中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第22讲：目标代码生成(1)

张献伟

xianweiz.github.io

DCS290, 6/6/2024

Quiz

Handwritten, or email to
chenhq79@mail2.sysu.edu.cn

- Q1: how does BB relate to CFG?

  Each BB is a node of the control flow graph.
  BBs are connected with directed edges.

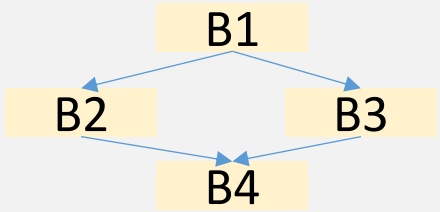
- Q2: how to partition code into BBs?

  Identify leader insts; a BB consists of a leader inst and subsequent insts before next leader.

- Q3: leader instructions of the code?

  (1), (3), (6), (8)

- Q4: CFG of the code?



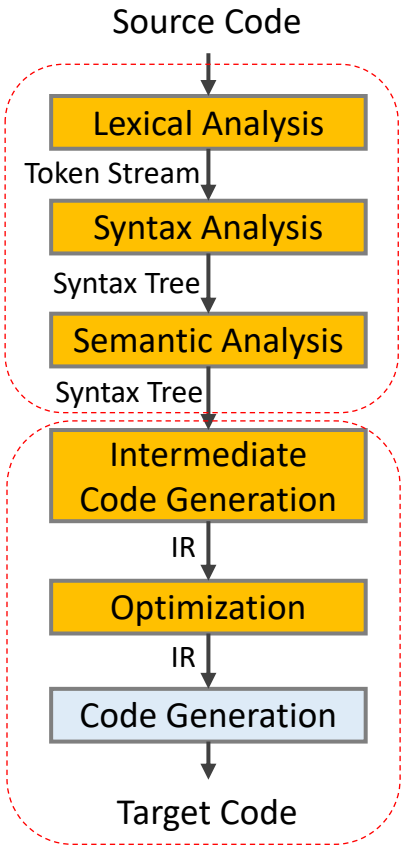
- Q5: optimize the code.

  x = 8

```
(1)      w = 5                 B1
(2)      if w != 0: goto L1
(3)      y = w                 B2
(4)      x = x + y
(5)      goto L2
(6) L1:  y = 3                 B3
(7)      x = w
(8) L2:  x = x + y             B4
```

Target Code Generation[目标代码生成]

- What we have now
  - Optimized IR of the source program
    - And, symbol table
- Target code
  - Binary (machine) code
  - Assembly code
- Goals of target code generation
  - Correctness: the target program must preserve the semantic meaning of the source program
  - High-quality: the target program must make effective use of the available resources of the target machine
  - Fast: the code generator itself must runs efficiently



src → IR → exe: Example

```
1 int x = 1;
2 int y = 2;
3 int z = 3;
4
5 int main() {
6   int rst = x + y + z;
7
8   return rst;
9 }
```

```
          +- 0: input, "test0.c", c
       +- 1: preprocessor, {0}, cpp-output
    +- 2: compiler, {1}, ir
 +- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
```

$clang -emit-llvm -S -O1 asm_test.c

```
@x = dso_local local_unnamed_addr global i32 1, align 4
@y = dso_local local_unnamed_addr global i32 2, align 4
@z = dso_local local_unnamed_addr global i32 3, align 4

; Function Attrs: norecurse nounwind readonly
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = load i32, i32* @x, align 4, !tbaa !2
  %2 = load i32, i32* @y, align 4, !tbaa !2
  %3 = add nsw i32 %2, %1
  %4 = load i32, i32* @z, align 4, !tbaa !2
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

$llvm-as asm_test.ll -o asm_test.bc

$llc -filetype=obj asm_test.bc -o asm_test.o

$clang asm_test.o -o asm_test

IR → asm: Example

```
@x = dso_local local_unnamed_addr global i32 1, align 4
@y = dso_local local_unnamed_addr global i32 2, align 4
@z = dso_local local_unnamed_addr global i32 3, align 4

; Function Attrs: norecurse nounwind readonly
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = load i32, i32* @x, align 4, !tbaa !2
  %2 = load i32, i32* @y, align 4, !tbaa !2
  %3 = add nsw i32 %2, %1
  %4 = load i32, i32* @z, align 4, !tbaa !2
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

$llvm-as asm_test.ll -o asm_test.bc

$llc -filetype=obj asm_test.bc -o asm_test.o

$objdump -d asm_test.o

```
0000000000000000 <main>:
   0:   90000008        adrp    x8, 0 <main>
   4:   90000009        adrp    x9, 4 <main+0x4>
   8:   b9400108        ldr     w8, [x8]
   c:   b9400129        ldr     w9, [x9]
  10:   9000000a        adrp    x10, 8 <main+0x8>
  14:   b940014a        ldr     w10, [x10]
  18:   0b080128        add     w8, w9, w8
  1c:   0b0a0100        add     w0, w8, w10
  20:   d65f03c0        ret
```

$clang asm_test.o -o asm_test

$objdump -d asm_test

```
0000000000400574 <main>:
  400574:       b0000088        adrp    x8, 411000 <__libc_start_main@GLIBC_2.17>
  400578:       b0000089        adrp    x9, 411000 <__libc_start_main@GLIBC_2.17>
  40057c:       b9402908        ldr     w8, [x8, #40]
  400580:       b9402d29        ldr     w9, [x9, #44]
  400584:       b000008a        adrp    x10, 411000 <__libc_start_main@GLIBC_2.17>
  400588:       b940314a        ldr     w10, [x10, #48]
  40058c:       0b080128        add     w8, w9, w8
  400590:       0b0a0100        add     w0, w8, w10
  400594:       d65f03c0        ret
  400598:       d503201f        nop
  40059c:       d503201f        nop
```

ARM vs. X86: IR

ARM

```
; ModuleID = 'asm_test.c'
source_filename = "asm_test.c"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128"
target triple = "aarch64-unknown-linux-gnu"

@x = dso_local local_unnamed_addr global i32 1, align 4
@y = dso_local local_unnamed_addr global i32 2, align 4
@z = dso_local local_unnamed_addr global i32 3, align 4

; Function Attrs: norecurse nounwind readonly
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = load i32, i32* @x, align 4, !tbaa !2
  %2 = load i32, i32* @y, align 4, !tbaa !2
  %3 = add nsw i32 %2, %1
  %4 = load i32, i32* @z, align 4, !tbaa !2
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

```
1 int x = 1;
2 int y = 2;
3 int z = 3;
4
5 int main() {
6   int rst = x + y + z;
7
8   return rst;
9 }
```

x86

```
; ModuleID = 'asm_test.c'
source_filename = "asm_test.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@x = dso_local local_unnamed_addr global i32 1, align 4
@y = dso_local local_unnamed_addr global i32 2, align 4
@z = dso_local local_unnamed_addr global i32 3, align 4

; Function Attrs: norecurse nounwind readonly uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = load i32, i32* @x, align 4, !tbaa !2
  %2 = load i32, i32* @y, align 4, !tbaa !2
  %3 = add nsw i32 %2, %1
  %4 = load i32, i32* @z, align 4, !tbaa !2
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

ARM vs. X86: assembly

ARM

```
asm_test.o:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000000000 <main>:
   0:   90000008        adrp    x8, 0 <main>
   4:   90000009        adrp    x9, 4 <main+0x4>
   8:   b9400108        ldr     w8, [x8]
   c:   b9400129        ldr     w9, [x9]
  10:   9000000a        adrp    x10, 8 <main+0x8>
  14:   b940014a        ldr     w10, [x10]
  18:   0b080128        add     w8, w9, w8
  1c:   0b0a0100        add     w0, w8, w10
  20:   d65f03c0        ret
```

```
1 int x = 1;
2 int y = 2;
3 int z = 3;
4
5 int main() {
6   int rst = x + y + z;
7
8   return rst;
9 }
```

ADRP: Address of 4KB page at a PC-relative offset.

x86

```
asm_test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 6 <main+0x6>
   6:   03 05 00 00 00 00       add    0x0(%rip),%eax        # c <main+0xc>
   c:   03 05 00 00 00 00       add    0x0(%rip),%eax        # 12 <main+0x12>
  12:   c3                      retq
```

RIP (instruction pointer) register points to next instruction to be executed.

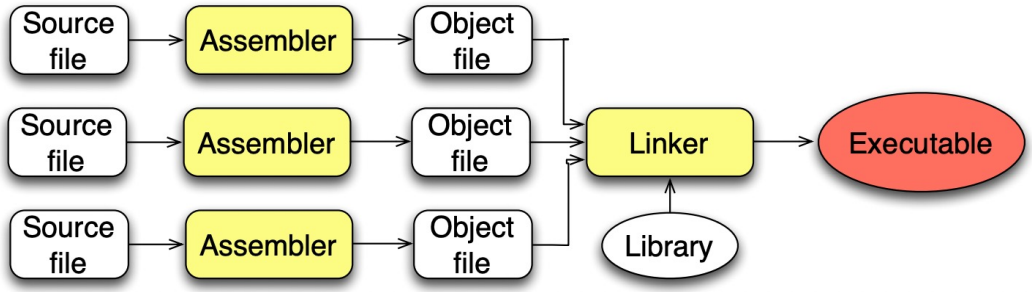
Assembly vs. Assembler

- **Assembly language**: a programming language that is close to machine language but not the same
  - Symbolic representation of a computer's binary machine lang
- **Assembler**: a program (a mini-compiler) that translates assembly language into real machine code (long sequences of 0s and 1s)
  - Translate commands in assembly language like addi t3 t6 t8 into machine code
  - Convert symbolic addresses such as main or loop into machine addresses like 10001101001101001101001101010100 1
    - This task is sometimes deferred to the **linker**

https://users.sussex.ac.uk/~mfb21/compilers/slides/11-handout.pdf

Assembler & Linker

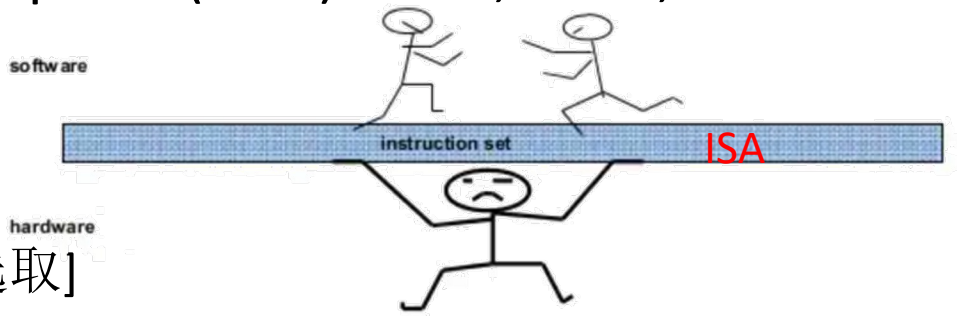
- **Assembler** translates source files to object files, which are machine code, but contains 'holes' (basically references to external code)
  - Because of holes, object files (a.k.a., relocatable object file) cannot be executed directly
  - The holes arise because the assembler translates each file separately
- The **linker** gets all object files and libraries and puts the right addresses into holes, yielding an executable



https://users.sussex.ac.uk/~mfb21/compilers/slides/11-handout.pdf

Translating IR to Machine Code[翻译]

- Machine code generation is machine ISA dependent*
  - Complex instruction set computer (CISC): x86
  - Reduced instruction set computer (RISC): ARM, MIPS, RISC-V



- Three primary tasks
  - **Instruction selection**[指令选取]
    - Choose appropriate target-machine instructions to implement the IR statements
  - **Register allocation** and assignment[寄存器分配]
    - Decide what values to keep in which registers
  - **Instruction ordering**[指令排序]
    - Decide in what order to schedule the execution of instructions

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)

Instruction Selection[指令选取]

- Code generation is to map the IR program into a code sequence that can be executed by the target machine[选择适当的目标机器指令来实现IR]
  - ISA of the target machine
    - If there is 'INC', then for a = a + 1, 'INC a' is better than 'LD a; ADD a, 1'
  - Desired quality of the generated code
    - Many different generations, naïve translation is usually correct but very inefficient

TAC code:

```
a = b + c
d = a + e
```

Target code:

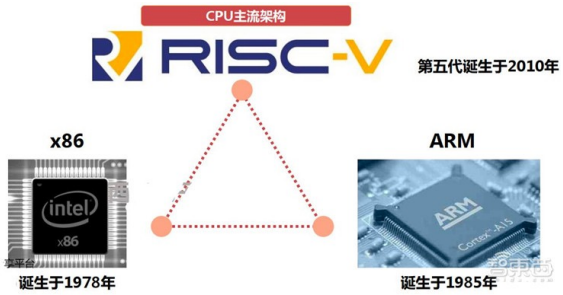
```
LD R0, b          // R0 = b
ADD R0, R0, c     // R0 = R0 + c
ST a, R0          // a = R0
LD R0, a          // R0 = a
ADD R0, R0, e     // R0 = R0 + e
ST d, R0          // d = R0
```

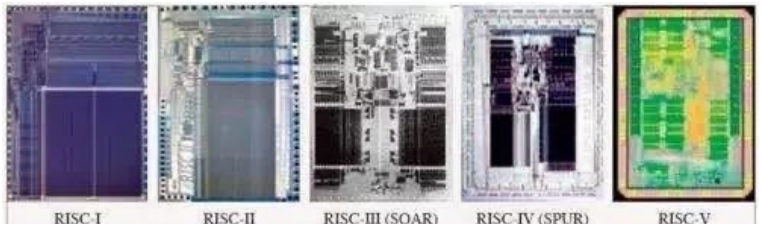
Register Allocation & Evaluation Order

- **Register allocation**: a key problem in code generation is deciding what values to hold in what registers[寄存器分配]
  - Registers are the fastest storage unit but are of limited numbers
    - Values not held in registers need to reside in memory
    - Insts involving register operands are much shorter and faster
  - Finding an optimal assignment of registers to variables is NP-hard
- **Evaluation order**: the order in which computations are performed can affect the efficiency of the target code[执行顺序]
  - Some computation orders require fewer registers to hold intermediate results than others
  - However, picking a best order in the general case is NP-hard

x86 → ARM → RISC-V[进行中的变革]

- The war started in mid 1980's
  - CISC won the high-end commercial war (1990s to today)
  - RISC won the embedded computing war
- But now, things are changing …
  - Fugaku ARM supercomputer, Apple M1 chip, Nvidia Superchip
- RISC-V: a freely licensed open standard (Linux in hw)
  - Builds on 30 years of experience with RISC architecture, "cleans up" most of the short-term inclusions and omissions
    - Leading to an arch that is easier and more efficient to implement

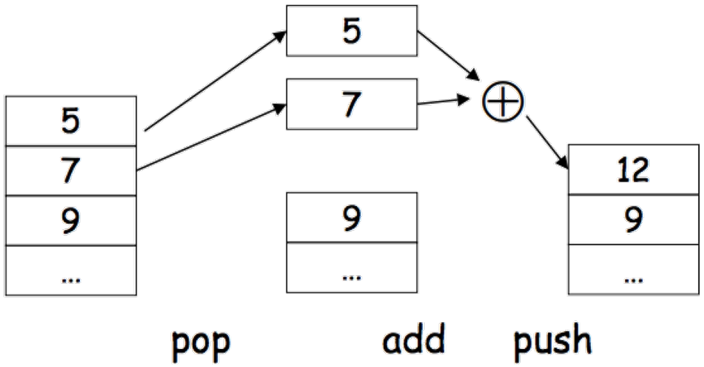




https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html
The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC

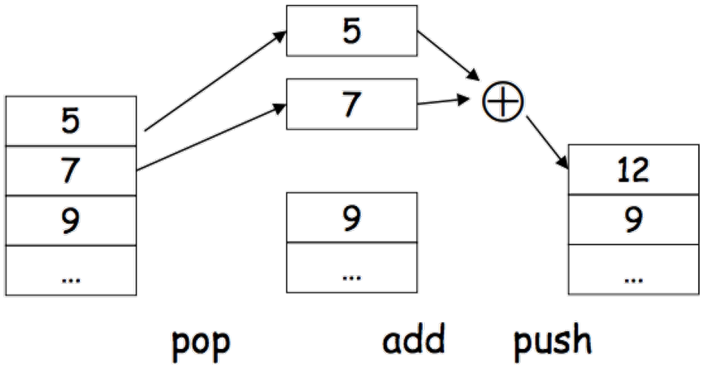
Stack Machine[栈式计算机]

- A simple evaluation model[一个简单模型]
  - No variables or registers
  - A stack of values for intermediate results
- Each instruction[指令任务]
  - Takes its operands from the top of the stack[栈顶取操作数]
  - Removes those operands from the stack[从栈中移除操作数]
  - Computes the required operation on them[计算]
  - Pushes the result on the stack[将计算结果入栈]



Example

- Consider two instructions
  - *push i* - place the integer *i* on top of the stack
  - *add* - pop two elements, add them and put the result back on the stack
- A program to compute *7 + 5*
  - *push 7*
  - *push 5*
  - *add*



Optimize the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed

```
push 7
push 5
add
```

- **Idea**: keep the top of the stack in a register (called *accumulator*)[使用寄存器]
  - Register accesses are much faster
- The "add" instruction is now
  - *acc* ← *acc* + *top_of_stack*
  - Only one memory operation