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23讲：代码优化(3)

张献伟

xianweiz.github.io

DCS290, 5/30/2023

Review Questions

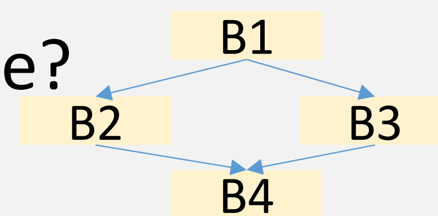
- Q1: what is a Basic Block?

  A straight-line sequence of code with only one entry point and only one exit.

- Q2: how to partition code into BBs?

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

- Q3: CFG of the listed code?



- Q4: Global vs local optimization?

  Across BBs vs. single BB.

- Q5: Usage of DAG?

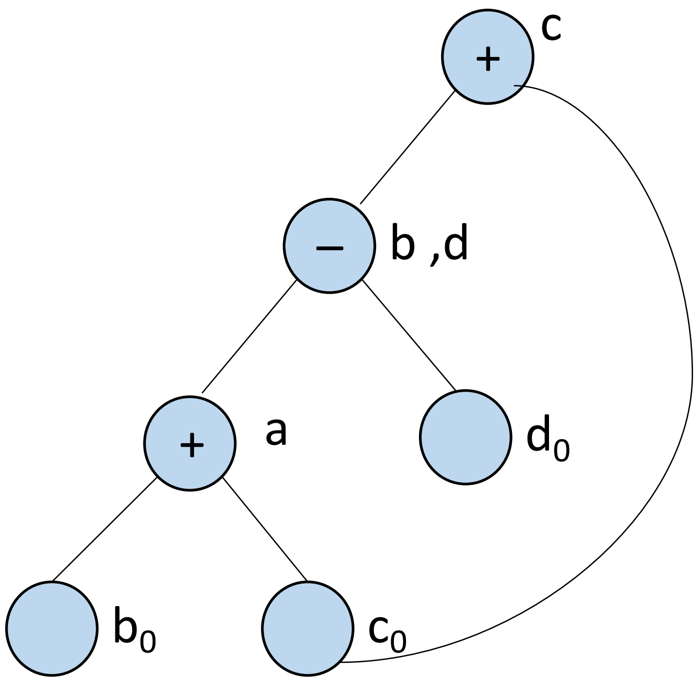
  Directed acyclic graph of a BB to identify local optimizations.

```
    w = 0                 ┐
    y = 0                 │ B1
    x = x + y             │
    if x > z: goto L1     ┘
    y = z                 ┐
    z ++                  │ B2
    goto L2               ┘
L1: y = x                 ┐ B3
    x ++                  ┘
L2: w = x + z             } B4
```

Local Opt.: Elimination

- If *c* is not live on exit from the block
  - No need to keep *c = b + c*
- If both *b* and *d* are live
  - Remove either (2) or (4) : **common subexpr elimination**
  - Add a 4th statement to copy one to the other
- If only *a* is live on exit
  - Then remove nodes from the DAG correspond to dead code
    - c -> b,d -> d_0
  - This is actually **dead code elimination**

(1) a = b + c
(2) b = a - d
(3) c = b + c
(4) d = a - d

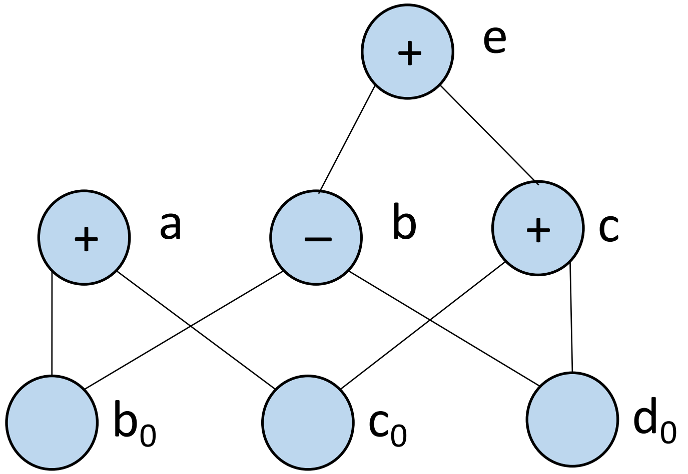


Local Opt.: Elimination (cont.)

- When finding common subexprs, we really are finding exprs that are guaranteed to compute the same value, no matter how that value is computed[过于严苛]
  - Thus miss the fact that (1) and (4) are the same
    - $b + c = (b - d) + (c + d) = b_0 + c_0$

(1) a = b + c
(2) b = b - d
(3) c = c + d
(4) e = b + c

- **Solution**: algebraic identities[代数恒等式]



Local Opt.: Algebraic Identities[代数恒等式]

- Eliminate computations by applying mathematical rules[使用数学规则]
  - Identities: a * 1 ≡ a, a * 0 ≡ 0, b & true ≡ b
  - Reassociation and commutativity[重组合、交换]
    - (a + b) + c ≡ a + (b + c) , a + b ≡ b + a
- **Strength Reduction**[强度削减]
  - Replacing expensive operations (*multiplication*, *division*) by less expensive operations (*add*, *sub*, *shift*)
  - Some ops can be replaced with cheaper ops
  - Examples
    - x=y/8 --> x=y»3
    - y=y*8 --> x=y«3
    - x^2 --> x * x
    - 2 * x --> x + x

Local Opt.: Constant Folding[常量折叠]

- **Constant Folding**
  - Computing operations on constants at compile time
  - Example:

```
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print(”error”);
```

  - After constant folding

```
x = 200;
if (false) print(”error”);
```

  - Dead code elimination can further remove the above *if* statement
  - Inherently local since scope limited to statement

Local Opt.: Constant Propagation[常量传播]

- **Constant Propagation**
  - Substituting values of known constants at compile time
  - Local Constant Propagation (LCP)

```
x = 3;
y = x * 2;
```
→
```
x = 3;
y = 3 * 2;
```
→
```
x = 3;
y = 6;
```

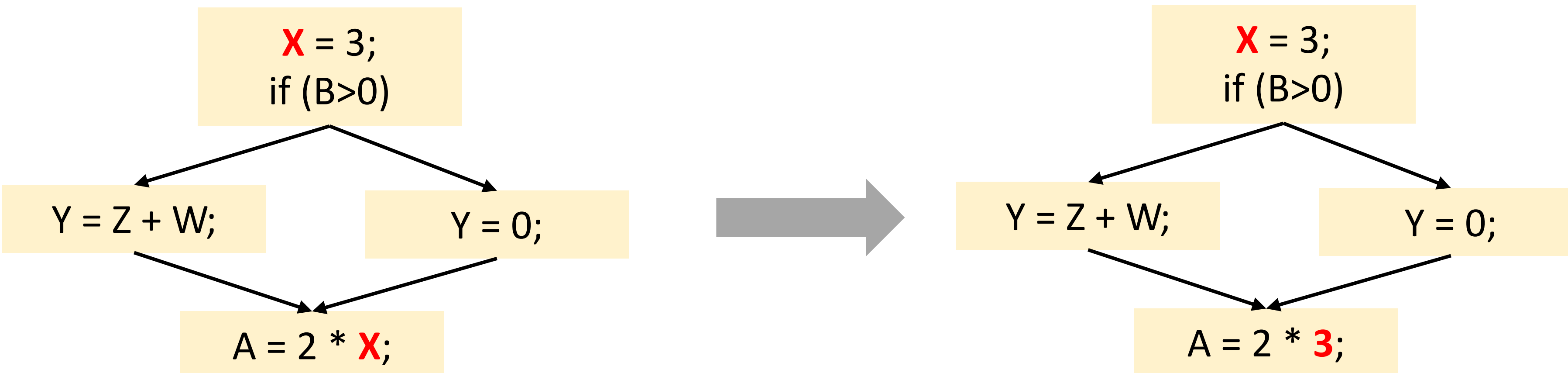
- Some optimizations have both local and global versions
  - Global Constant Propagation (GCP)

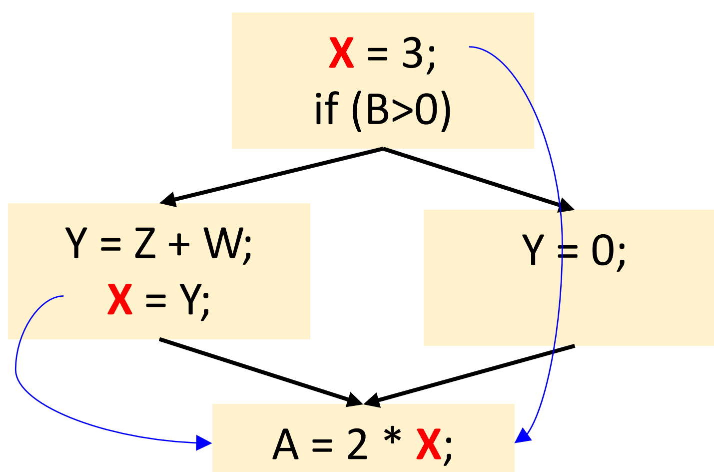
```
a = 1;
x = 3;
if (…)
  x = a + 2;
y = x;
```
→
```
a = 1;
x = 3;
if (…)
  x = 1 + 2;
y = x;
```
→
```
a = 1;
x = 3;
if (…)
  x = 3;
y = 3;
```

  - GCP more powerful than LCP but also more complicated
    - Must determine *x* is constant across all paths reaching *x*

Global Optimizations

- Extend optimizations to flow of control, i.e. CFG
  - Along **<u>all paths</u>**, the last assignment to X is “X=C”
  - Optimization must be stopped if incorrect in even one path





Global Opt.: Conservative[需保守]

- Compiler must prove some property X at a particular point
  - Need to prove at that point property X holds along all paths
  - Need to be **conservative** to ensure correctness
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say **don't know**
    - If analysis result is **don't know**, no optimization done
    - May lose opt. opportunities but guarantees correctness
- Property X often involves data flow of program
  - E.g. Global Constant Propagation (GCP):

```
X = 7;
...
Y = X + 3; // Does value of 7 flow into this use of X?
```

  - Needs knowledge of **data flow**, as well as control flow
    - Whether data flow is interrupted between points A and B

Global Opt.: Data Flow[数据流]

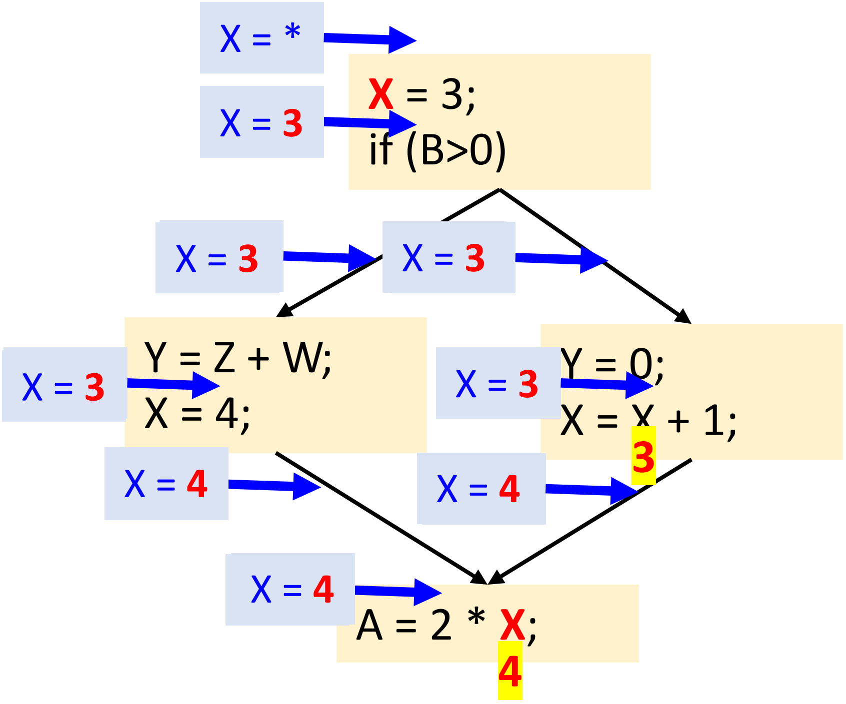
- Most optimizations rely on a property at given point
  - For Global Constant Propagation (GCP):
    A = B + C; // Property: {A=?, B=10, C=?}
  - After optimization:
    A = 10 + C;
- For this discussion, let's call these properties ***values***
- **Dataflow analysis**: compiler analysis that calculates values for each point in a program
  - Values get propagated from one statement to the next
  - Statements can modify values (for GCP, assigning to variables)
  - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: a framework for dataflow analysis that guarantees correctness for **all paths**
  - Does *not* traverse all possible paths (could be infinite)
  - To be feasible, makes **conservative** approximations

Global Constant Propagation (GCP)

- Let's apply dataflow analysis to compute values for GCP
  - Emulates what human does when tracing through code
- Let's use following notation to express the state of a var:
  - x=*: not assigned (default)
  - x=1, x=2, ...: assigned to a constant value
  - x=#: assigned to multiple values
- All values start as x=* and are iteratively refined
  - Until they stabilize and reach a fixed point
- Once fixed point is reached, can replace with constants:
  - x=*: replace with any constant (typically 0)
  - x=1, x=2, ...: replace with given constant value
  - x=#: cannot do anything

Example

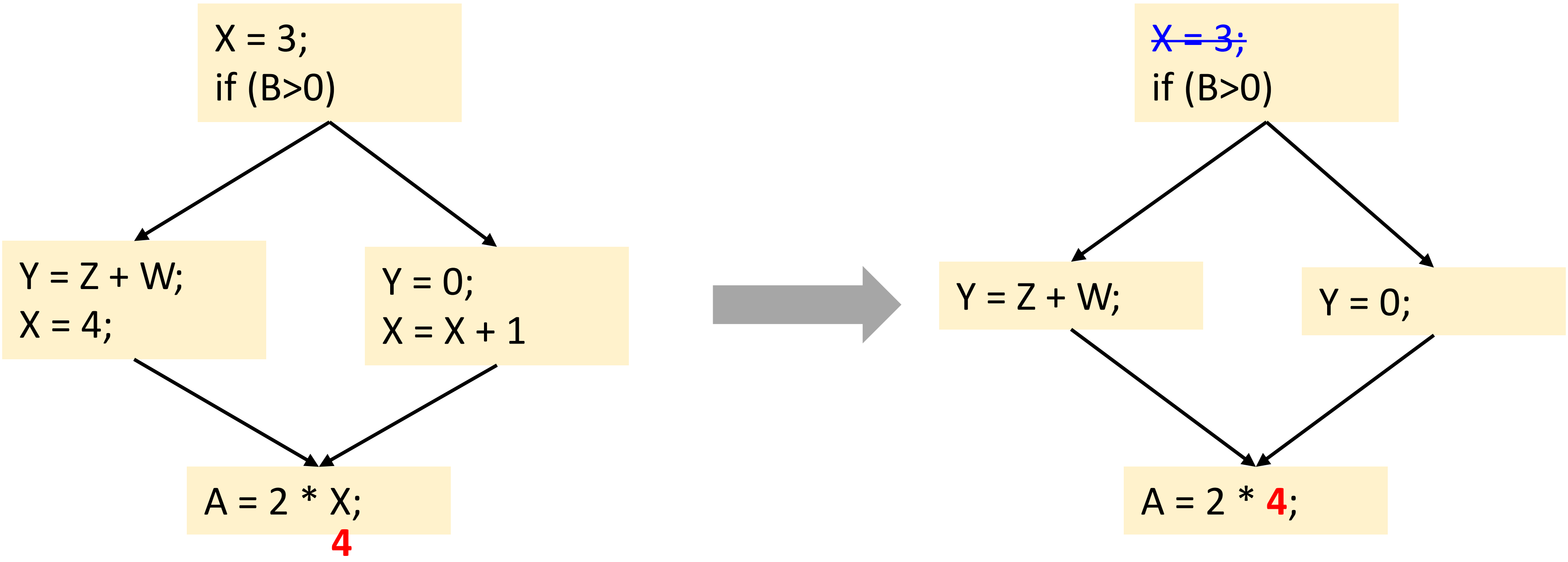
- In this example, constants can be propagated to **X+1**, **2*X**
- Statements visited in reverse postorder (predecessor first)



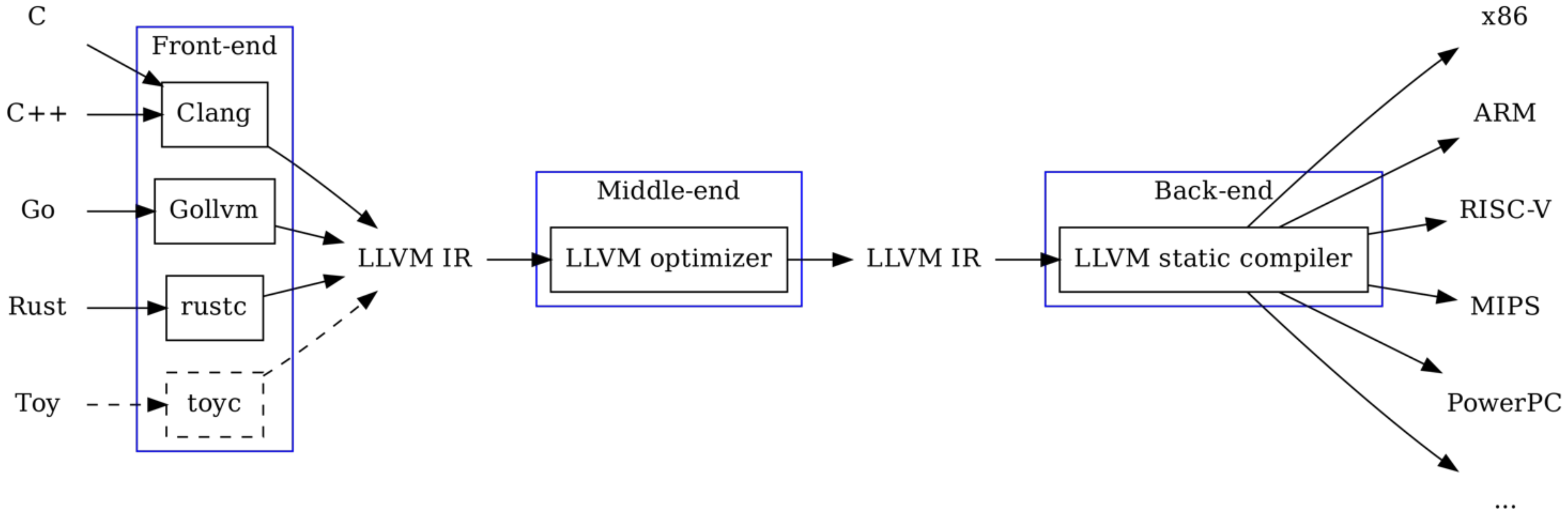
x=*: not assigned (default)
x=1, x=2, ...: assigned to a constant value
x=#: assigned to multiple values

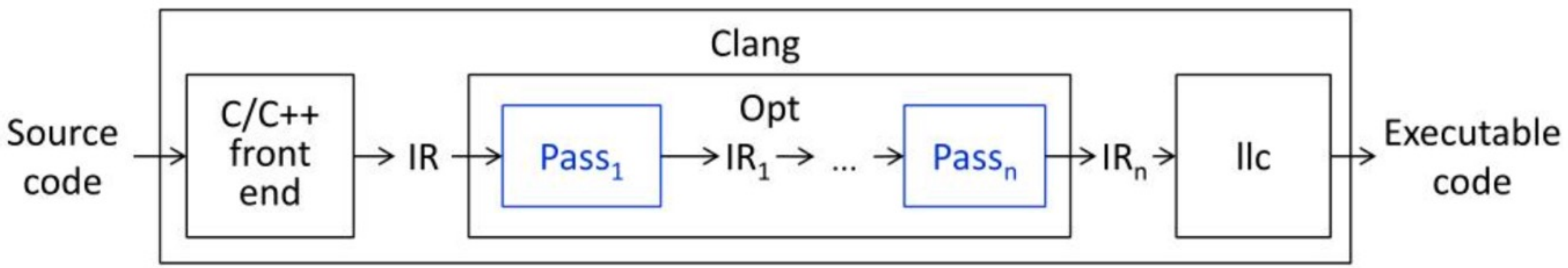
Example (cont.)

- Once constants have been globally propagated, we would like to eliminate the dead code



IR Optimization of LLVM





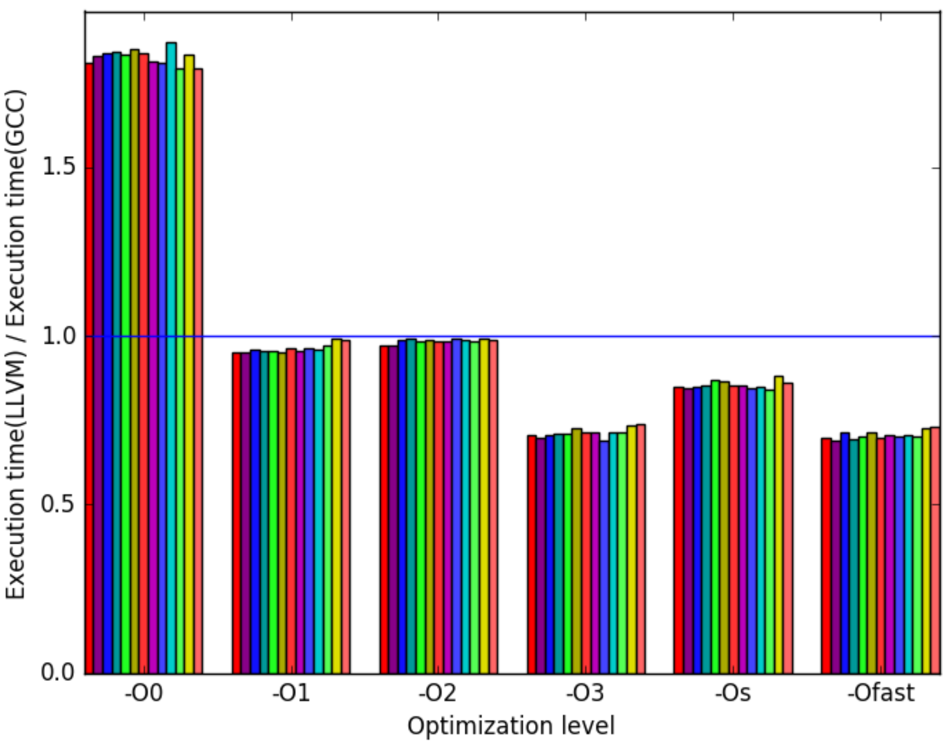
https://www.slideserve.com/quinlan-dominguez/llvm-pass-and-code-instrumentation

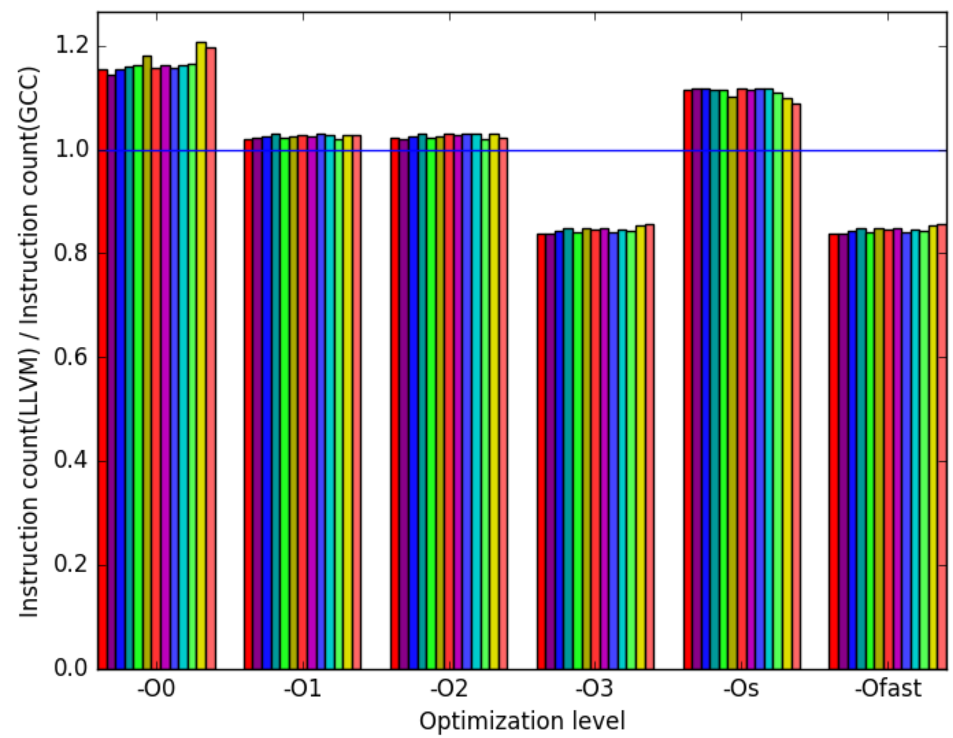
LLVM Optimization Flags

- O0: no optimization
  - Compiles the fastest and generates the most debuggable code
- O1: somewhere between O0 and O2
- O2: moderate level of optimization enabling most optimizations
- O3: like O2,
  - except that it enables opts that take longer to perform or that may generate larger code (in an attempt to make the program run faster)
- Os: like O2 with exta opts to reduce code size
- Oz: like Os, but reduce code size further
- O4: enables link-time opt Clang has support for O4, but not opt

Performance at Varying Flags

- Compare the performance of the benchmark when compiled with either GCC or LLVM
  - Compile benchmark at six optimization levels
  - Each workload was run 3 times with each executable on the Intel Core i7-2600 machines

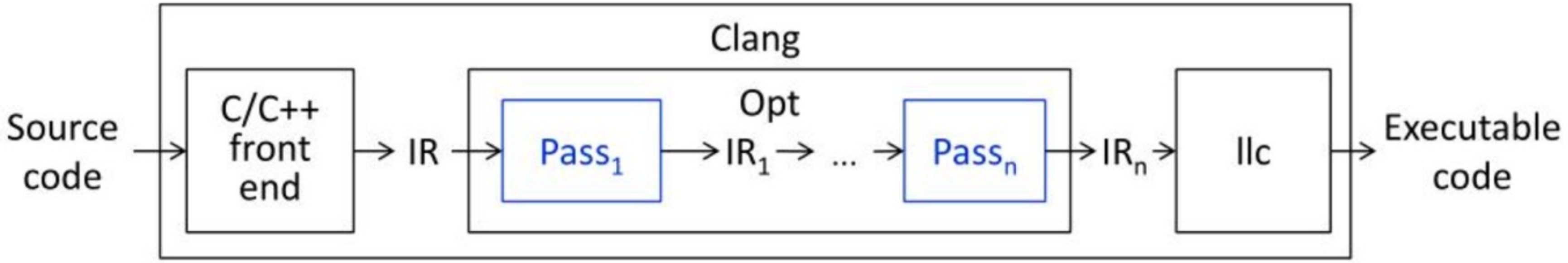




https://webdocs.cs.ualberta.ca/~amaral/AlbertaWorkloadsForSPECCPU2017/reports/exchange2_report.html#x1-12003r1

LLVM Passes

- Optimizations are implemented as **Passes** that traverse some portion of a program to either collect information or transform the program
- A Pass receives an LLVM IR and performs analyses and/or transformations
  - Using opt, it is possible to run each Pass
- A Pass can be executed in a middle of compiling process from source code to binary code
  - The pipeline of Passes is arranged by Pass Manager



https://www.slideserve.com/quinlan-dominguez/llvm-pass-and-code-instrumentation

LLVM Passes (cont.)

- **Analysis** passes: compute info that other passes can use or for debugging or program visualization purposes
  - -memdep: Memory Dependence Analysis (https://llvm.org/doxygen/MemDepPrinter_8cpp_source.html)
  - -instcount: Counts the various types of Instructions (https://llvm.org/doxygen/InstCount_8cpp_source.html)
  - … (https://llvm.org/doxygen/dir_a25db018342d3ae6c7e6779086c18378.html)
- **Transform** passes: can use (or invalidate) the analysis passes, all mutating the program in some way
  - -dce: Dead Code Elimination (https://llvm.org/doxygen/DCE_8cpp_source.html)
  - -loop-unroll: Unroll loops (https://llvm.org/doxygen/LoopUnrollPass_8cpp_source.html)
  - … (https://llvm.org/doxygen/dir_a72932e0778af28115095468f6286ff8.html)
- **Utility** passes: provides some utility but don't otherwise fit categorization
  - -view-cfg: View CFG of function
  - …

https://www.llvm.org/docs/Passes.html

Example

- $clang -emit-llvm -S sum.c

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

- $opt sum.ll -debug-pass=Structure -mem2reg -S -o sum-O1.ll

```
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -assumption-cache-tracker -domtree -mem2reg -verify -print-module
Target Library Information
Target Transform Information
Target Pass Configuration
Assumption Cache Tracker
  ModulePass Manager
    FunctionPass Manager
      Dominator Tree Construction
      Promote Memory to Register
      Module Verifier
    Print Module IR
```

$opt sum.ll -debug-pass=Structure -O1 -S -o sum-O1.ll
$opt sum.ll -time-passes -O1 -o sum-tim.ll

- $opt sum.ll -time-passes -mem2reg -o sum-tim.ll

```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0002 ( 91.1%)   0.0001 ( 90.2%)   0.0003 ( 90.8%)   0.0003 ( 90.6%)  Bitcode Writer
   0.0000 (  3.7%)   0.0000 (  4.5%)   0.0000 (  4.0%)   0.0000 (  3.7%)  Module Verifier
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.8%)  Dominator Tree Construction
   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.3%)   0.0000 (  2.4%)  Promote Memory to Register
   0.0000 (  0.5%)   0.0000 (  0.8%)   0.0000 (  0.6%)   0.0000 (  0.6%)  Assumption Cache Tracker
   0.0002 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)  Total

===-------------------------------------------------------------------------===
                         LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0006 seconds (0.0006 wall clock)
```

Compilation Principle
编 译 原 理

第23讲：目标代码生成(1)

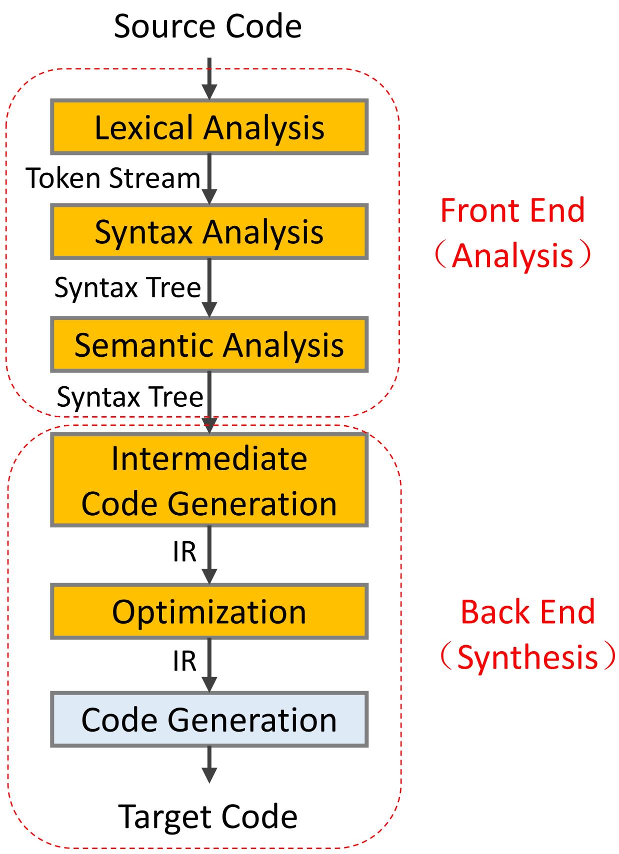
张献伟

xianweiz.github.io

DCS290, 5/30/2023

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

```c
int x = 1;
int y = 2;
int z = 3;

int main() {
  int rst = x + y + z;

  return rst;
}
```

```
            +- 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

```llvm
@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

EXE

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
}
```

```
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
```

```
$llvm-as asm_test.ll -o asm_test.bc
$llc -filetype=obj asm_test.bc -o asm_test.o
```

```
$objdump -d asm_test.o
```

```
$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
}
```

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
```

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
```

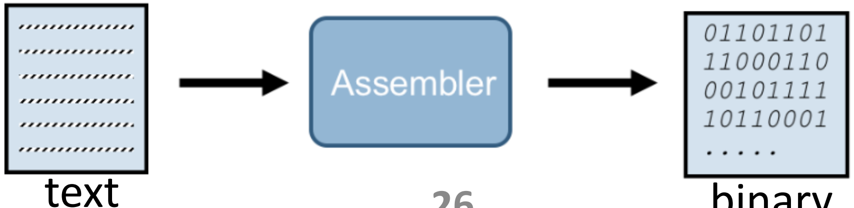
RIP (instruction pointer) register points to next instruction to be executed.

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Assembly vs. Assembler

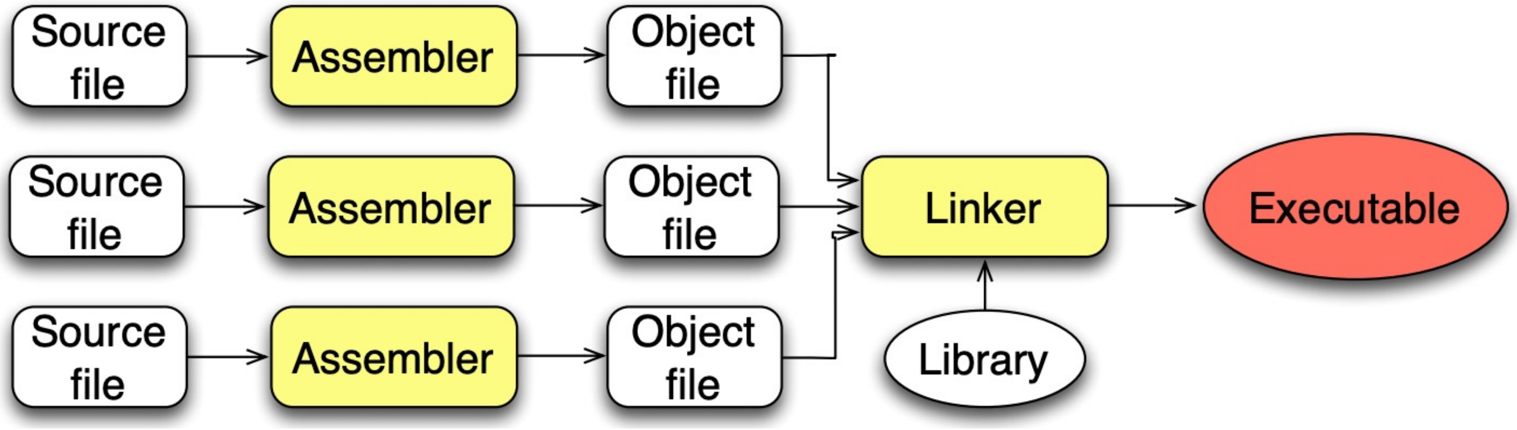
- **Assembly language**: a programming language that is close to machine language but not the same
  - Symbolic representation of a computer's binary machine language
- **Assembler**: a program (a mini-compiler) that translates assembly language into real machine code (long sequences of 0s and 1s)
  - Translate commands in assembly language like addi t3 t6 t8 into machine code
  - Convert symbolic addresses such as main or loop into machine addresses such as 1000110100110100110100110101001. This task is sometimes deferred to the **linker**



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

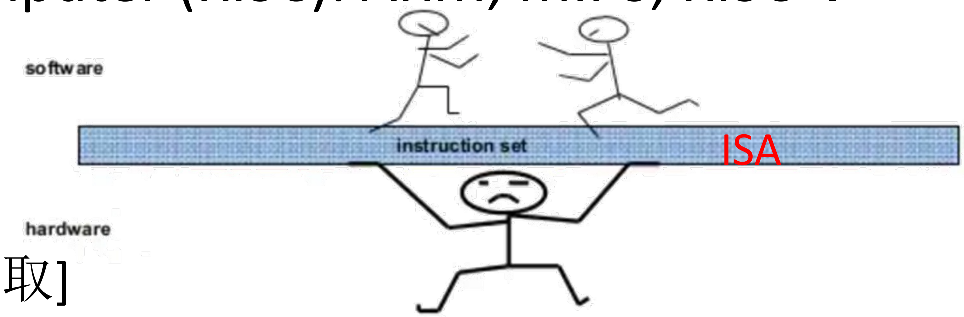
Assembler & Linker

- **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



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:

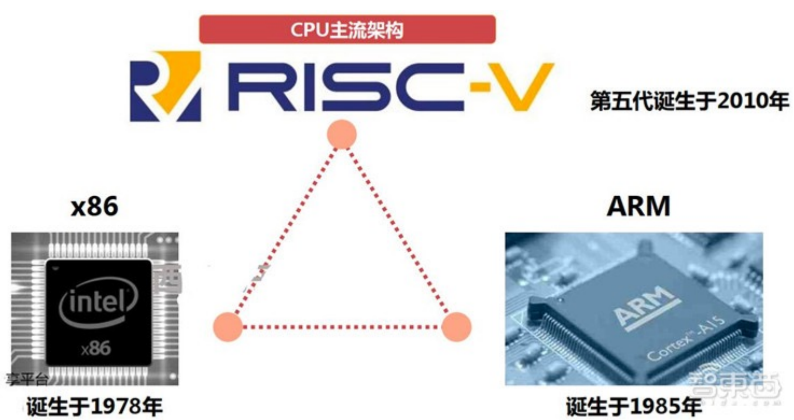
```
LD R0, b          // R0 = b
ADD R0, R0, c     // R0 = R0 + c
ST a, R0          // a = R0
LD R0, a          // R0 = a
ADD R0, R0, e     // R0 = R0 + e
ST d, R0          // d = R0
```

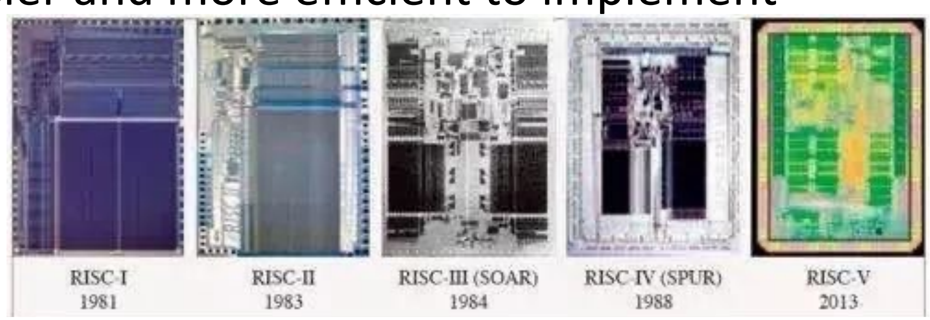
Register Allocation & Evaluation Order

- **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-based supercomputer, Apple ARM-based M1 chip
- 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

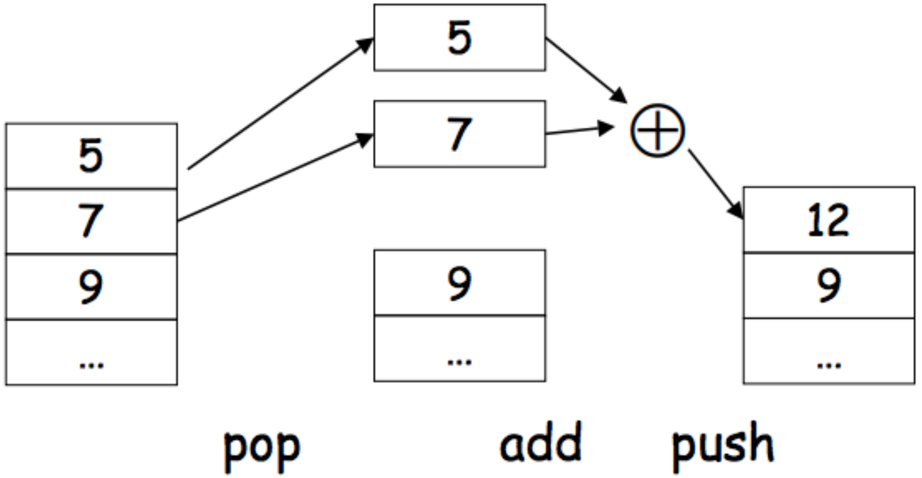




https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html
The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known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

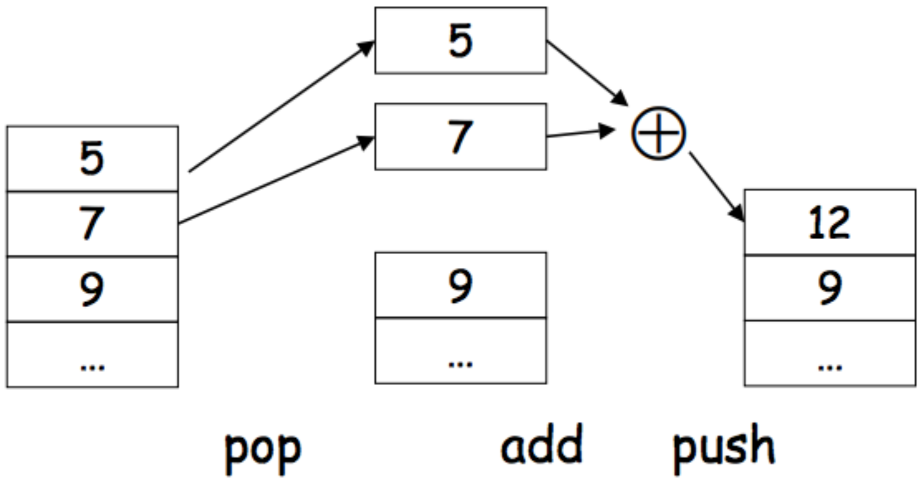
Stack Machine[栈式计算机]

- 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*

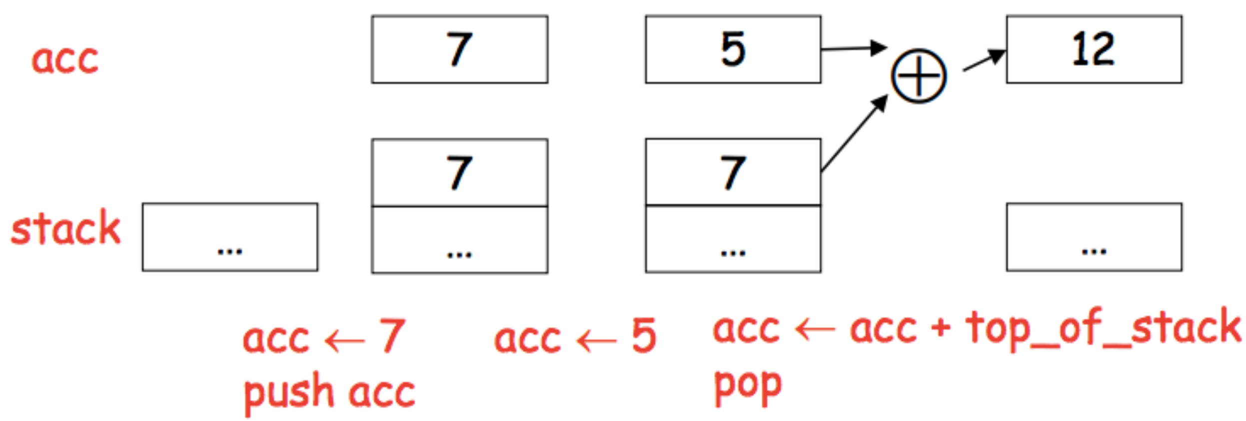


Optimize the Stack Machine

- 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



SUN YAT-SEN UNIVERSITY