LLVM 總是打開你的心：
從電玩模擬器看編譯器應用實例

Jim Huang (黃敬群) <jserv@0xlab.org>

Jul 11, 2013 / 新竹碼農

Rights to copy

© Copyright 2013 **0xlab**

http://0xlab.org/

contact@0xlab.org

creative commons
COMMONS DEED

Attribution – ShareAlike 3.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Corrections, suggestions, contributions and translations are welcome!

Latest update:Jul 22, 2013

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

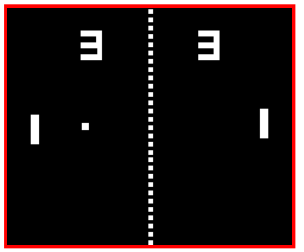
Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

姊妹議程：
〈窮得只剩下 Compiler〉
OSDC.tw 2009

http://www.slideshare.net/jserv/what-can-compilers-do-for-us

遊戲產業加速技術革新



1972 Pong (硬體)

1980 Zork (高階直譯語言)

1993 DOOM (C)

1998 Unreal (C++, Java-style scripting)

2005-6 Xbox 360, PlayStation 3
with 6-8 hardware threads

2009 Next console generation
Unification of the CPU, GPU.
Massive multi-core, data parallelism, etc.

目標：重新編譯任天堂遊戲為 Native Game

http://andrewkelley.me/post/jamulator.html

提綱

(1) 任天堂遊戲機概況

(2) Code Generation: 使用 LLVM

(3) Optimizations & LLVM

(4) 建構 NES 執行環境

任天堂遊戲機

NES (Nintendo Entertainment System)

任天堂電玩主機歡度誕生 30 周年！

http://gnn.gamer.com.tw/6/82966.html

- 1983 年 7 月 15 日在日本推出、陪伴許多 6 、 7 年級玩家度過童年時光的任天堂 Family Computer(簡稱 FAMICOM / FC ，台灣俗稱任天堂紅白機) 電視遊樂器主機，歡度誕生 30 周年紀念
- 因為《超級瑪利歐兄弟》等眾多人氣經典遊戲的推出而帶動了爆發性的銷售
- 自 1983 年推出到 2003 年停產為止，全球總計出貨 6291 萬台

NES 硬體簡述

- 四個獨立運作的單元
  - CPU: 8-bit processor - MOS 6502.
  - Graphics: Picture Processing Unit (PPU)
  - Sound: Audio Processing Unit (APU)
  - (Memory) Mapper: 提供額外 ROM 的客製化硬體

Source: https://en.wikipedia.org/wiki/Nintendo_Entertainment_System

NES 硬體定址空間

- NES 使用 MMIO (memory-mapped I/O)，可藉由讀寫特定的記憶體位址，存取到週邊硬體
- 定址空間為 64 KB，所有位址可用 2 bytes 表示

起始位址	結束位址	描述
$0000	$07FF	RAM: 除非使用 Mapper，否則 NES 遊戲只有 2K 的 RAM 空間
$2000	$2007	保留給 PPU 使用，此區段的記憶體內容將會在螢幕上顯示
$4000	$4015	保留給 APU 使用，作為音效輸出
$4016	$4017	遊戲控制狀態的保存
$8000	$FFFF	唯讀記憶體，遊戲本身載入於此區段 [code+data]

$ 表示 16 進位

6502 程式組譯後成為 32 KB binary，將在系統啓動後，被載入到 $8000 - $FFFF 的範圍

撰寫 Hello World! 程式

- 在 NES 開發程式的門檻不低，即便是 Hello World 等級的程式
- 難度在於得自行處理以下
  - 硬體週邊
  - Interrupt
    - Reset
    - IRQ: 以 CLI/SEI 指令控制觸發與否
    - NMI (Non-Maskable Interrupt): 無硬體指令可控制此類；vertical blank 起始時，會觸發 NMI
- 定義自訂的 ABI，以簡化處理

位址	描述
$2008	Write a byte to this address and the byte will go to standard out.
$2009	Write a byte to this address and the program will exit with that byte as the return code.

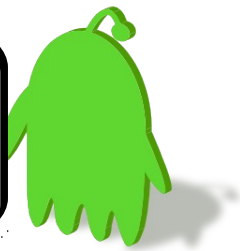
```
org $C000

msg: .data "Hello, world!", 10, 0

Reset_Routine:

LDX #$00        ; put the starting index, 0, in register X

loop: LDA msg, X      ; read 1 char
BEQ loopend     ; end loop if we hit the \0
STA $2008       ; putchar (custom ABI)
INX             ; (Register X)++
JMP loop        ; repeat

loopend:
LDA #$00        ; return code 0
STA $2009       ; exit (custom ABI)

IRQ_Routine: rti ; do nothing
NMI_Routine: rti ; do nothing

org    $FFFA
dc.w   NMI_Routine
dc.w   Reset_Routine
dc.w   IRQ_Routine
```

code/data 將自 $C000 位址開始，所以 "Hello, World!\n\0" 字串的 W 字元將被載入到位址 $C007

若 zero flag 為 1 ，跳往 loopend 標記，否則移往下一道指令

將暫存器 A 的內容存到記憶體位址 $2008 ，這意味著，將暫存器 A 的值輸出到 stdout

NES 程式僅有 32 KB ，這指令告知組譯器，填寫空白直到位址 $FFFA



```
                ; memcpy --
                ; Copy a block of memory from one location to another.
                ;
                ; Entry parameters
                ;       SRC - Address of source data block
                ;       DST - Address of target data block
                ;       CNT - Number of bytes to copy
0040                            ORG     $0040       ;Parameters at $0040
0040  00 00     SRC            DW      $0000
0042  00 00     DST            DW      $0000
0044  00 00     CNT            DW      $0000

0600                            ORG     $0600       ;Code at $0600
0600  A4 44     MEMCPY         LDY     CNT+0       ;Set Y = SRC.L
0602  D0 05                    BNE     LOOP        ;If SRC.L > 0, then loop
0604  A5 45                    LDA     CNT+1       ;If CNT.H > 0,
0606  D0 01                    BNE     LOOP        ; then loop
0608  60                       RTS                 ;Return
0609  B1 40     LOOP           LDA     (SRC),Y     ;Load A from ((SRC)+Y)
060B  91 42                    STA     (DST),Y     ;Store A to ((DST)+Y)
060D  88                       DEY                 ;Decr CNT.L
060E  D0 F9                    BNE     LOOP        ;if CNT.L > 0, then loop
0610  E6 41                    INC     SRC+1       ;Incr SRC += $0100
0612  E6 43                    INC     DST+1       ;Incr DST += $0100
0614  88                       DEY                 ;Decr CNT.L
0615  C6 45                    DEC     CNT+1       ;Decr CNT.H
0617  D0 F0                    BNE     LOOP        ;If CNT.H > 0, then loop
0619  60                       RTS                 ;Return
061A                            END
```

Source: https://en.wikipedia.org/wiki/MOS_Technology_6502

程式開發

- 在 NES 遊戲中，記憶體區段 $FFFA 到 $FFFF 有特別含意
- 當遊戲啓動時， NES 會從 $FFFC 與 $FFFD 讀取 2 bytes ， 並使用其內含值，決定第一道執行指令的位址
- 6502 暫存器定義

名稱	寬度	描述
A	8-bit	"main" register. Most of the arithmetic operations are performed on this register.
X	8-bit	Another general purpose register. Fewer instructions support X than A. Often used as an index, as in the above code.
Y	8-bit	Pretty much the same as X. Fewer instructions support Y than X. Also often used as an index.
P	8-bit	The "status" register. When the assembly program mentions the "zero flag" it is actually referring to bit 1 (the second smallest bit) of the status register. The other bits mean other things, which we will get into later.
SP	8-bit	The "stack pointer". The stack is located at $100 - $1ff. The stack pointer is initialized to $1ff. When you push a byte on the stack, the stack pointer is decremented, and when you pull a byte from the stack, the stack pointer is incremented.
PC	16-bit	The program counter. You can't directly modify this register but you can indirectly modify it using the stack.

用 Go 語言寫工具程式

- Assembler
  - lexer
  - asm6502.nex → asm6502.nn.go ; asm6502.y → y.go
- Disassembler
  - 首先面對的問題：如何區分 code 與 data?

```
org    $FFFA
dc.w   NMI_Routine
dc.w   Reset_Routine
dc.w   IRQ_Routine
```

  - 建立一個 AST，使每個單一 byte 都是個 .db 敘述
  - 對於 $FFFA 到 $FFFB 範圍的記憶體區間，若有參照 NMI_Routine 標籤的話，以 .dw 敘述替換掉 .db 敘述
  - 計算 $FFFA 到 $FFFB 之間有參照的 .db 敘述，並適當地該敘述前，安插 NMI_Routine
  - 視該位址的 .db 敘述為一道指令

設計 Disassembler，分析遊戲運作

- Disassembler: 如何區分 code 與 data?
- 解析機械碼指令的策略

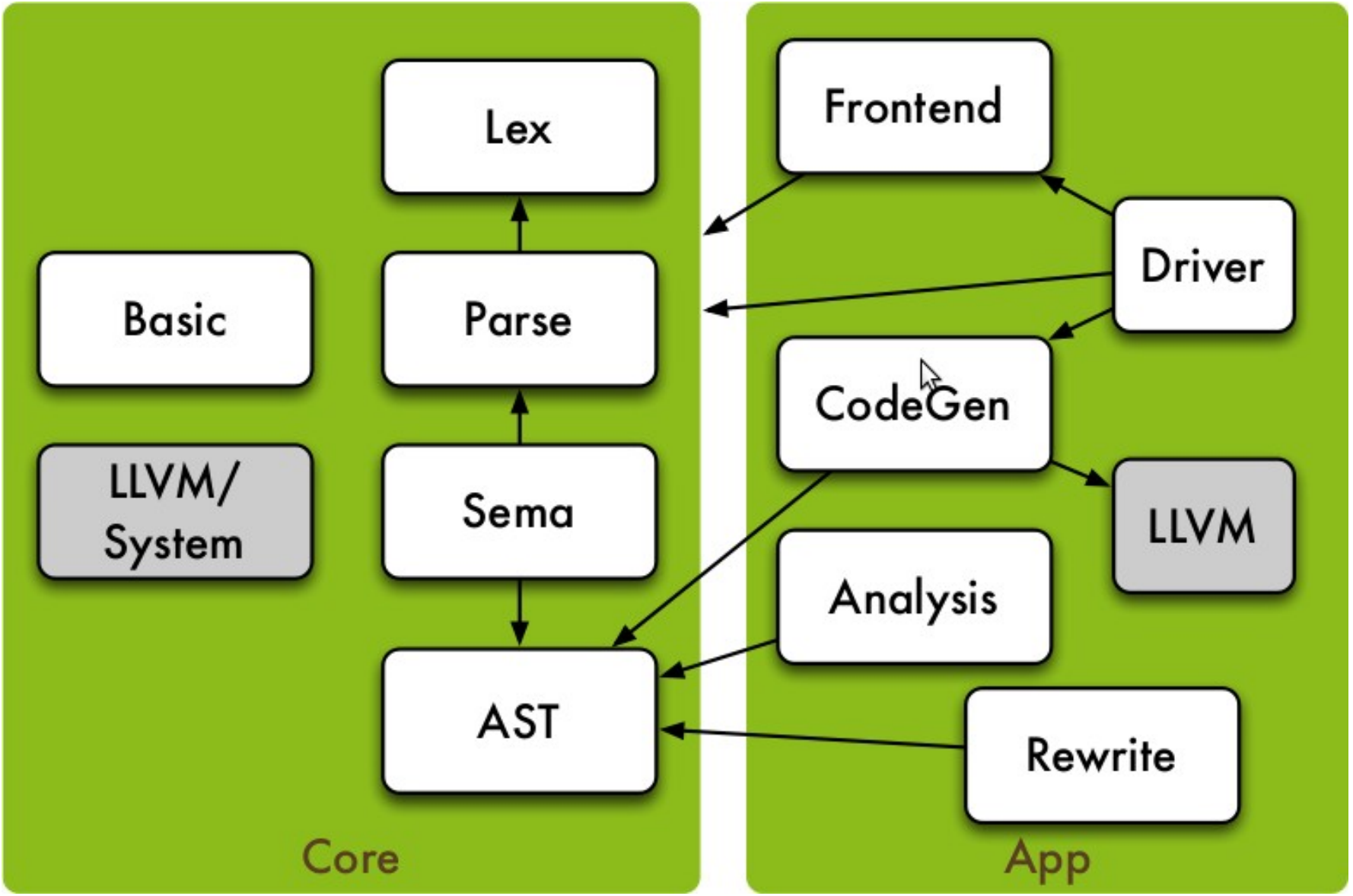
指令	策略
`BPL, BMI, BVC, BVS, BCC, BCS, BNE, BEQ, JSR`	Mark the jump target address and the next address as an instruction.
`JMP absolute`	Mark the jump target address as an instruction

- 以 B 字母開頭的指令，表示 conditional branch;
- JSR = Jump to SubRoutine; RTS = ReTurn from Subroutine

Code Generation: 使用 LLVM

LLVM

- bit-code
- 完整的編譯器基礎建設
  - 可重用的、用以建構編譯器的軟體元件
  - 允許更快更完整的打造新的編譯器
  - static compiler, JIT, trace-based optimizer, ...
- 開放的編譯器框架
  - 多種程式語言支援
  - 高彈性的自由軟體授權模式 (BSD License)
  - 活躍的開發
  - 豐富的編譯輸出： ARM, x86, MIPS, GPU, ...


Lex
Frontend
Basic
Parse
Driver
CodeGen
LLVM/
System
Sema
LLVM
Analysis
AST
Rewrite
Core
App

Frontend LLVM IR Backend

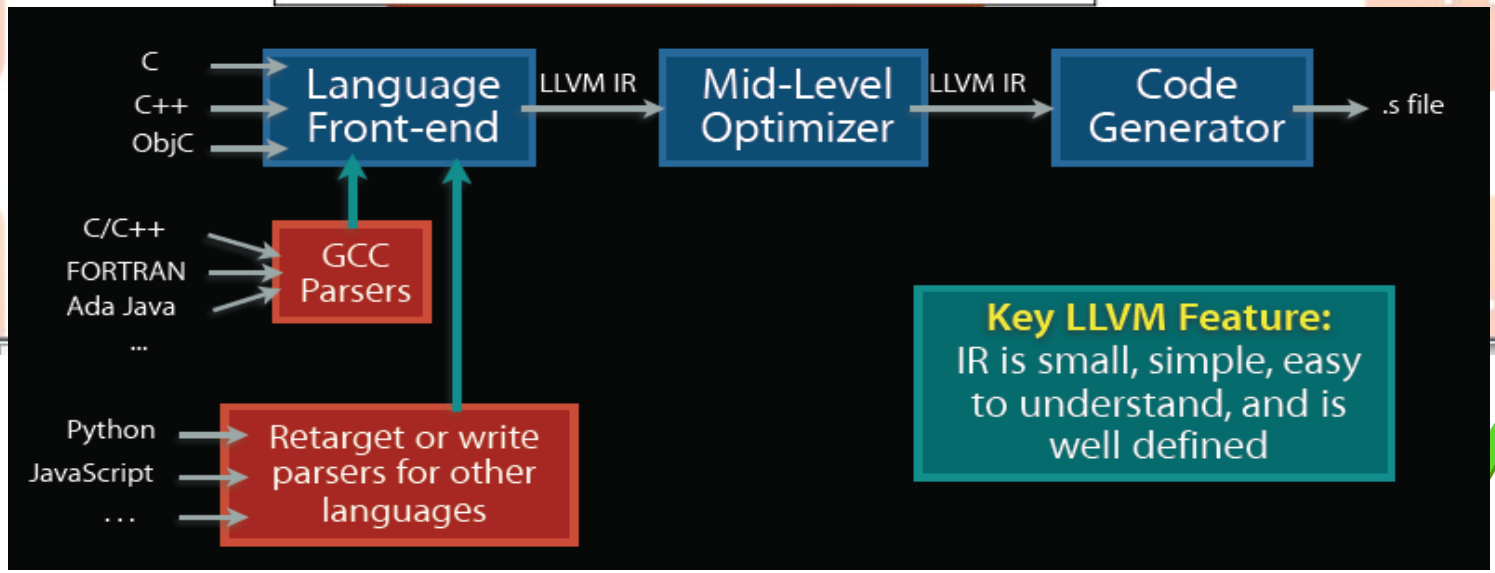
C/C++

x86

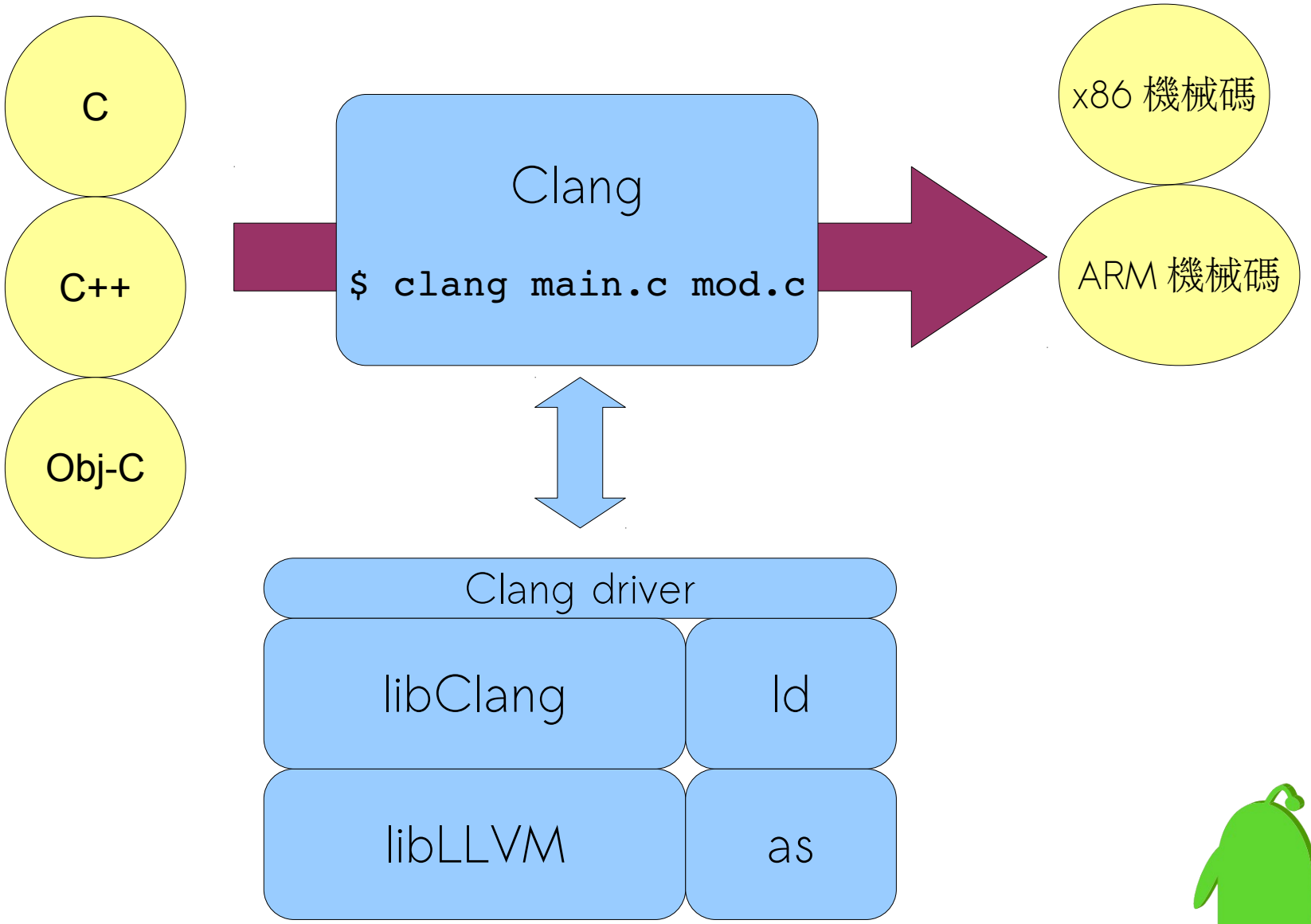
```
int add_func(
  int a, int b)
{
  return a + b;
}
```

```
define i32 @add_func(i32 %a, i32 %b) {
entry:
     %tmp3 = add i32 %b, %a
     ret i32 %tmp3
}
```

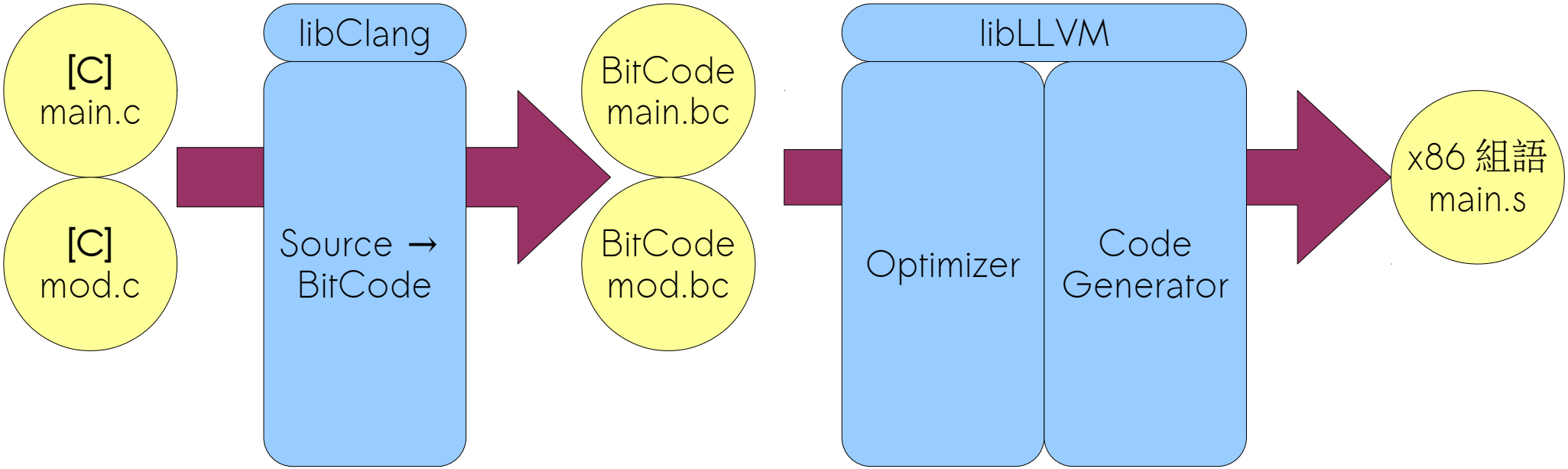
```
_add_func:
  movl  8(%esp), %eax
  addl  4(%esp), %eax
    ret
```



Clang: LLVM 的程式語言前端

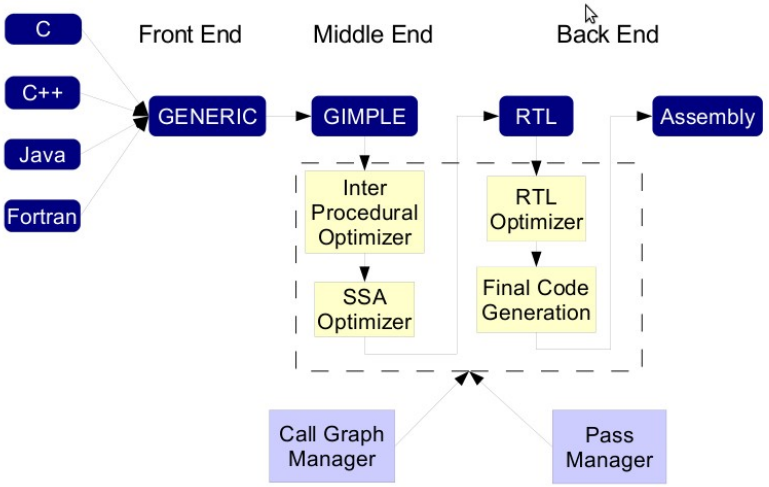


Clang 與 LLVM 的關聯

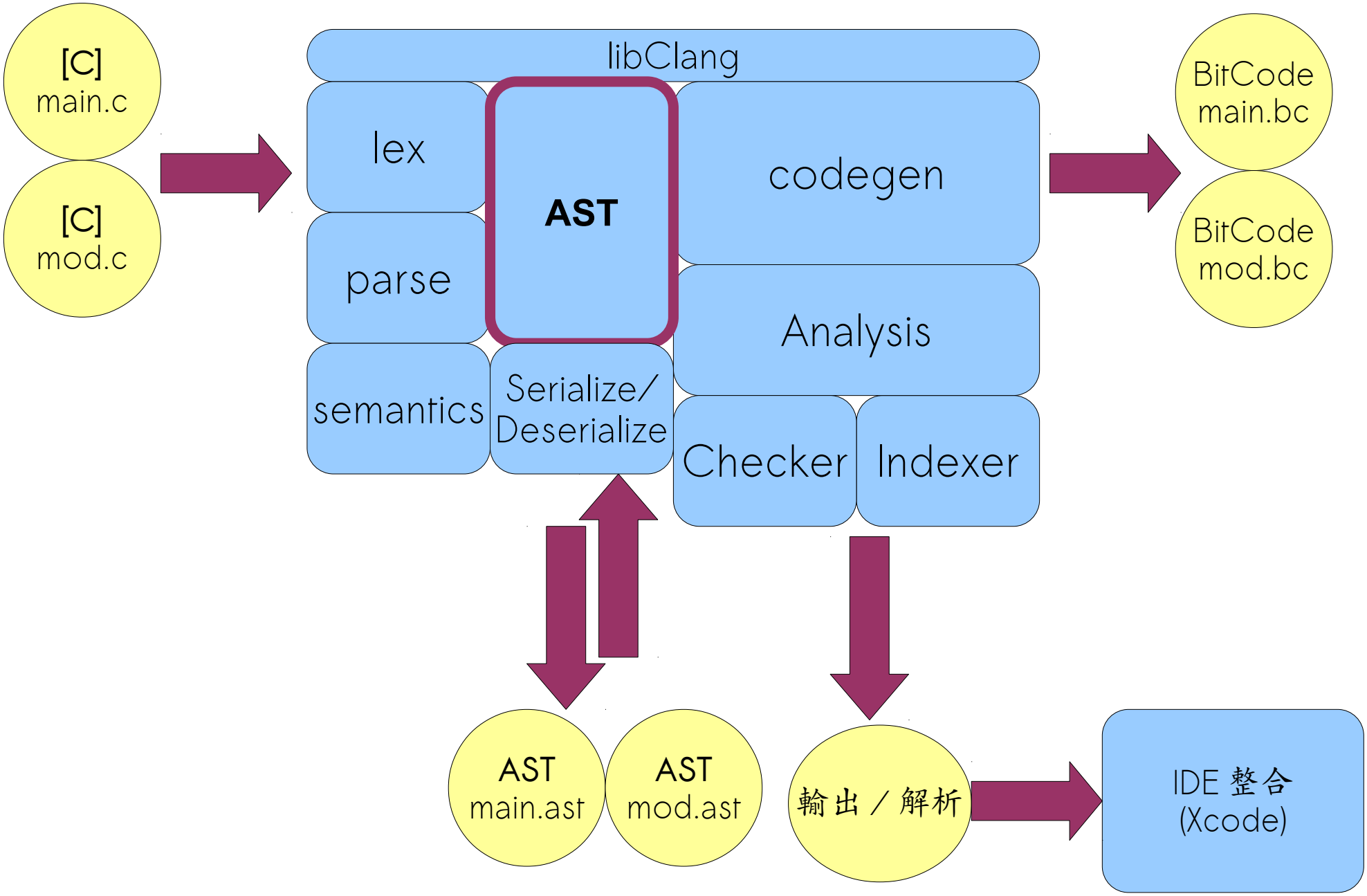


Compiler pipeline

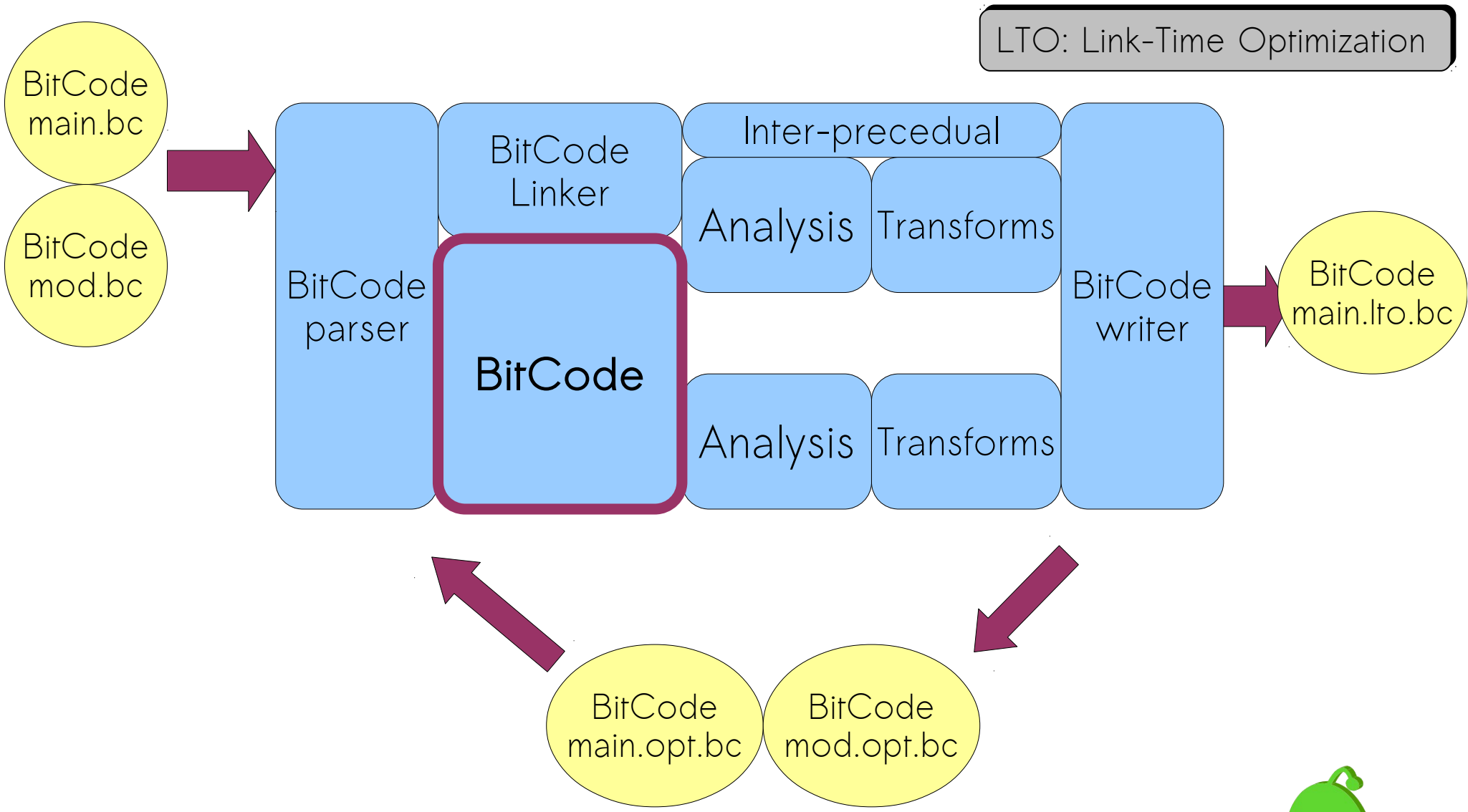
Google



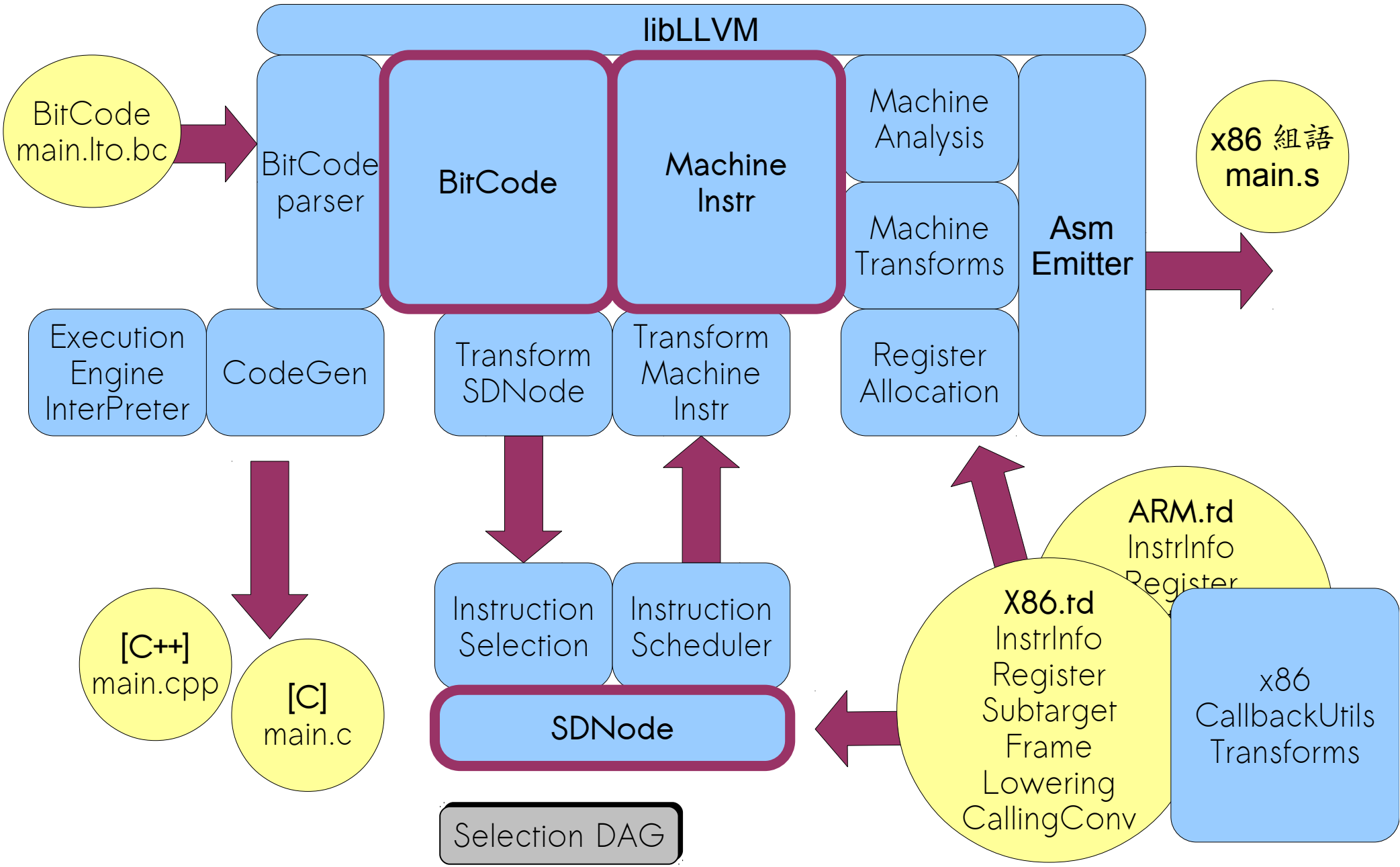
Clang 功能示意


[C]
main.c
[C]
mod.c
libClang
lex
parse
AST
semantics
Serialize/
Deserialize
codegen
Analysis
Checker
Indexer
BitCode
main.bc
BitCode
mod.bc
AST
main.ast
AST
mod.ast
輸出／解析
IDE 整合
(Xcode)

BitCode + Optimizer



LLVM Code Generation



先從 Hello World 開始

- 完整的 Compiler Driver

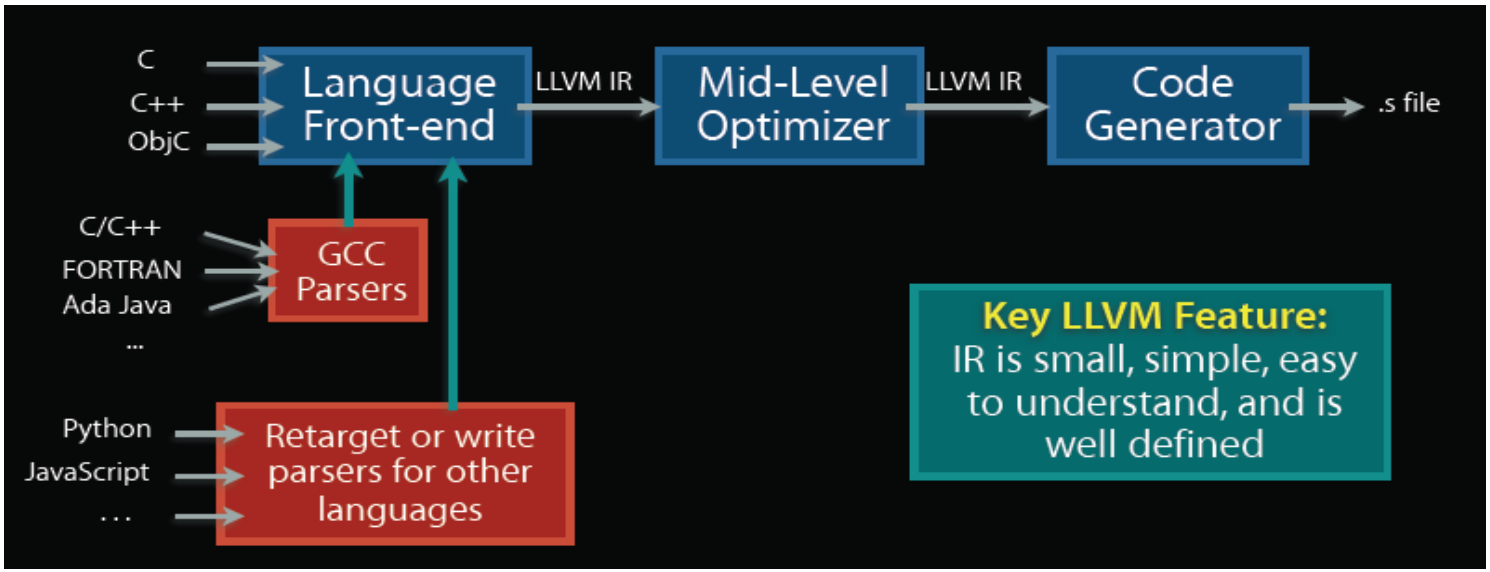
```
$ clang hello.c -o hello
```

- 生成 IR

```
$ clang -O3 -emit-llvm hello.c -c -o hello.bc
```

- 以 Just-In-Time compiler 模式執行 BitCode

```
$ lli hello.bc
```



Getting Started with the LLVM System
http://llvm.org/docs/GettingStarted.html

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

函式 printf() 後方僅有一個字串參數，前端預設將其轉換為 puts()

- 反組譯 BitCode

```
$ llvm-dis < hello.bc
```

```
; ModuleID = '<stdin>'
target datalayout = "e-p:32:32:32-...
target triple = "i386-pc-linux-gnu"

@str = internal constant [13 x i8] c"Hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @str, i32 0, i32 0))
  ret i32 0
}

Declare i32 @puts(i8* nocapture) nounwind
```

- 輸出 x86 後端組合語言

```
$ llc hello.bc -o hello.s
```

Hello World in NES

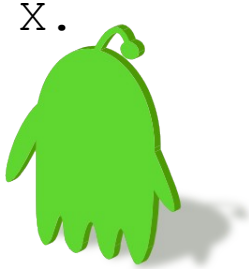
```
org $C000

msg: .data "Hello, world!", 10, 0

Reset_Routine:

LDX #$00        ; put the starting index, 0, in register X

loop: LDA msg, X      ; read 1 char
BEQ loopend     ; end loop if we hit the \0
STA $2008       ; putchar (custom ABI)
INX             ; (Register X)++
JMP loop        ; repeat

loopend:
LDA #$00        ; return code 0
STA $2009       ; exit (custom ABI)

IRQ_Routine: rti ; do nothing
NMI_Routine: rti ; do nothing

org   $FFFA
dc.w  NMI_Routine
dc.w  Reset_Routine
dc.w  IRQ_Routine
```

NES to LLVM IR (1)

```
; "constant" means that this data is read-only.
; [15 x i8] is the type of this data. i8 means an 8-bit integer.
@msg = private constant [15 x i8] c"Hello, World!\0a\00"

; Here we declare a dependency on the `putchar` symbol.
declare i32 @putchar(i32)

; `noreturn` indicates that we do not expect this function to return.
; It will end the process, after all.
; `nounwind` has to do with LLVM's error handling model.
; We use `nounwind` because we know that `exit` will not
; throw an exception.
declare void @exit(i32) noreturn nounwind

define i32 @main() {
Entry:
; Here we allocate some variables on the stack.
; These are X, Y, and A,
; 3 of the 6502's 8-bit registers.
      %X = alloca i8
      %Y = alloca i8
      %A = alloca i8
```

NES to LLVM IR (2)

```
        %S_neg = alloca i1
        %S_zero = alloca i1
; Send control flow to the Reset_Routine basic block.
        br label %Reset_Routine

Reset_Routine:
; This is the code to generate for
; LDX #$00
  ; Store 0 in the X register.
        store i8 0, i8* %X

  ; Clear the negative status bit, because we just stored 0
  ; in X, and 0 is not negative.
        store i1 0, i1* %S_neg

  ; Set the zero status bit, because we just stored 0 in X.
        store i1 1, i1* %S_zero
...
```



驗證 LLVM IR 與執行

- LLVM static compiler

```
$ llc -filetype=obj hello.llvm
```

- Link ELF image

```
$ gcc hello.llvm.o
```

- Veirfy

```
$ ./a.out
Hello, World!
```

```
#include <stdio.h>
void foo(char c) { putchar(c); }
```

```
define void @foo(i8 signext %c) nounwind {
  %1 = alloca i8, align 1                         ; <i8*> [#uses=2]
  store i8 %c, i8* %1
  %2 = load i8* %1                                ; <i8> [#uses=1]
  %3 = sext i8 %2 to i32                          ; <i32> [#uses=1]
  %4 = call i32 @putchar(i32 %3)                  ; <i32> [#uses=0]
  ret void
}

declare i32 @putchar(i32)
```

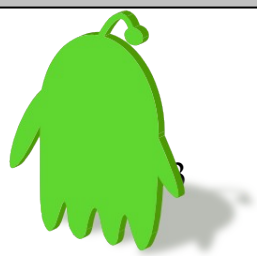
```
// declare i32 @putchar(i32)
Function* putchar = cast<Function>(
    module->getOrInsertFunction(
        "putchar", voidType, cellType, NULL));
putchar->setCallingConv(CallingConv::C);
```

呼叫底層系統 libc 的 putchar 函式

```
Function* makeFunc(Module* module,
                   const char* source,
                   int tapeSize = 400) {
  ...
  Value* zero = ConstantInt::get(cellType, 0);
  Value* one = ConstantInt::get(cellType, 1);
  Value* minOne = ConstantInt::get(cellType, -1);
  ...

  BasicBlock* block =
      BasicBlock::Create(getGlobalContext(), "code", main);
  std::stack<bfLoopInfo> loops;
  IRBuilder<> codeIR(block);
  Value *head = codeIR.CreateAlloca(cellType,
      ConstantInt::get(indexType, tapeSize));
  Value *it = head;
  for (int i = 0; i < tapeSize; i++) {
    codeIR.CreateStore(zero, it);
    it = codeIR.CreateGEP(it, one);
  }
```

在 LLVM IR 中，預先若干定義的常數
zero = 0, one =1, minOne = -1

建立 LLVM IR



Code Generation: LDX

LDX (LoaD X register)

```
func (i *Instruction) Compile(c *Compilation) {
    v := llvm.ConstInt(llvm.Int8Type(),
                       uint64(i.Value), false)
    switch i.OpCode {
    case 0xa2: // ldx
        c.builder.CreateStore(v, c.rX)
        c.testAndSetZero(i.Value)
        c.testAndSetNeg(i.Value)
        // other immediate insn
    }
}
```

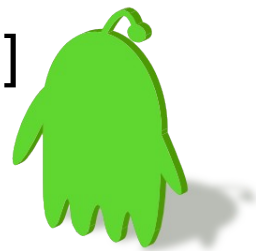
```
func (c *Compilation)
testAndSetZero(v int) {
    if v == 0 {
        c.setZero()
        return
    }
    c.clearZero()
}
```

```
func (c *Compilation)
testAndSetNeg(v int) {
    if v&0x80 == 0x80 {
        c.setNeg()
        return
    }
    c.clearNeg()
}
```

Optimizations

LLVM IR is designed to be optimized

- **Constant Propagation** [ConstantPropagationPass]
  - Looks for instructions involving only constants and replaces them with a constant value instead of an instruction.
- **Combine Redundant Instructions** [InstructionCombining]
  - Combines instructions to form fewer, simple instructions. For example if you add 1 twice, it will instead add 2.
- **Promote Memory to Register** [PromoteMemoryToRegister]
  - This pass allows us to load every "register" variable (X, Y, A, etc) before performing an instruction, and then store the register variable back after performing the instruction.
- **Global Value Numbering** [GVN]
  - Eliminates partially and fully redundant instructions, and delete redundant load instructions.
- **Control Flow Graph Simplification** [CFGSimplification]
  - Removes unnecessary code and merges basic blocks together when possible.



套用各項優化策略

```
engine, err := llvm.NewJITCompiler(c.mod, 3)
if err != nil {
    c.Errors = append(c.Errors, err.Error())
    return
}
defer engine.Dispose()

pass := llvm.NewPassManager()
defer pass.Dispose()

pass.Add(engine.TargetData())
pass.AddConstantPropagationPass()
pass.AddInstructionCombiningPass()
pass.AddPromoteMemoryToRegisterPass()
pass.AddGVNPass()
pass.AddCFGSimplificationPass()
pass.Run(c.mod)
```

原本的輸出 (1)

```
@Label_c000 = private global [15 x i8] c"Hello, World!\0A\00"
declare i32 @putchar(i32)
declare void @exit(i32) noreturn nounwind
define i32 @main() {
Entry:
  %X = alloca i8
  %Y = alloca i8
  %A = alloca i8
  %S_neg = alloca i1
  %S_zero = alloca i1
  br label %Reset_Routine

Reset_Routine:                                    ; preds =
%Entry
  store i8 0, i8* %X
  store i1 true, i1* %S_zero
  store i1 false, i1* %S_neg
  br label %Label_c011
```

原本的輸出 (2)

```
Label_c011:                ; preds = %else, %Reset_Routine
  %0 = load i8* %X
  %1 = getelementptr [15 x i8]* @Label_c000, i8 0, i8 %0
  %2 = load i8* %1
  store i8 %2, i8* %A
  %3 = and i8 %2, -128
  %4 = icmp eq i8 %3, -128
  store i1 %4, i1* %S_neg
  %5 = icmp eq i8 %2, 0
  store i1 %5, i1* %S_zero
  %6 = load i1* %S_zero
  br i1 %6, label %Label_c01d, label %else
```

原本的輸出 (3)

```
else:                             ; preds = %Label_c011
  %7 = load i8* %A
  %8 = zext i8 %7 to i32
  %9 = call i32 @putchar(i32 %8)
  %10 = load i8* %X
  %11 = add i8 %10, 1
  store i8 %11, i8* %X
  %12 = and i8 %11, -128
  %13 = icmp eq i8 %12, -128
  store i1 %13, i1* %S_neg
  %14 = icmp eq i8 %11, 0
  store i1 %14, i1* %S_zero
  br label %Label_c011
```

原本的輸出 (4)

```
Label_c01d:                                       ; preds = %Label_c011
  store i8 0, i8* %A
  store i1 true, i1* %S_zero
  store i1 false, i1* %S_neg
  %15 = load i8* %A
  %16 = zext i8 %15 to i32
  call void @exit(i32 %16)
  unreachable

IRQ_Routine:                                      ; preds = %Label_c01d
  unreachable

NMI_Routine:                                      ; No predecessors!
  unreachable
}
```

```
@Label_c000 = private global [15 x i8] c"Hello, World!\0A\00"
declare i32 @putchar(i32)
declare void @exit(i32) noreturn nounwind
define i32 @main() {
Entry:
  br label %Label_c011
Label_c011:                                       ; preds = %else, %Entry
  %storemerge = phi i8 [ 0, %Entry ], [ %6, %else ]
  %0 = sext i8 %storemerge to i64
  %1 = getelementptr [15 x i8]* @Label_c000, i64 0, i64 %0
  %2 = load i8* %1, align 1
  %3 = icmp eq i8 %2, 0
  br i1 %3, label %Label_c01d, label %else
else:                                             ; preds = %Label_c011
  %4 = zext i8 %2 to i32
  %5 = call i32 @putchar(i32 %4)
  %6 = add i8 %storemerge, 1
  br label %Label_c011
Label_c01d:                                       ; preds = %Label_c011
  call void @exit(i32 0)
  unreachable
}
```

經過 LLVM optimization pass

建構 NES 執行環境

NES ROM 組成

- 16 byte header + metadata
  - 是否使用 mapper
  - PPU 使用垂直或水平鏡像處理
- 組譯的 32 KB 程式碼將會在開機時，載入到 $8000 - $ffff 記憶體區段，若 mapper 存在，會有更多的程式區段可用
- CHR-ROM: 8 KB 大小的圖形資料，將會在開機時，載入到 PPU 。同樣的，若有 mapper ，可允許更多的圖形資料段

NES ROM 組成

- `$ ./jamulate -unrom roms/Mario.nes`

  `loading roms/Mario.nes`

  `disassembling to roms/Mario`
- `$ ls roms/Mario/`

  `chr0.chr` **`Mario.jam`** `prg.asm`

program metadata in a simple key-value text format

chr0.chr is binary data;
it is the 8KB of graphics data mentioned before which is loaded into the PPU on bootup.

http://0xlab.org