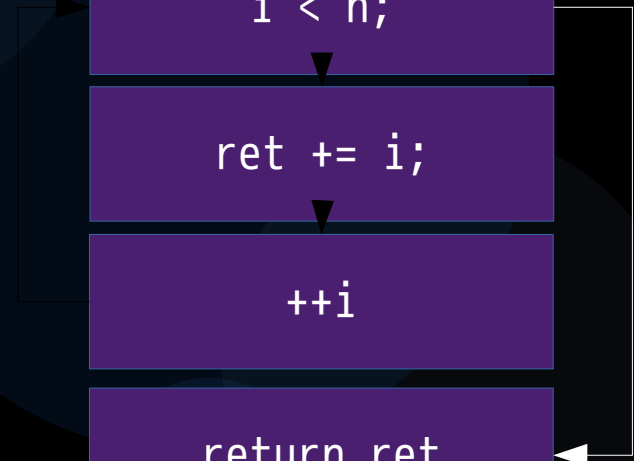
```
i = 0;
```

```
i < n;
```

```
ret += i;
```

```
++i
```

```
return ret
```



# 從教科書到真實世界

- 「龍書」的關鍵概念 : Syntax Directed Translator

- 課本終究還是課本
- 程式的語法樹結構將無法和處理器架構脫鉤
- 函式參數的傳遞順序、堆疊處理

- GCC 的特性與挑戰 :

- 支援多種語言 ( 前端 ; front-end )
- 支援多種處理器 ( 後端 ; back-end )

```
int main()
{
    return 0;
    IamDeadCode();
}
```

- 思考 : 有些最佳化和語言跟硬體平台無關

- 如: 消除沒有用到的程式碼 (**Dead code elimination**)
- GCC 引入中間層最佳化 ( 編譯器相關最佳化 )



# Code Motion ( 程式碼搬移 )

---

- 有些運算提前執行，可能會比較快，例如 **Loop Invariant**
- **Loop Invariant**: 迴圈內有些運算，不論跑幾次都是相同結果，這類的運算拉出迴圈外則只要算一次即可

# 資料流程分析 (Data-flow analysis)

- 為什麼需要資料流程分析？
- 兩個指令在什麼狀況可交換位置 (code motion) ?
  - 如果兩個指令彼此沒有相依性的話
- 資料流程分析 = 偵測指令相依性
- **Reaching definition** for a given instruction is another instruction, the target variable of which may reach the given instruction without an intervening assignment
  - Compute [use-def chains](#) and [def-use chains](#)

# Code Motion & Pointer Aliasing

- C 語言中阻擋編譯器最佳化的一大原因
- **Pointer Aliasing**: 兩個不同的指標指到同一個記憶體位置

考慮以下最佳化技巧

```
while (y < z) {  
    int x = a + b;  
    y += x;  
}
```



```
int t1 = a + b;  
while (y < z) {  
    y += t1;  
}
```

那麼以下程式碼是否適用前述最佳化?

```
void foo(int *a, int *b, int *y, int z) {  
    while (*y < z) {  
        int x = *a + *b;  
        *y += x;  
    }  
}
```

既然  $x=a+b$  涉及重複的加法和變數指定，而  $a$  和  $b$  在 `while` 陳述中未曾改變過內含值，於是  $x$  必為不會變動，所以最佳化的策略即為在 `while` 陳述前預先計算  $a+b$  的值，並在 `while` 陳述中帶入

# Code Motion & Pointer Aliasing

- 貿然施加最佳化，會導致什麼後果？

```
void foo(int *a, int *b, int *y, int z) {  
    while (*y < z) {  
        int x = *a + *b;  
        *y += x;  
    }  
}
```



```
void foo(int *a, int *b, int *y, int z) {  
    int t1 = *a + *b;  
    while (*y < z) {  
        *y += t1;  
    }  
}
```

- 編譯器**不能**把  $(*a) + (*b)$  移到迴圈外
- 因為不確定開發者是否這樣寫：  
`foo(&num, &num, &num, 5566);`
- 這樣  $a$ ,  $b$ ,  $y$  互為別名 (alias),  $*y$  的值改變會使  $*a$  和  $*b$  連帶改變，所以不能提出到迴圈外

# Strict Aliasing & Restrict Pointer Aliasing

- 如果 **Pointer Aliasing** 無限上綱
  - 一旦遇到 **C pointer**，編譯器所有的最佳化都無用武之地
- **Strict Aliasing Rule**
  - 現代編譯器假設不同 **type** 之間的 **pointer** 沒有 **aliasing**
  - 但有很多舊的程式碼並非這樣認定，造成了很多問題
  - **GCC** 中可用 `-fno-strict-aliasing` 編譯參數，來抑制 **Strict Aliasing rule**
- **C99: Restrict Pointer Aliasing**
  - 使用 `restrict` 關鍵字（**GCC** 需要一併使用 `--std=c99` 編譯參數讓編譯器採用 **C99** 規格）告知編譯器指定的 **pointer** 不會有 **Pointer Aliasing** 疑慮

# Static Single Assignment, SSA Form

- 什麼是 SSA Form
  - 靜態單賦值形式

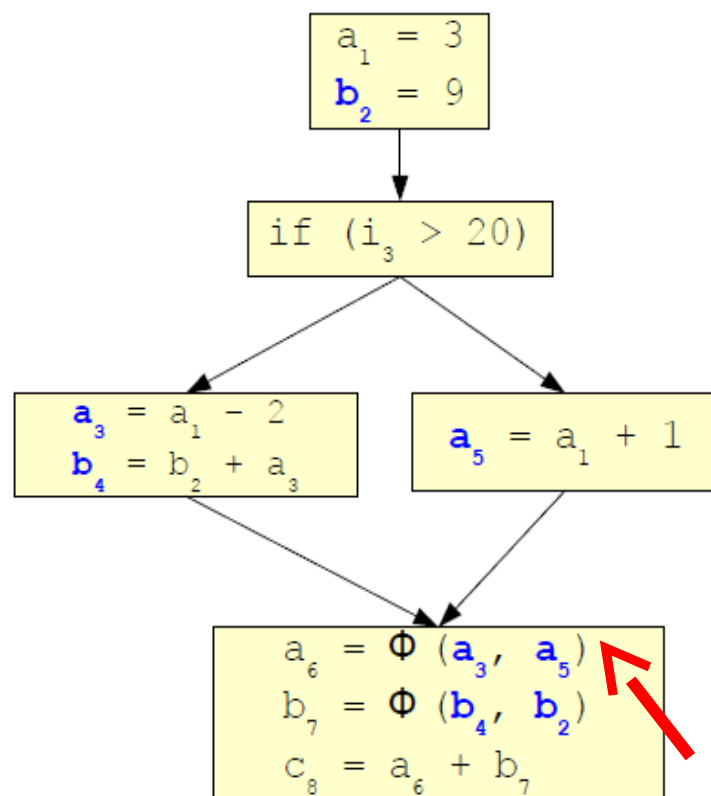
用白話講，在這個表示法當中

- 每個被 Assign 的變數只會出現一次
- 作法 1. 每個被 Assign 的變數給一個
- Version number 2. 使用  $\Phi$  functions

- Example:

- INPUT:  $y = 1, y = 2, x = y$
- SSA:  $y_1 = 1, y_2 = 2, x_1 = y_2$

Diego Novillo, GCC Internals - IRs





# Static Single Assignment, SSA Form

```
int foo (ini n)
{
    int ret;
    if (n > 10)
        ret1 = n * 2;
    else
        ret2 = n + 2;
    return ret?;
}
```

```
int foo (ini n)
{
    int ret;
    if (n > 10)
        ret1 = n * 2;
    else
        ret2 = n + 2;
    ret3 = ? (ret1, ret2)
    return ret3;
}
```

程式中有分歧點會合時  
無法判定是從何而來

此時需要使用  $\Phi$  function  
來處理這種情況，表示值的  
定義需由程式流程決定  
並給予新的版本號

# 使用 SSA 可加强 / 加快以下的最佳化

---

- Constant propagation
- Sparse conditional constant propagation
- Dead code elimination
- Global value numbering
- Partial redundancy elimination
- Strength reduction
- Register allocation

# SSA: Constant Propagation (常數傳遞)

- 有了 SSA，每次 constant propagation 都考 100 分

```
int main()
{
    int a = 11, b = 13, c = 5566;
    int i, result;
    for (i = 0 ; i < 10000 ; i++)
        result = (a*b + i) / (a*c);
    return result;
}
```

gcc3 的 ARM 組合語言輸出結果可見 add 和 bl \_\_divsi3 (呼叫 libgcc 提供的除法函式)，顯然確有計算  $(a*b+i)/(a*c)$  的動作

## GCC-3.x (No-SSA)

```
main:
...
mov     r4, #0
.L5:
mov     r1, #61184
add     r0, r4, #143
add     r1, r1, #42
add     r4, r4, #1
bl      __divsi3
cmp     r4, r5
ble     .L5
...
```

gcc4 卻直接輸出 0，ARM 組合語言相當於 C 語言的 return 0 為什麼呢？因為整數除法裡頭，被除數永遠小於除數，結果必為零

## GCC-4.x (Tree SSA)

```
main:
mov     r0, #0
bx      lr
```

# 解說：切割 Basic Block

```
int main()  
{  
    int a = 11, b = 13, c = 5566;  
    int i, result;  
    for (i = 0 ; i < 10000 ; i++)  
        result = (a*b + i) / (a*c);  
    return result;  
}
```



```
int main () {  
bb0:  
    a = 11  
    b = 13 op  
    c = 5566  
    result = UNDEFINED  
bb1:  
    i = 0  
bb2:  
    if (i < 10000) goto bb5  
    else goto bb3  
bb3:  
    result = (a*b + i) / (a*c)  
bb4:  
    i = i + 1  
    goto bb2  
bb5:  
    return result  
}
```

# 解說：合併 Basic Block

```
int main () {  
bb0:  
    a = 11  
    b = 13 op  
    c = 5566  
    result = UNDEFINED  
bb1:  
    i = 0  
bb2:  
    if (i < 10000) goto bb5  
    else goto bb3  
bb3:  
    result = (a*b+i)/(a*c)  
bb4:  
    i = i + 1  
    goto bb2  
bb5:  
    return result  
}
```



```
int main() {  
bb0:  
    a = 11  
    b = 13  
    c = 5566  
    result = UNDEFINED  
    i = 0  
bb2:  
    if (i < 10000) goto bb5  
    else goto bb3  
bb3:  
    result = (a*b+i)/(a*c)  
    i = i + 1  
    goto bb2  
bb5:  
    return result  
}
```

# 解說：最佳化 CFG

```
int main() {  
bb0:  
    a = 11  
    b = 13  
    c = 5566  
    result = UNDEFINED  
    i = 0  
  
bb2:  
    if (i < 10000) goto bb5  
    else goto bb3  
  
bb3:  
    result = (a*b+i)/(a*c)  
    i = i + 1  
    goto bb2  
  
bb5:  
    return result  
}
```



```
int main() {  
bb0:  
    a = 11  
    b = 13  
    c = 5566  
    result = UNDEFINED  
    i = 0  
    goto bb2  
bb3:  
    result = (a*b + i) / (a*c)  
    i = i + 1  
bb2:  
    if (i < 10000) goto bb3  
    else goto bb5  
bb5:  
    return result  
}
```

# 解說：轉化為 3-address code

```
int main() {  
bb0:  
  a = 11  
  b = 13  
  c = 5566  
  result = UNDEFINED  
  i = 0  
  goto bb2  
bb3:  
  result = (a*b+i)/(a*c)  
  i = i + 1  
bb2:  
  if (i < 10000) goto bb3  
  else goto bb5  
bb5:  
  return result  
}
```



```
int main() {  
bb0:  
  a = 11  
  b = 13  
  c = 5566  
  result = UNDEFINED  
  i = 0  
  goto bb2  
bb3:  
  t0 = a * b  
  t1 = t0 + i  
  t2 = a * c  
  result = t1 / t2  
  i = i + 1  
bb2:  
  if (i < 10000) goto bb3  
  else goto bb5  
bb5:  
  return result  
}
```

# 解說：轉成 SSA form

每個被 **assign** 的變數只會出現一次

- 每次被 **assign** 的變數加上 **version number**

```
int main() {
bb0:
  a = 11
  b = 13
  c = 5566
  result = UNDEFINED
  i = 0
  goto bb2
bb3:
  t0 = a * b
  t1 = t0 + i
  t2 = a * c
  result = t1 / t2
  i = i + 1
bb2:
  if (i < 10000) goto bb3
  else goto bb5
bb5:
  return result
}
```



```
int main() {
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = a0*b0, t1 = t0+i1, t2 =a0*c0
  result1 = t1/t2, i2 = i1+1
bb2:
  i1 =  $\Phi$ (i0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 =  $\Phi$ (result0, result1)
  return result2
}
```

此時需要使用  $\Phi$  function 來處理這種情況，表示值的定義需由程式流程決定，並給予新的版本號



# 解說: Constant Propagation

```
int main() {  
bb0:  
  a0 = 11, b0 = 13, c0 = 5566  
  result0 = UNDEFINED, i0 = 0  
  goto bb2  
bb3:  
  t0 = a0*b0, t1 = t0+i1  
  t2 = a0*c0  
  result1 = t1/t2  
  i2 = i1+1  
bb2:  
  i1 =  $\Phi$ (i0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (result0, result1)  
  return result2  
}
```

傳遞 11, 13, 5566  
等常數到 t0, t2

傳遞常數 0 到 i0



```
int main() {  
bb0:  
  a0 = 11, b0 = 13, c0 = 5566  
  result0 = UNDEFINED, i0 = 0  
  goto bb2  
bb3:  
  t0 = 11*13, t1 = t0+i1  
  t2 = 11*5566  
  result1 = t1/t2  
  i2 = i1+1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (UNDEFINED,  
              result1)  
  return result2  
}
```

# 解說: Constant Folding

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 11*13, t1 = t0+i1
  t2 = 11*5566
  result1 = t1/t2
  i2 = i1+1
bb2:
  i1 =  $\Phi$ (0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 =  $\Phi$ (UNDEFINED,
              result1)
  return result2
}
```



展開常數，也就是  
計算出 11\*13，  
11\*5566 等表示式  
的常數值

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 143, t1 = t0+i1
  t2 = 61126
  result1 = t1/t2, i2 = i1+1
bb2:
  i1 =  $\Phi$ (0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 =  $\Phi$ (UNDEFINED,
              result1)
  return result2
}
```

# 解說: Constant Propagation

```
int main() {  
bb0:  
  a0 = 11, b0 = 13, c0 = 5566  
  result0 = UNDEFINED, i0 = 0  
  goto bb2  
bb3:  
  t0 = 143, t1 = t0+i1  
  t2 = 61126  
  result1 = t1/t2, i2 = i1+1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (UNDEFINED,  
              result1)  
  return result2  
}
```



```
int main() {  
bb0:  
  a0 = 11, b0 = 13, c0 = 5566  
  result0 = UNDEFINED, i0 = 0  
  goto bb2  
bb3:  
  t0 = 143, t1 = 143 + i1  
  t2 = 61126  
  result2 = t1/61126, i2 = i1+1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (UNDEFINED,  
              result1)  
  return result2  
}
```

# 解說: Dead Code Elimination

```
int main() {  
bb0:  
  a0 = 11, b0 = 13, c0 = 5566  
  result0 = UNDEFINED, i0 = 0  
  goto bb2  
bb3:  
  t0 = 143, t1 = 143 + i1  
  t2 = 61126  
  result2 = t1/61126, i2 = i1+1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (UNDEFINED,  
              result1)  
  return result2  
}
```



```
int main() {  
bb0:  
  goto bb2  
bb3:  
  t1 = 143 + i1  
  result1 = t1 / 61126  
  i2 = i1 + 1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 =  $\Phi$ (UNDEFINED,  
              result1)  
  return result2  
}
```

# 解説: Value Range Propagation

```
int main() {  
bb0:  
    goto bb2  
bb3:  
    t1 = 143 + i1  
    result1 = t1 / 61126  
    i2 = i1 + 1  
bb2:  
    i1 =  $\Phi$ (0, i2)  
    if (i1 < 10000) goto bb3  
    else goto bb5  
bb5:  
    result2 =  $\Phi$ (UNDEFINED,  
                result1)  
    return result2  
}
```

result0 = 0 (Treat UNDEFINE as 0)  
i1 = [0, 10000], i2 = [1, 10000]  
t1 = [143, 10143]  
result1 = t1 / 61126  
result2 = 0 or t1 / 61126

result1 = [143, 10143] / 61126  
result2 = 0 or [143, 10143] / 61126

result1 = [143/61126, 10143/61126]  
result2 = 0 or [143/61126, 10143/61126]

result1 = [0, 0]  
result2 = 0 or [0, 0]

result1 = 0  
result2 = 0 or 0

```
int main() {  
bb0:  
    goto bb2  
bb3:  
    t1 = 143 + i1  
    result1 = 0  
    i2 = i1 + 1  
bb2:  
    i1 =  $\Phi$ (0, i2)  
    if (i1 < 10000) goto bb3  
    else goto bb5  
bb5:  
    result2 = 0  
    return result2  
}
```

result2 = 0



# 解說: Constant Propagation

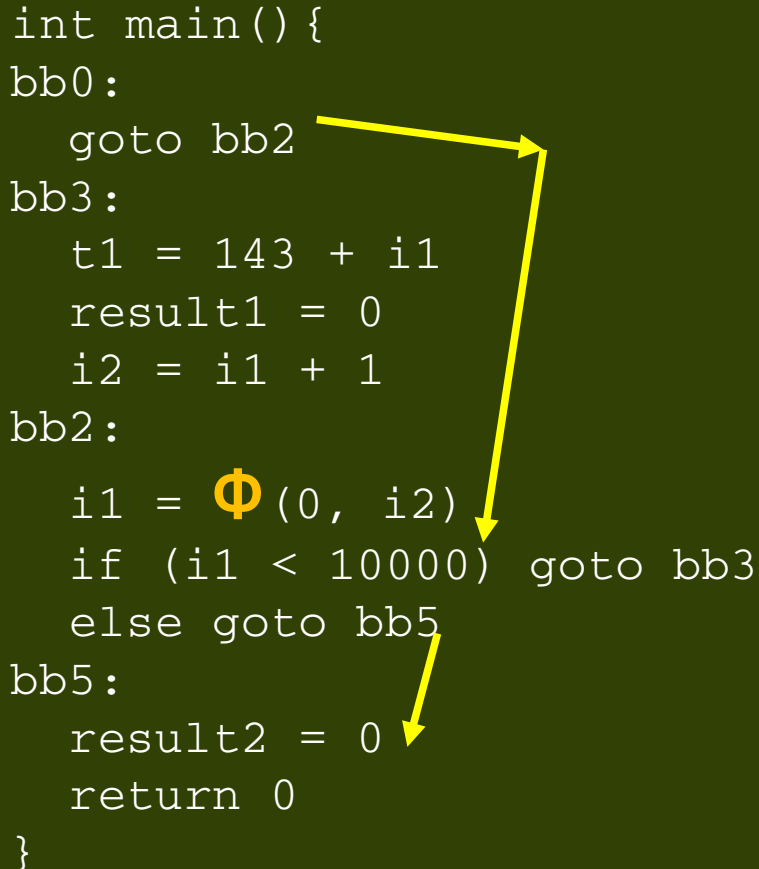
```
int main() {  
bb0:  
    goto bb2  
bb3:  
    t1 = 143 + i1  
    result1 = 0  
    i2 = i1 + 1  
bb2:  
    i1 =  $\Phi$ (0, i2)  
    if (i1 < 10000) goto bb3  
    else goto bb5  
bb5:  
    result2 = 0  
    return result2  
}
```




```
int main() {  
bb0:  
    goto bb2  
bb3:  
    t1 = 143 + i1  
    result1 = 0  
    i2 = i1 + 1  
bb2:  
    i1 =  $\Phi$ (0, i2)  
    if (i1 < 10000) goto bb3  
    else goto bb5  
bb5:  
    result2 = 0  
    return 0  
}
```

# 解說: Dead Code Elimination

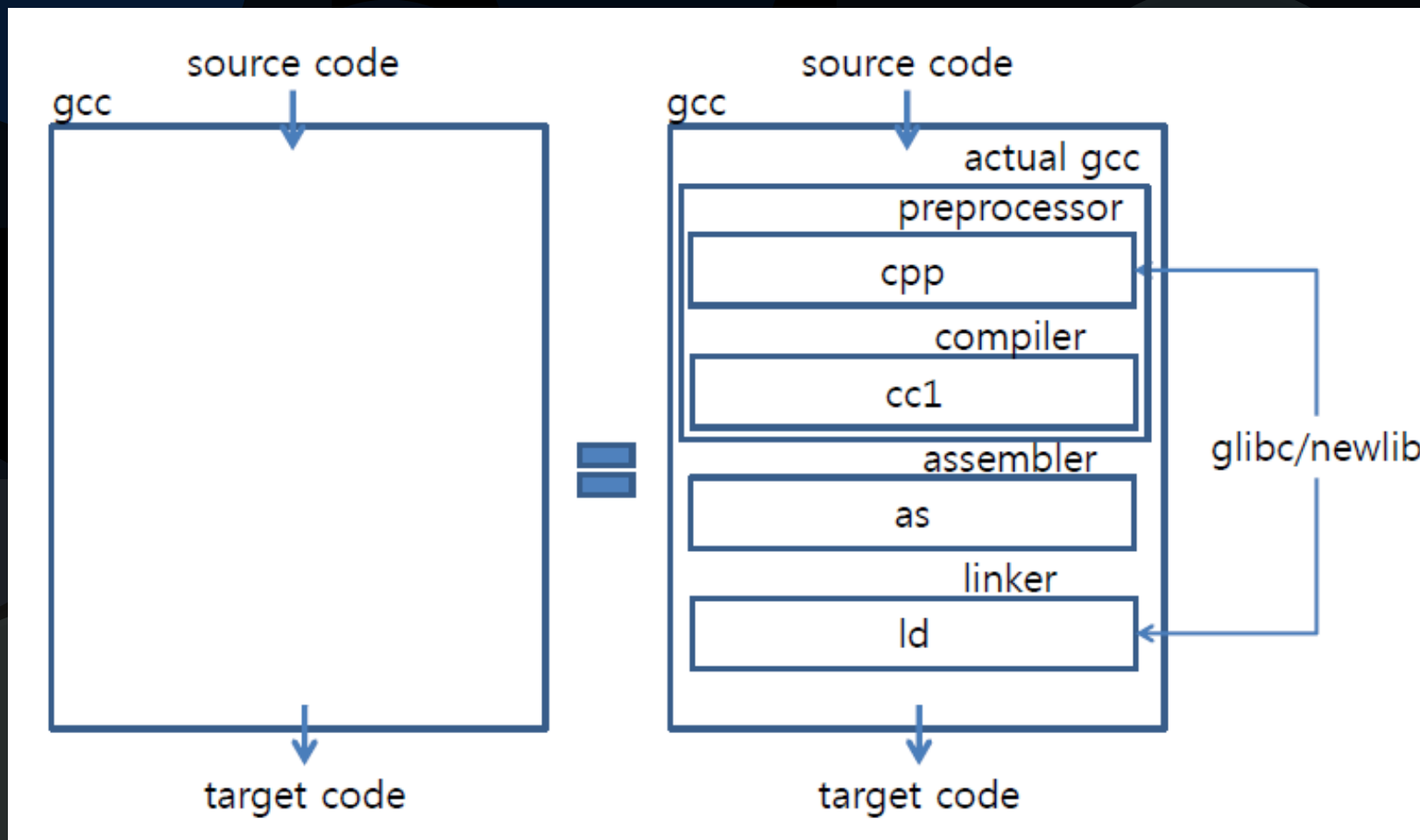
```
int main() {  
bb0:  
  goto bb2  
bb3:  
  t1 = 143 + i1  
  result1 = 0  
  i2 = i1 + 1  
bb2:  
  i1 =  $\Phi$ (0, i2)  
  if (i1 < 10000) goto bb3  
  else goto bb5  
bb5:  
  result2 = 0  
  return 0  
}
```



```
int main() {  
  return 0  
}
```

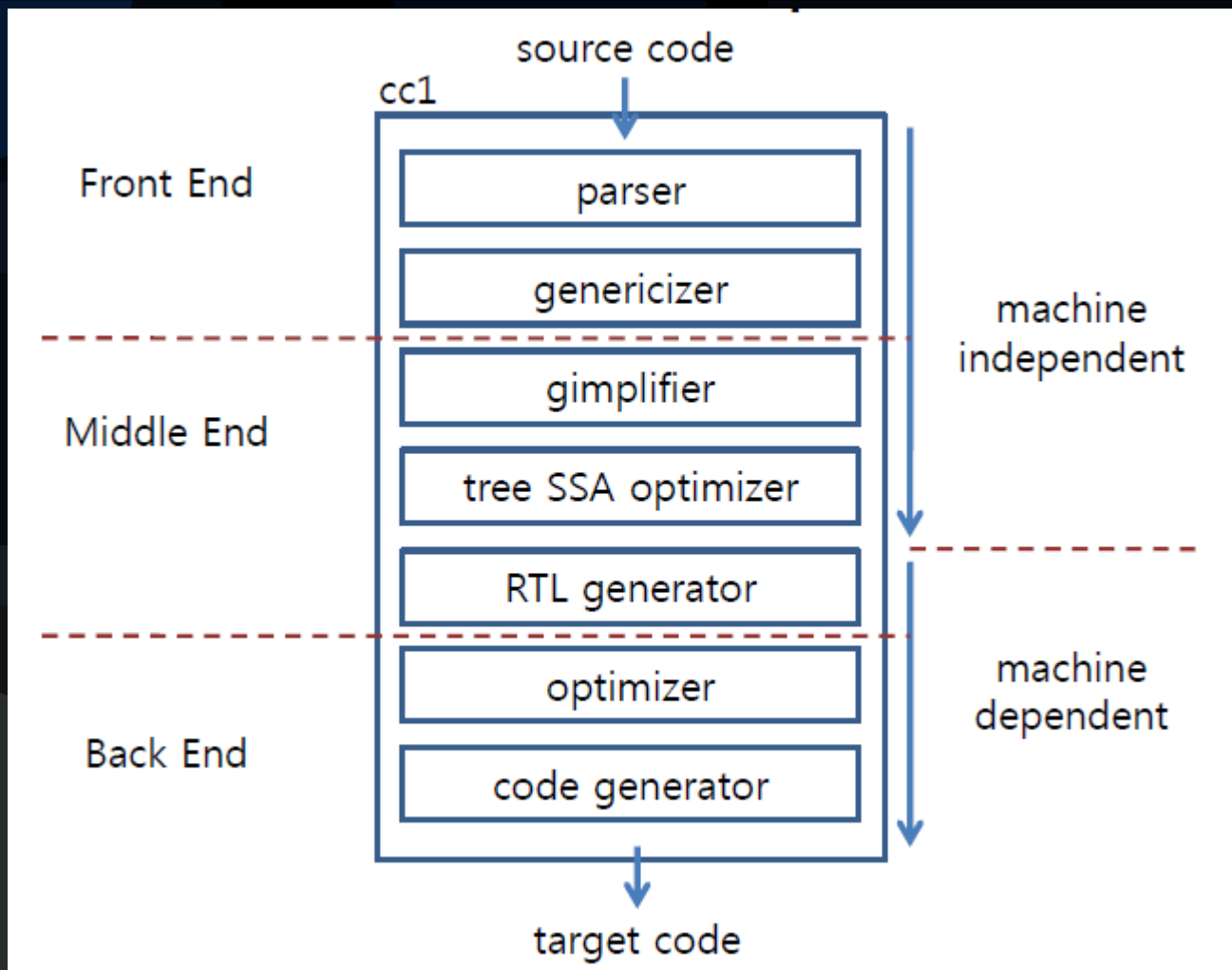


# GCC 結構

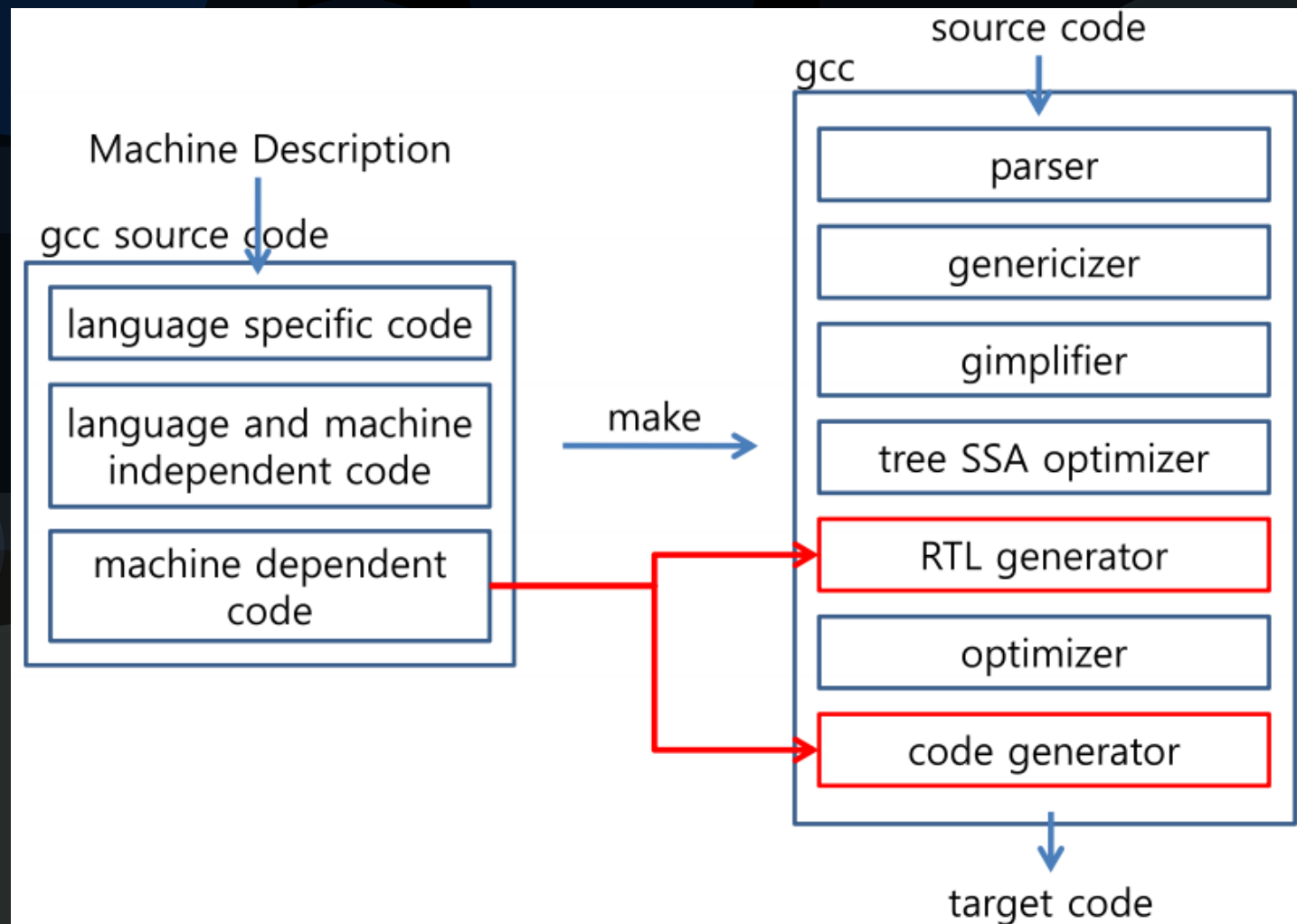




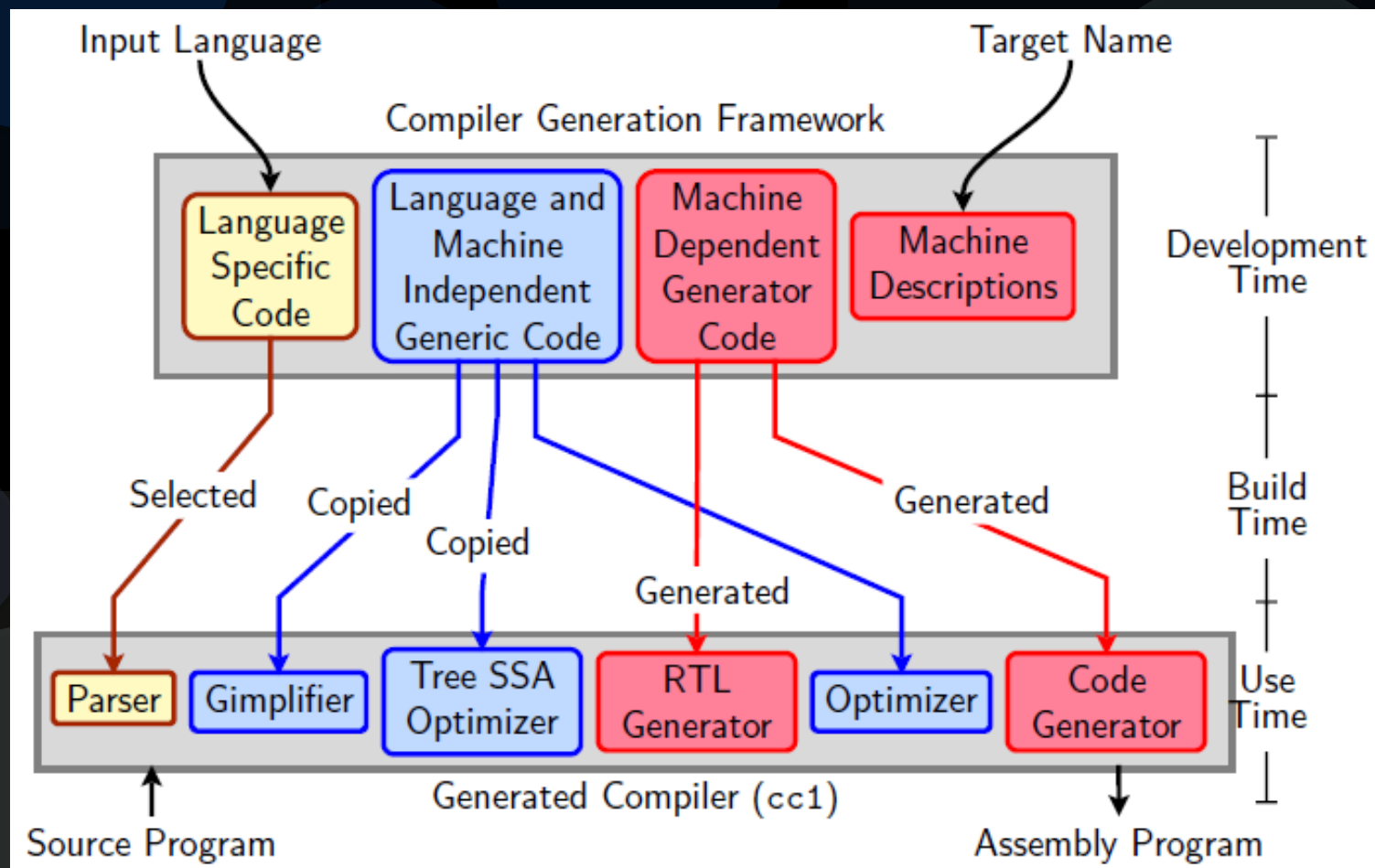
# GCC 結構



# GCC 結構



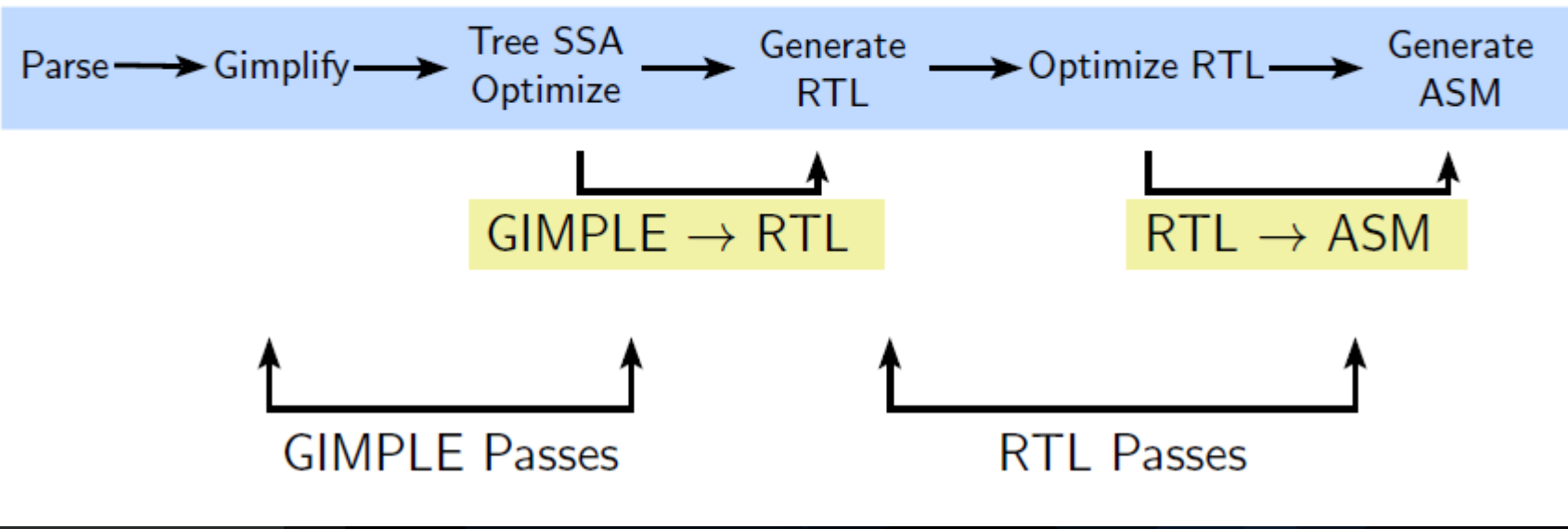
# GCC 結構



# GCC 的 Transformation : 語言間轉換

Transformation from a language to a *different* language

← Target Independent → Target Dependent →



# cc1: 真正的 C 編譯器

1

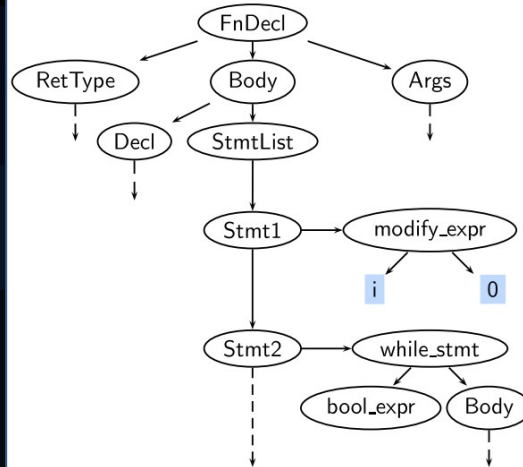
## The Source

```
int f(char *a)
{
    int n = 10; int i, g;

    i = 0;
    while (i < n) {
        a[i] = g * i + 3;
        i = i + 1;
    }
    return i;
}
```

2

## Simplified AST



3

## Gimple IR

```
f (a)
{
    unsigned int i.0; char * i.1;
    char * D.1140; int D.1141;
    ...
    goto <D1136>;
<D1135>;
    ...
    D.1140 = a + i.1; D.1141 = g * i;
    ...
<D1136>;
    if (i < n) {goto <D1135>;}
    ...
}
```

4

## Tree SSA form

```
f (a)
{
    ... int D.1144; ...
<bb 0>:
    n_2 = 10; i_3 = 0;
    goto <bb 2> (<L1>);
<L0>;
    ...
    D.1140_9 = a_8 + i.1_7;
    D.1141_11 = g_10 * i_1;
    ...
<L1>;
    if (i_1 < n_2) goto <L0>; else goto <L2>
    ...
}
```

5

## RTL IR (fragment)

```
(insn 21 20 22 2 (parallel [
  (set (reg:SI 61 [ D.1141 ])
    (mult:SI (reg:SI 66)
      (mem/i:SI
        (plus:SI
          (reg/f:SI 54 ...)
          (const_int -8 ...))))))
  (clobber (reg:CC 17 flags))
]) -1 (nil)
(nil))
```

6

## Final ASM (partial)

```
.file "sample.c"
...
f:
pushl %ebp
...
movl -4(%ebp), %eax
imull -8(%ebp), %eax
addb $3, %al
...
leave
ret
```

# GCC 中端:

## Gimple & Tree SSA Optimizer

---

- **Gimple** 衍生自 **Generic Gimplify**
  - 被限制每個運算只能有兩個運算元 (3-Address IR)  
 $t1 = A \text{ op } B$  (op 即 operator, 像是 + - \* / 等等)
  - 被限制語法只能有某些控制流程
  - **Gimple** 可以被化簡成 **SSA Form**
  - 可使用 `-fdump-tree-<type> -<option>` 來觀看其結構
- **Tree SSA Optimizer**
  - 100+ Passes
  - Loop, Scalar optimization, alias analysis ...etc
  - Inter-procedural analysis, Inter-procedural optimization

# GCC 後端：

## Register Transfer Language (RTL)

$b = a - 56$

```
(set (reg:SI 60 [b])  
      (plus:SI (reg:SI 61 [a])  
                (const_int -56 [0xffffffffc8])))
```

- 類似 LISP 的表示法 (S-expression)
- RTL 使用 virtual Register (具有無限多個 register)
- Register Allocation (Virtual Register -> Hard Register)
- Instruction scheduling (Pipeline scheduling)
- Peephole optimization
- GCC built-in operation and architecture-defined operation

# GCC 後端： RTL Patten Match Engine

MIPS.md

```
(set (reg:SI 60 [b])  
      (plus:SI (reg:SI 61 [a])  
                (const_int -56 [0xffffffc8])))
```

```
(define_insn "*addsi3"  
  [(set (match_operand:GPR 0 "register_operand" "=d,d")  
        (plus:GPR (match_operand:GPR 1 "register_operand" "d,d")  
                  (match_operand:GPR 2 "arith_operand" "d,Q")))]
```

"TARGET\_MIPS16"

"@"

```
addu\t%0,%1,%2  
addiu\t%0,%1,%2"
```

```
[(set_attr "type" "arith")  
 (set_attr "mode" "si")]
```

d 代表是 register  
Q 代表是整數

指令的屬性（用於管線排程）

指令限制（非 MIPS16 才可使用）

b = a - 56

addiu \$2, \$3, -56

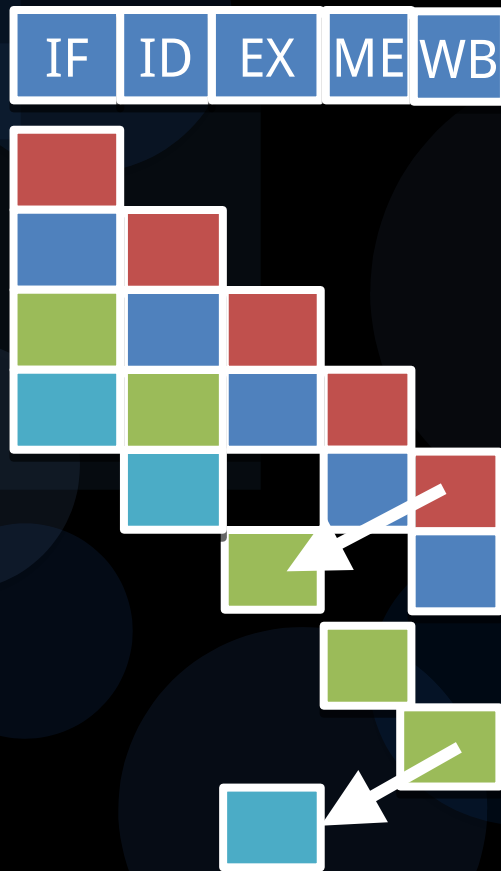


# GCC 後端： 暫存器分配 (Register Allocation)

- 暫存器分配其實是個著色問題 (NP-Complete)
  - 給一個無向圖，相鄰的兩個 **vertex** 不能著同一種顏色
  - 每個 **vertex** 代表的是一個變數，每個 **edge** 代表的是兩個變數的生存時間有重疊
- GCC 的暫存器分配，其實比著色問題更複雜
  - 在有些平台，某些指令其實只能搭配某組的暫存器
  - 其實有些變數是常數，或是 **memory form**，或是他是參數或回傳值，所以只能分配到某些特定暫存器
- Old GCC RA: Reload Pass
  - GCC 把 **pseudo-registers** 根據指令的限制對應到硬體暫存器的過程
  - 其實是個複雜到極點的程式，所以到最後就沒人看得懂了
- 於是後來又重寫
  - IRA (**Integrated** Register Allocator)：整合 **Coalescing**, **Live range splitting** 以及挑選較好的 **Hard Register** 的做法

# GCC 後端：管線排程

- 以經典課本的 5-stage pipeline 為例



Clock 0  
Clock 1  
Clock 2  
Clock 3  
Clock 4  
Clock 5  
Clock 6  
Clock 7  
Clock 8

# GCC 後端：Peephole optimization

- 以管窺天最佳化法
- 掃過整個 IR，看附近  $2 \sim n$  個 IR 有沒有最佳化的機會
  - 感覺起來就像是用奧步
  - 可是如果奧步有用，那就得要好好使用
  - 在某些時候，這個最佳化很好用

```
:: sub rd, rn, #1  
:: cmn rd, #1 (equivalent to cmp rd, #-1)  
:: bne dest
```



```
:: subs rd, rn, #1  
:: bcs dest ((unsigned)rn >= 1)  
  
:: This is a common looping idiom (while (n--))
```

# 進階編譯器最佳化技術

---

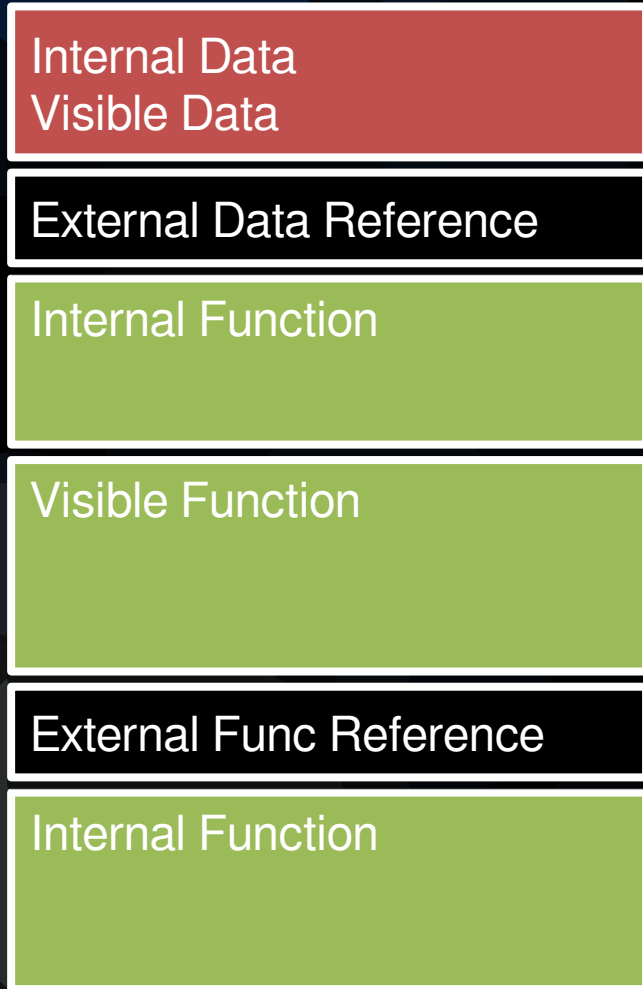
- Loop Optimization
- Inter Procedure Optimization
- Auto Vectorization
- Auto Parallelization

# 由若干個 C 程式原始檔構成專案

- 為什麼要分很多檔案？
  - 因為要讓事情變得簡單
- 為什麼編譯程式時可下 `make -j 24`？
  - 因為編譯器在編譯每個 `.c` 檔案時視為獨立個體，彼此之間沒有相依性
  - 用編譯器的術語稱為 **Compilation Unit**
- **Static Global Variable vs Non-Static Global Variable**
  - **static**: 只有這個 **Compilation Unit** 可以看到這個變數
  - **non-static**: 所有 **Compilation Unit** 可以看到這個變數
    - Q: 怎麼使用別的別的 **Compilation Unit** 內的變數
    - A: 使用 **extern** 關鍵字，如 `extern int i;`

# 深入淺出 Compilation Unit

## Compilation Unit



- **Compilation Unit 產生的限制**
- 關於沒有人使用的 **static global variable**
  - **Compiler** 可以砍掉它來節省空間
- 關於沒有人使用的 **non-static global variable**
  - 編譯器**不能**砍掉它
  - 因為不能確定別的 **Compilation Unit** 會不會用到它
- **注意**: 若確定沒有別的檔案會使用到這個變數或函式, 請宣告成 **static**

# Compilation Unit 盡量使用區域變數

```
static int i;
int main()
{
    for (i = 0 ; i < 10; i++)
        printf ("Hello %d\n", i);
}
```

main:

```
mov    r3, #0
stmfd  sp!, {r4, lr}
movw   r4, #:lower16:.LANCHOR0
movt   r4, #:upper16:.LANCHOR0
mov    r1, r3
str    r3, [r4, #0]
```

.L2:

```
ldr    r0, .L4
bl     printf
ldr    r1, [r4, #0]
add    r1, r1, #1
str    r1, [r4, #0]
cmp    r1, #9
ble   .L2
ldmfd  sp!, {r4, pc}
```

.L4:

```
.word .LC0
```

```
int main()
{
    int i;
    for (i = 0 ; i < 10; i++)
        printf ("Hello %d\n", i);
}
```

main:

```
stmfd  sp!, {r4, lr}
mov    r4, #0
```

.L2:

```
mov    r1, r4
ldr    r0, .L4
add    r4, r4, #1
bl     printf
cmp    r4, #10
bne   .L2
ldmfd  sp!, {r4, pc}
```

.L4:

```
.word .LC0
```

# GCC 內建函式 (Built-in Function)

- GCC 為了作最佳化，會辨認標準函式（因為語意有標準規範），若符合條件，就會以處理器最適合的方式來處理
  - `strcpy` → x86 字串處理指令
  - `printf` → `puts`, `putc`
  - `memcpy` → Block Transfer Instruction
- 這對開發嵌入式系統者常造成困擾
  - 因為往往需要客製化的 `printf`
    - 注意：使用 `-fno-builtin-XXX` 或 `-fno-builtin` 來關閉最佳化



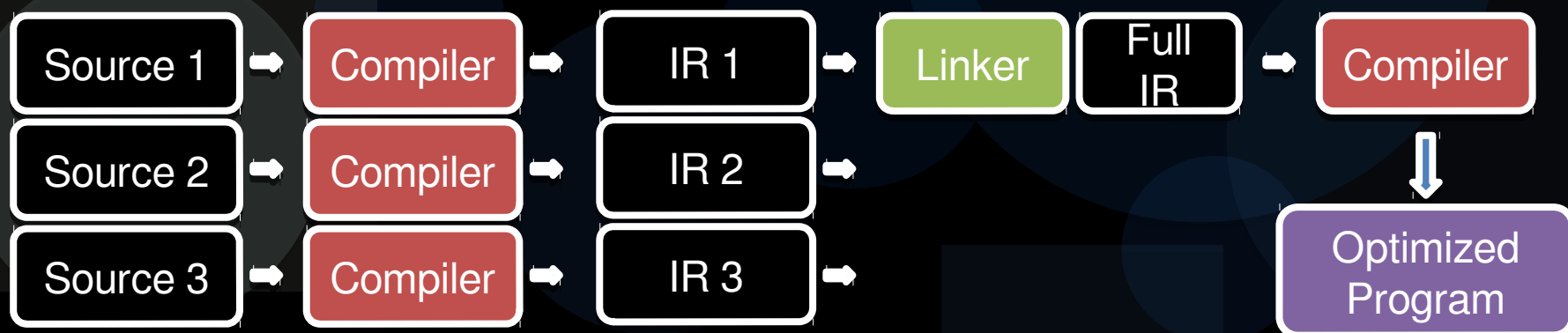
# IPO (Inter-Procedural Optimization)

- **Procedure / Function** : 結構化程式的主軸，增加可用性，減少維護成本
  - 副程式呼叫的執行成本
    - 準備參數：把變數移到到特定 register 或 push 到 stack
    - 呼叫副程式，儲存和回復 register Context
  - 現代化程式：有許多很小的副程式

```
int IsOdd(int num) {  
    return (num & 0x1);  
}
```
- 最簡單的 IPO 就是 function inlining
  - 簡單來說，把整個 function body 複製一份到 caller 的程式碼去

# IPO (Inter-Procedural Optimization)

- 如何 inline 一個外部函式？
  - 基本上編譯器無法做到
    - 因為 compiler 在編譯程式的時候不會去看別的 Compilation Unit 的內容，不知道內容自然就沒辦法 inline 一個看不見的函式
- 解法：Link Time Optimization (LTO)
  - 或叫：Whole-Program Optimization (WPO)
  - 關鍵 Link Time Code Generation



# Linker 也大有學問，特別是配合 LTO

- Linker 的命令列本身就是一種語言
- 個別選項的意義取決於 [ 位置 ] 和 [ 其他部份 ]
- 甚至還有複雜的語法

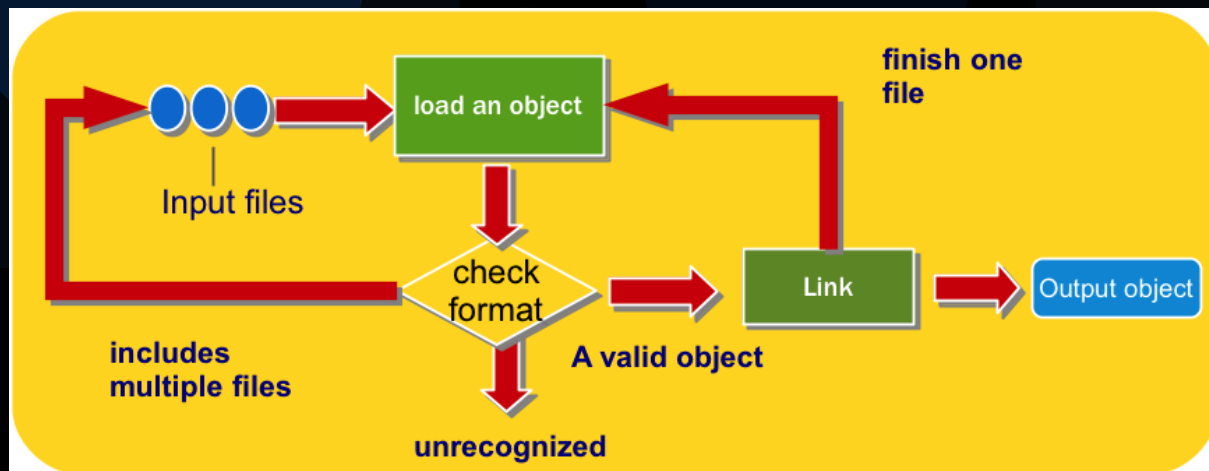
## ■ Four categories of the options

- Input files
- Attributes of the input files
- Linker script options
- General options

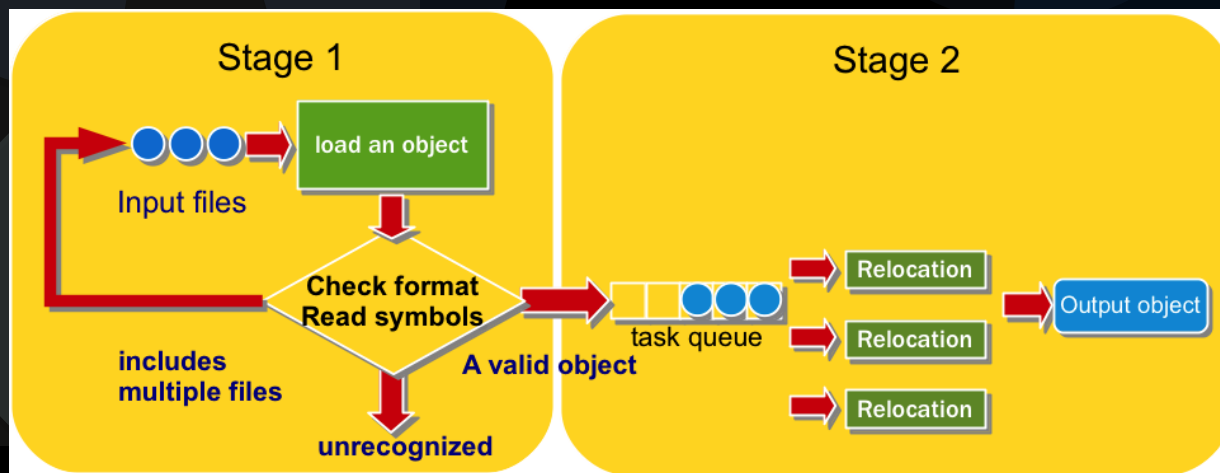
## ■ Examples

```
ld /tmp/xxx.o -lpthread
ld -as-needed ./yyy.so
ld -defsym=cgo13=0x224
ld -L/opt/lib -T ./my.x
```

# GNU ld linker vs. Google gold linker



GNU ld 仰賴 BFD (Binary File Descriptor), 慢且難以維護



Google gold 效率比 GNU ld 快 2 倍。 僅支援 ELF

# 21 世紀的另外一種選擇：LLVM

- LLVM (以前稱為 Low Level Virtual Machine)
  - 用 現代 C++ 寫成的 Compiler infrastructure
  - 設計來進行全時最佳化 (compile-time, link-time, run-time, and idle-time optimizations)
- LLVM 起源
  - 2000 年 UIUC Vikram Adve 與 Chris Lattner 的研究發展而成 (碩士論文)
  - Apple Inc. 採用而大放異彩
  - FreeBSD 10 使用 Clang/LLVM 為預設的編譯器



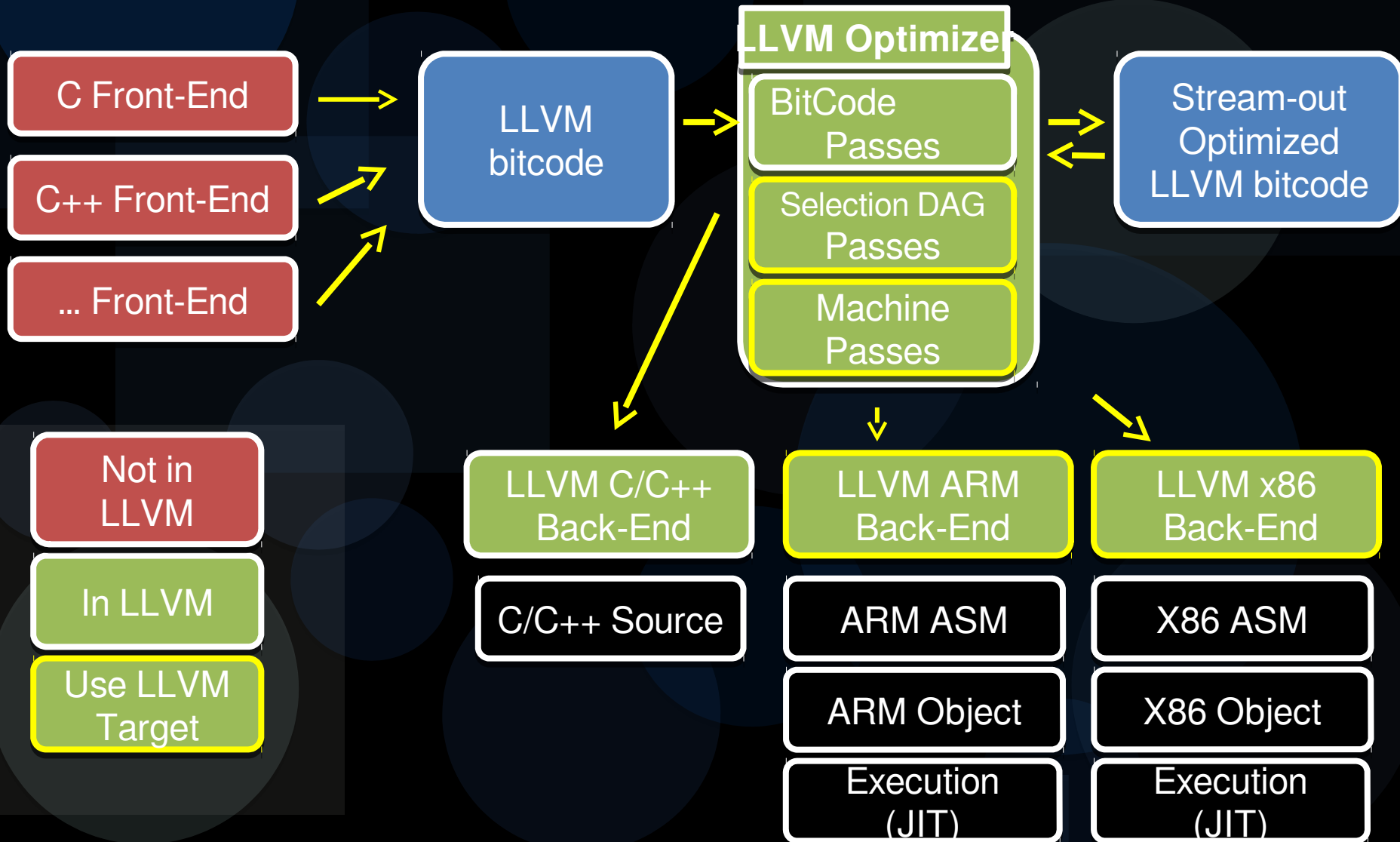
# LLVM 核心觀念：

## LLVM is a Compiler IR

---

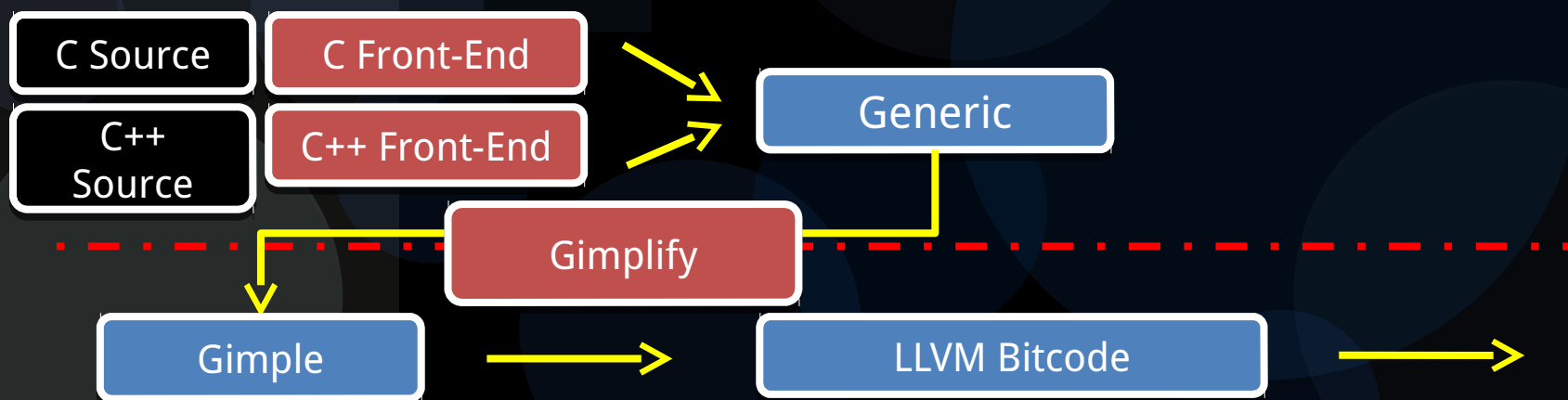
- 既然 Compiler framework 要支援多種 front-end 和多種 back-end
  - 使用類似 RISC 平台無關的指令 (LLVM Instruction) 和資料 (Metadata) 來表達原始碼
- LLVM IR 的三變化 ( 等價形式 )
  - In-Memory Form: JIT
  - Assembly language representation: ASM format (.ll)
  - Bitcode representation: Object code format (.bc)
- 特點
  - Self-contained Representation 且有良好規範的 Compiler IR
  - 可以將編譯器處理到一半的結果序列化

# LLVM 的架構：從原始碼到二進制



# LLVM Toolchain (llvm-gcc)

- 既然 LLVM 只是個 Compiler-IR，就代表 LLVM 在剛開始的時候是沒有辦法走完整個編譯流程的
- LLVM-GCC (gcc-4.2)
  - 利用 GCC 已經存在而且衆多大眾使用的 Front-end
  - 後來 Richard Stallman 反對此方式 (GCC 4.3, GPLv3)
  - 後來 Apple Inc. 就跳出來發展自己的 front-end (Clang)





# LLVM Toolchain (Clang)

- Clang : C, C++, Objective-C 的編譯器前端
  - Apple 為了取代 GCC 重新打造的 Frontend
  - 重寫一個前端代價是非常昂貴的
    - GNU/Linux 世界使用了 GCC 行之有年的非標準語法
  - Clang 設計成模組化 Frontend : Clang C API
    - 可 export AST (如 Android RenderScript 來產生 Java 的 Binding)
    - 可用來實作自動完成、Refactor Tool 等等
    - 可用來實作靜態程式碼分析工具 (Linting tool 等等)
  - Clang 改善了錯誤訊息 (expressive diagnostics)
  - 快速編譯，低記憶使用量，容易學習和延展的 Frontend

# 範例：Clang Expressive Diagnostics

```
struct a {  
    virtual int bar();  
};  
struct foo : public virtual a  
{  
};  
void test(foo *P) {  
    return P->bar() + *P;  
}
```

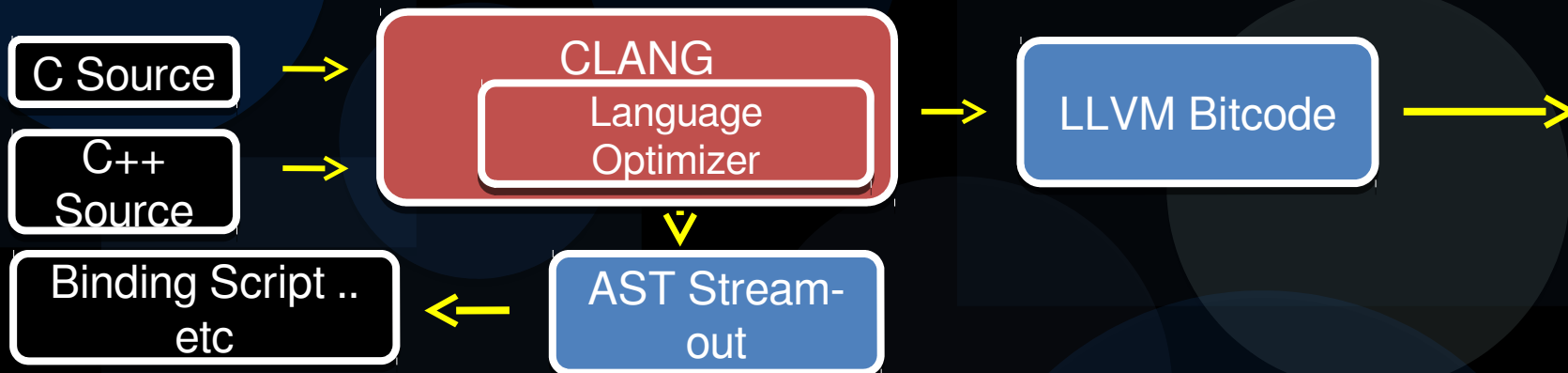
g++-4.7

```
xx.cc: In function 'void test(foo*)':  
xx.cc:9:24: error: no match for 'operator+' in '(((a*)P) + ((sizetype)*((long int*)(P->foo::<anonymous>.a::_vptr.a + -32u))))->a::bar() + * P'  
xx.cc:9:24: error: return-statement with a value, in function returning 'void' [-fpermissive]
```

clang-3.1

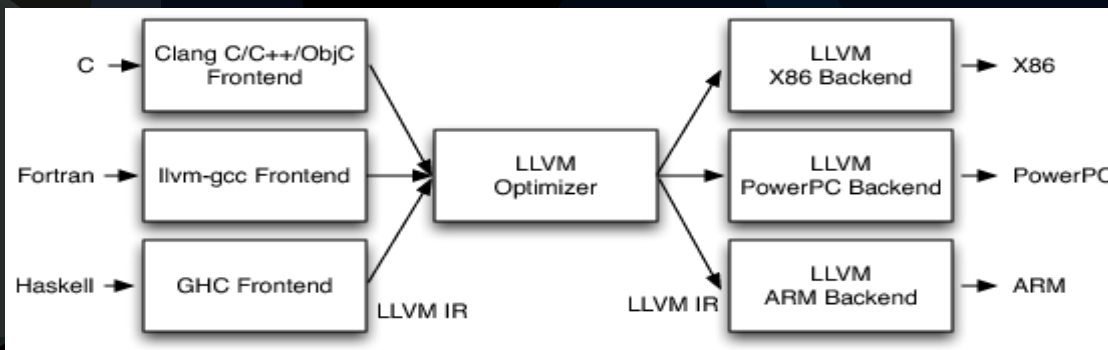
```
xx.cc:9:21: error: invalid operands to binary expression ('int' and 'foo')  
    return P->bar() + *P;  
                   ~~~~~ ^ ~
```

# Clang/LLVM Toolchain



- Clang Compiler Driver

- 盡可能和 GCC 相容，遇到不認識的 Option 不會停下來
- 不需要 Cross Compiler，本身就有 Cross Compiler 的能力
  - Cross Compiler != Cross Compiler Toolchain



# LLVM TableGen (Target Description)

Instruction Encoding (For MC Assembler)

Input/Output

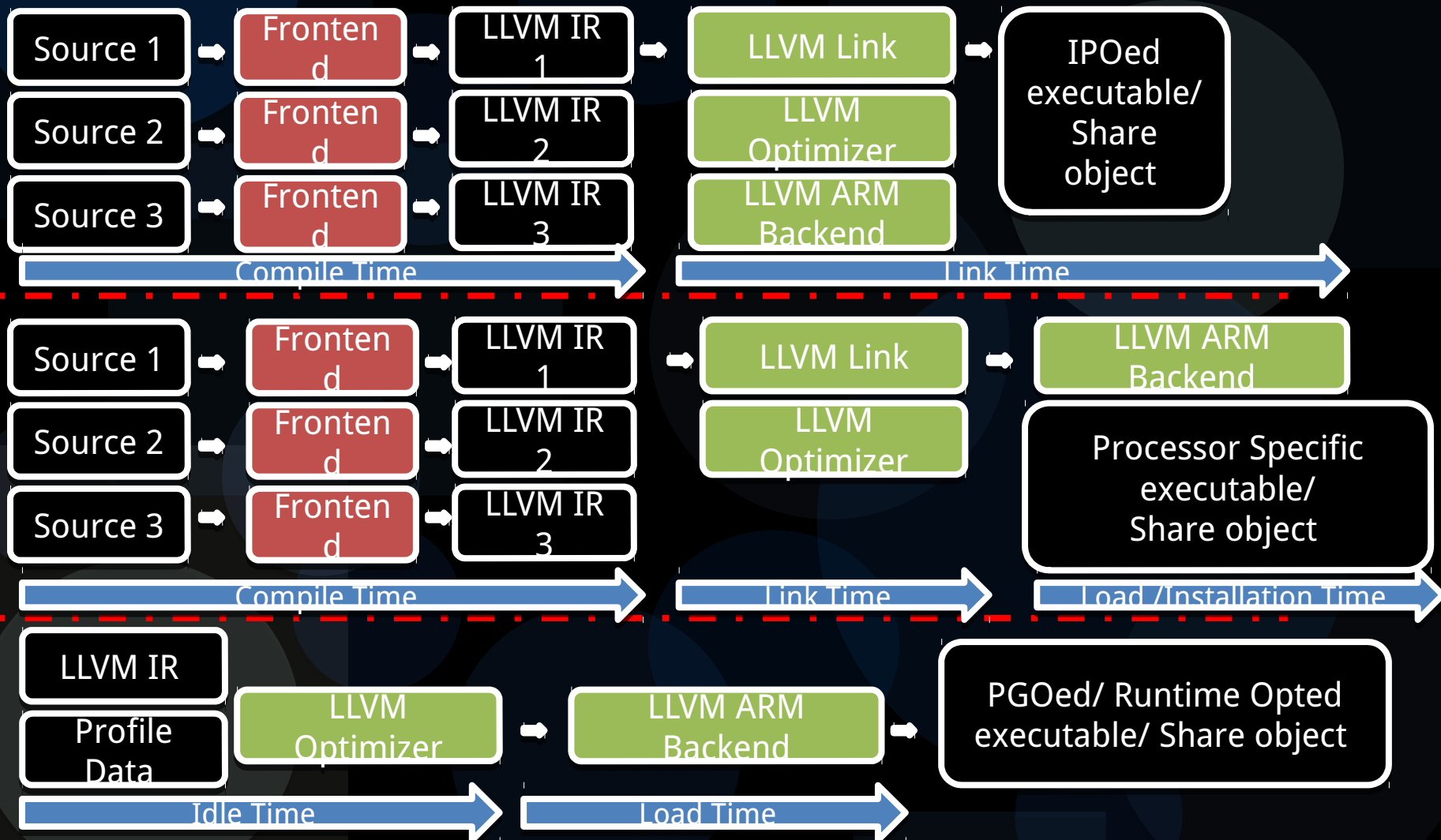
Assembly (For ASMPrinter)

```
def ADDPS : PI<0x58,  
(outs VR256:$dst),  
(ins VR256:$src1, VR256:$src2),  
"addps $src2, $src1, $dst",  
[(set VR256:$dst, (fadd VR256:$src1, VR256:$src2))]>;
```

Instruction Selection Pattern  
(For Selection DAG)

- 編譯器 Backend 常會引入特別的程式語言來表示後端機械碼生成
  - GCC RTL Pattern : (MD, Machine Description file)
  - LLVM TableGen : (TD, Target Description file)

# 編譯器進化論：LLVM 的多變化



# LLVM 的發展

---

- 最美好的時代，最壞的時代
- LLVM Bitcode 用來當傳遞格式還有很多問題
  - Binary Compatibility 最為人所詬病
  - <http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20120521/143197.html>
- CASE STUDY: OpenCL SPIR
  - The OpenCL Khronos group proposes to extend LLVM to be the standard OpenCL IR (called SPIR)
    - [http://www.khronos.org/registry/cl/specs/spir\\_spec-1.0-provisional.pdf](http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf)
  - Khronos SPIR For OpenCL Brings Binary Compatibility
    - [http://www.phoronix.com/scan.php?page=news\\_item&px=MTE4MzM](http://www.phoronix.com/scan.php?page=news_item&px=MTE4MzM)

# 參考資料

---

- <http://www.redhat.com/magazine/002dec04/features/gcc/>
- <http://gcc.gnu.org/onlinedocs/gcc>
- [http://en.wikipedia.org/wiki/Interprocedural\\_optimization](http://en.wikipedia.org/wiki/Interprocedural_optimization)
- [http://en.wikipedia.org/wiki/Link-time\\_optimization](http://en.wikipedia.org/wiki/Link-time_optimization)
- <http://llvm.org/>
- <http://www.aosabook.org/en/llvm.html>
- Compiling a Compiler, 穆信成
- GNU Compiler Collection, Kyungtae Kim