INTRODUCTION TO LLVM

DCC 888

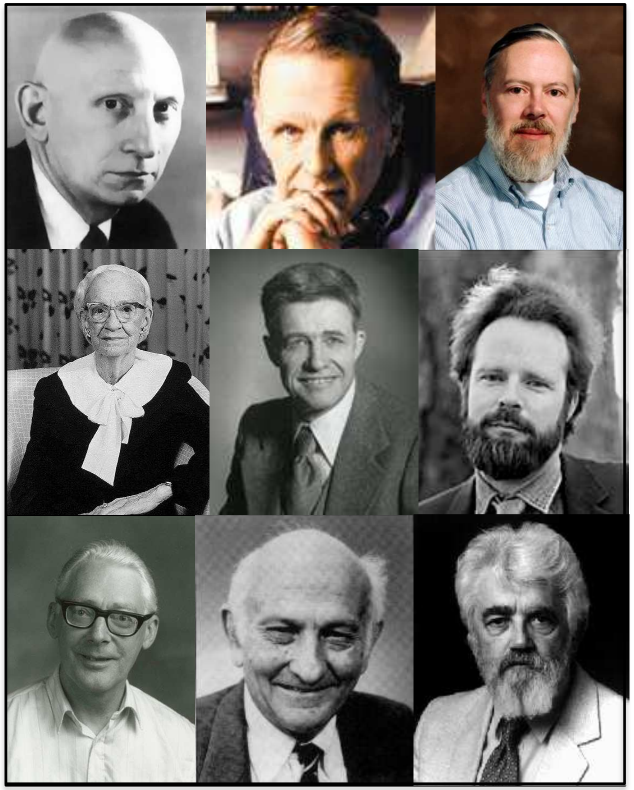
Why all this?

- Why should we study compilers?
- What is LLVM?
- Why should we study LLVM?
- What should we learn about LLVM?

Why to Learn Compilers?

"We do not need that many compiler guys. But those that we need, we need them badly."

François Bodin – CEO of CAPS



A lot of amazing people in computer science were working with compilers and programming languages.

Who do you know in these photos?

And many of the amazing things that we have today only exist because of compilers and programming languages!

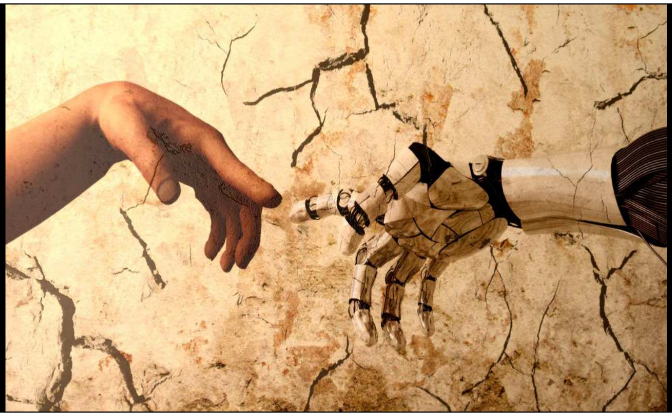
The Mission of the Compiler Writer

The goal of a compiler writer is to bridge the gap between programming languages and the hardware; hence, making programmers more productive

---

*A compiler writer builds **bridges** between **people** and **machines**, and this task is each day more challenging.*

Software engineers want abstractions that let them stay closer to the specification of the problems that they need to solve.

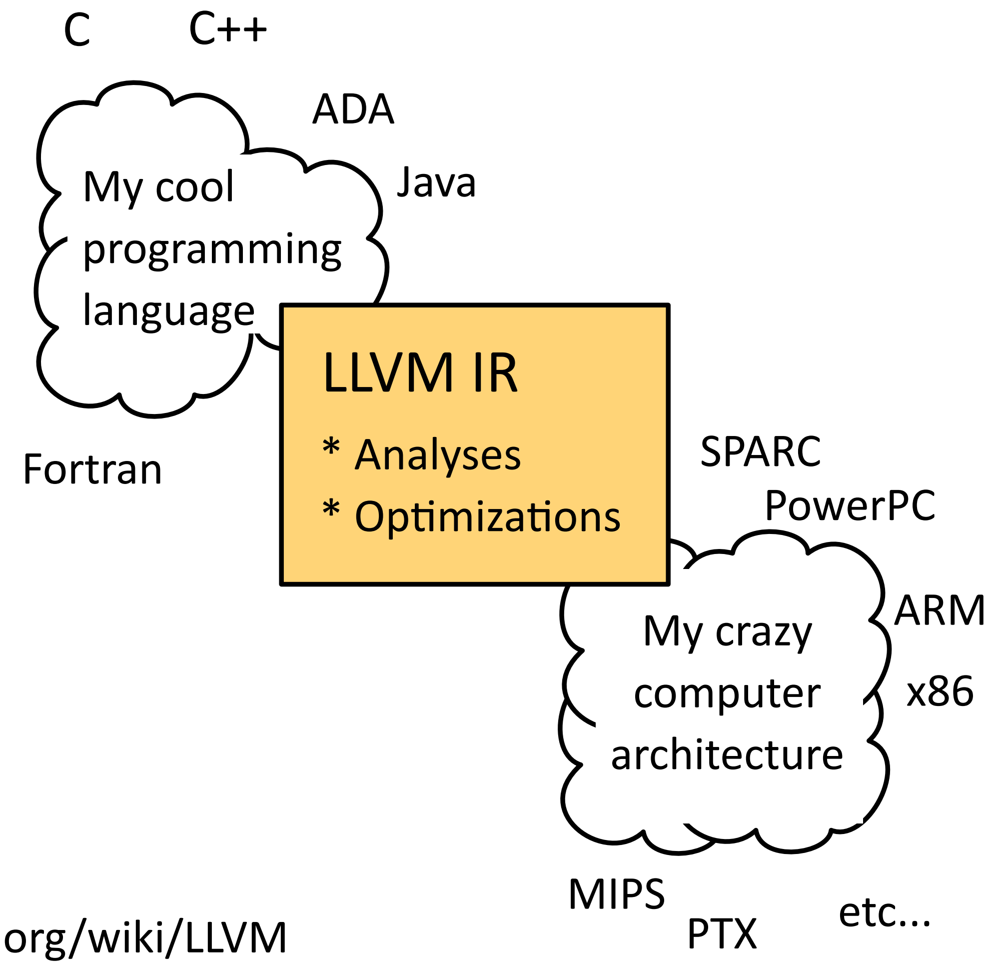


Hardware engineers want efficiency. To obtain every little nanosecond of speed, they build machines each time more (beautifully) complex.

What is LLVM?

LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces♤.

- Implemented in C++
- Several front-ends
- Several back-ends
- First release: 2003
- Open source
- `http://llvm.org/`



♤: Taken from wikipedia at http://en.wikipedia.org/wiki/LLVM

LLVM is a Compilation Infra-Structure

- It is a framework that comes with lots of tools to compile and optimize code.

```
$> cd llvm/Debug+Asserts/bin
$> ls
FileCheck          count                llvm-dis          llvm-stress
FileUpdate         diagtool             llvm-dwarfdump    llvm-symbolizer
arcmt-test         fpcmp                llvm-extract      llvm-tblgen
bugpoint           llc                  llvm-link         macho-dump
c-arcmt-test       lli                  llvm-lit          modularize
c-index-test       lli-child-target     llvm-lto          not
clang              llvm-PerfectSf       llvm-mc           obj2yaml
clang++            llvm-ar              llvm-mcmarkup     opt
llvm-as            llvm-nm              pp-trace          llvm-size
clang-check        llvm-bcanalyzer      llvm-objdump      rm-cstr-calls
clang-format       llvm-c-test          llvm-ranlib       tool-template
clang-modernize    llvm-config          llvm-readobj      yaml2obj
clang-tblgen       llvm-cov             llvm-rtdyld       llvm-diff
clang-tidy
```

LLVM is a Compilation Infra-Structure

- Compile C/C++ programs:

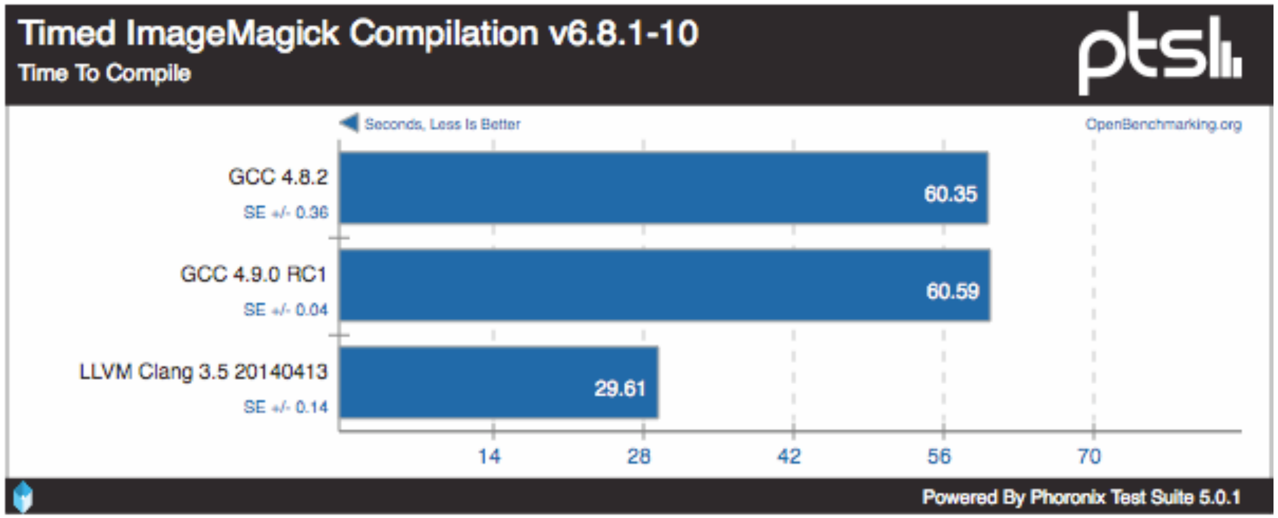
```
$> echo "int main() {return 42;}" > test.c
$> clang test.c
$> ./a.out
$> echo $?
42
```

Which compiler do you think generates faster code: LLVM or gcc?

clang/clang++ are very competitive when compared with, say, gcc, or icc. Some of these compilers are faster in some benchmarks, and slower in others. Usually clang/clang++ have faster compilation times. The Internet is crowed with benchmarks.

LLVM or GCC?

clang/clang++ are very competitive when compared with, say, gcc, or icc. Some of these compilers are faster in some benchmarks, and slower in others. Usually clang/clang++ have faster compilation times. The Internet is crowed with benchmarks:



http://www.phoronix.com/scan.php?page=article&item=gcc49_compiler_llvm35&num=2

Why to Learn LLVM?

- Intensively used in the academia♧:

LLVM: A compilation framework for lifelong program analysis & transformation
C Lattner, V Adve - Code Generation and Optimization, 2004. ..., 2004 - ieeexplore.ieee.org
ABSTRACT This paper describes **LLVM** (Low Level Virtual Machine), a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, **...**
Cited by 1660 Related articles All 68 versions Cite Save

- Used by many companies
  - LLVM is maintained by Apple.
  - ARM, NVIDIA, Mozilla, Cray, etc.
- Clean and modular interfaces.
- Important awards:
  - Most cited CGO paper; ACM Software System Award 2012

♧: 1660 citations in September 27th of 2014

Getting LLVM

LLVM is fairly easy to install. For a quick overview on the process, we recommend: http://llvm.org/releases/3.4/docs/GettingStarted.html

If you want to give it a try yourself, follow the steps below:

```
$> svn co http://llvm.org/svn/llvm-project/llvm/tags/RELEASE_34/final llvm
$> cd llvm/tools
$> svn co http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_34/final clang
$> cd ../projects/
$> svn co http://llvm.org/svn/llvm-project/compiler-rt/tags/RELEASE_34/final compiler-rt
$> cd ../tools/clang/tools/
$> svn co http://llvm.org/svn/llvm-project/clang-tools-extra/tags/RELEASE_34/final extra
```

We are installing LLVM version 3.4 using the commands above.

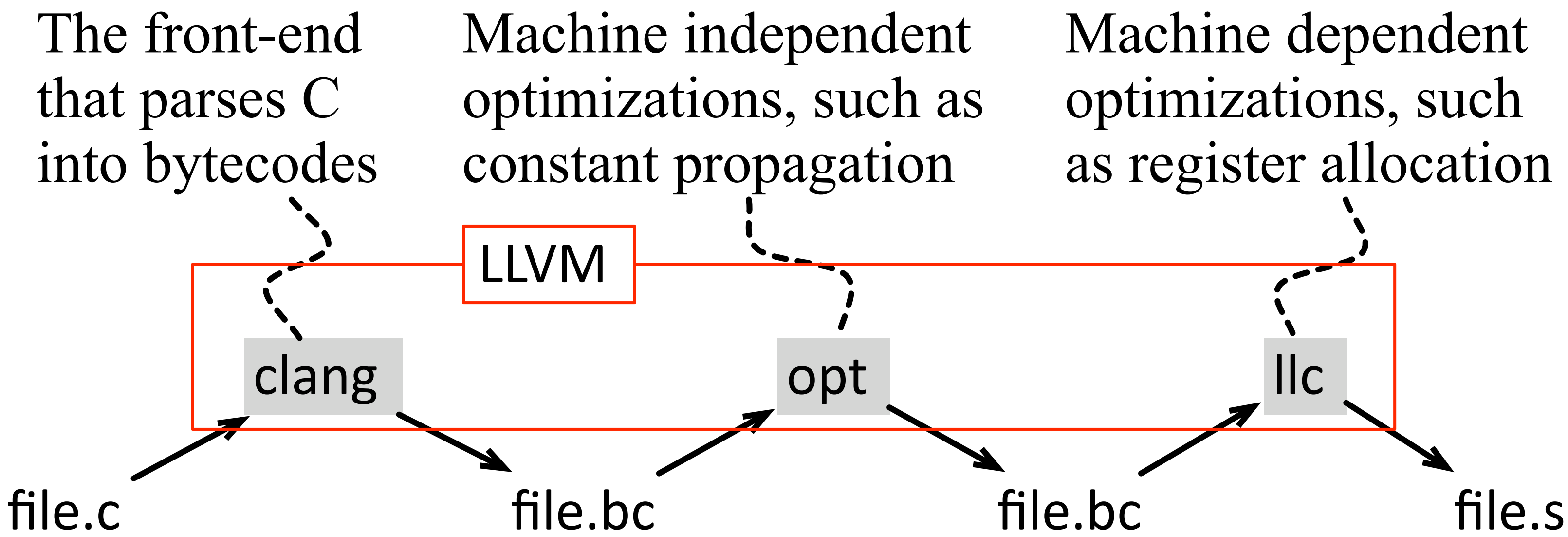
Compiling LLVM

Once you have gotten all the files, via svn, you must compile LLVM. There are more than one way to compile it. If you want to do it quickly, you can configure LLVM with the option --enable-optimized set. Otherwise, a default compilation, with debug symbols, will be performed.

```
$> cd ~/Programs/llvm # that's where I have downloaded it.

$> mkdir build

$> ../configure

$> make -j16          # Assuming you have more than 1 core.
```

Optimizations in Practice

- The **opt** tool, available in the LLVM toolbox, performs machine independent optimizations.
- There are many optimizations available through opt.
  - To have an idea, type `opt --help`.



Optimizations in Practice

```
$> opt --help
  Optimizations available:
    -adce                        - Aggressive Dead Code Elimination
    -always-inline               - Inliner for always_inline functions
    -break-crit-edges            - Break critical edges in CFG
    -codegenprepare              - Optimize for code generation
    -constmerge                  - Merge Duplicate Global Constants
    -constprop                   - Simple constant propagation
    -correlated-propagation      - Value Propagation
    -dce                         - Dead Code Elimination
    -deadargelim                 - Dead Argument Elimination
    -die                         - Dead Instruction Elimination
    -dot-cfg                     - Print CFG of function to 'dot' file
    -dse                         - Dead Store Elimination
    -early-cse                   - Early CSE
    -globaldce                   - Dead Global Elimination
    -globalopt                   - Global Variable Optimizer
    -gvn                         - Global Value Numbering
    -indvars                     - Induction Variable Simplification
    -instcombine                 - Combine redundant instructions
    -instsimplify                - Remove redundant instructions
    -ipconstprop                 - Interprocedural constant propagation
    -loop-reduce                 - Loop Strength Reduction
    -reassociate                 - Reassociate expressions
    -reg2mem                     - Demote all values to stack slots
    -sccp                        - Sparse Conditional Constant Propagation
    -scev-aa                     - ScalarEvolution-based Alias Analysis
    -simplifycfg                 - Simplify the CFG
    ...
```

What do you think each of these optimizations do?

Levels of Optimizations

- Like gcc, clang supports different levels of optimizations, e.g., -O0 (default), -O1, -O2 and -O3.
- To find out which optimization each level uses, you can try:

llvm-as is the LLVM assembler. It reads a file containing human-readable LLVM assembly language, translates it to LLVM bytecode, and writes the result into a file or to standard output.

```
$> llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments
```

In my system (LLVM/Darwin), -O1 gives me:

-targetlibinfo -no-aa -tbaa -basicaa -notti -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -always-inline -functionattrs -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -preverify -domtree -verify

Levels of Optimizations

- Like gcc, clang supports different levels of optimizations, e.g., -O0 (default), -O1, -O2 and -O3.
- To find out which optimization each level uses, you can try:

> Can you guess why the **same** analysis or optimization may run more than once?

```
$> llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments
```

In my system (LLVM/Darwin), -O1 gives me:

-targetlibinfo -no-aa -tbaa -basicaa -notti -globalopt -ipsccp -deadargelim -**instcombine** -simplifycfg -basiccg -prune-eh -inline-cost -always-inline -functionattrs -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -**instcombine** -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -**instcombine** -scalar-evolution -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -preverify -domtree -verify

Virtual Register Allocation

- One of the most basic optimizations that opt performs is to map memory slots into variables.
- This optimization is very useful, because the clang front end maps every variable to memory:

```c
void main() {
  int c1 = 17;
  int c2 = 25;
  int c3 = c1 + c2;
  printf("Value = %d\n", c3);
}
```

```
$> clang -c -emit-llvm const.c -o const.bc

$> opt –view-cfg const.bc
```

```
%0:
 %1 = alloca i32, align 4
 %c1 = alloca i32, align 4
 %c2 = alloca i32, align 4
 %c3 = alloca i32, align 4
 store i32 0, i32* %1
 store i32 17, i32* %c1, align 4
 store i32 25, i32* %c2, align 4
 %2 = load i32* %c1, align 4
 %3 = load i32* %c2, align 4
 %4 = add nsw i32 %2, %3
 store i32 %4, i32* %c3, align 4
 %5 = load i32* %c3, align 4
 %6 = call @printf(...)
 %7 = load i32* %1
 ret i32 %7
```

CFG for 'main' function

Virtual Register Allocation

- One of the most basic optimizations that opt performs is to map memory slots into variables.
- We can map memory slots into registers with the `mem2reg` pass:

```
void main() {
  int c1 = 17;
  int c2 = 25;
  int c3 = c1 + c2;
  printf("Value = %d\n", c3);
}
```

How could we further optimize this program?

```
$> opt -mem2reg const.bc > const.reg.bc

$> opt –view-cfg const.reg.bc
```

```
%0:
 %1 = add nsw i32 17, 25
 %2 = call @printf(...), i32 %1)
 ret i32 0
```

CFG for 'main' function

Constant Propagation

- We can fold the computation of expressions that are known at compilation time with the `constprop` pass.

```
%0:
 %1 = add nsw i32 17, 25
 %2 = call @printf(...), i32 %1)
 ret i32 0
```

CFG for 'main' function

```
%0:
 %1 = call i32 (i8*, ...)* @printf(..., i32 42)
 ret i32 0
```

CFG for 'main' function

```
$> opt -constprop const.reg.bc > const.cp.bc

$> opt –view-cfg const.cp.bc
```

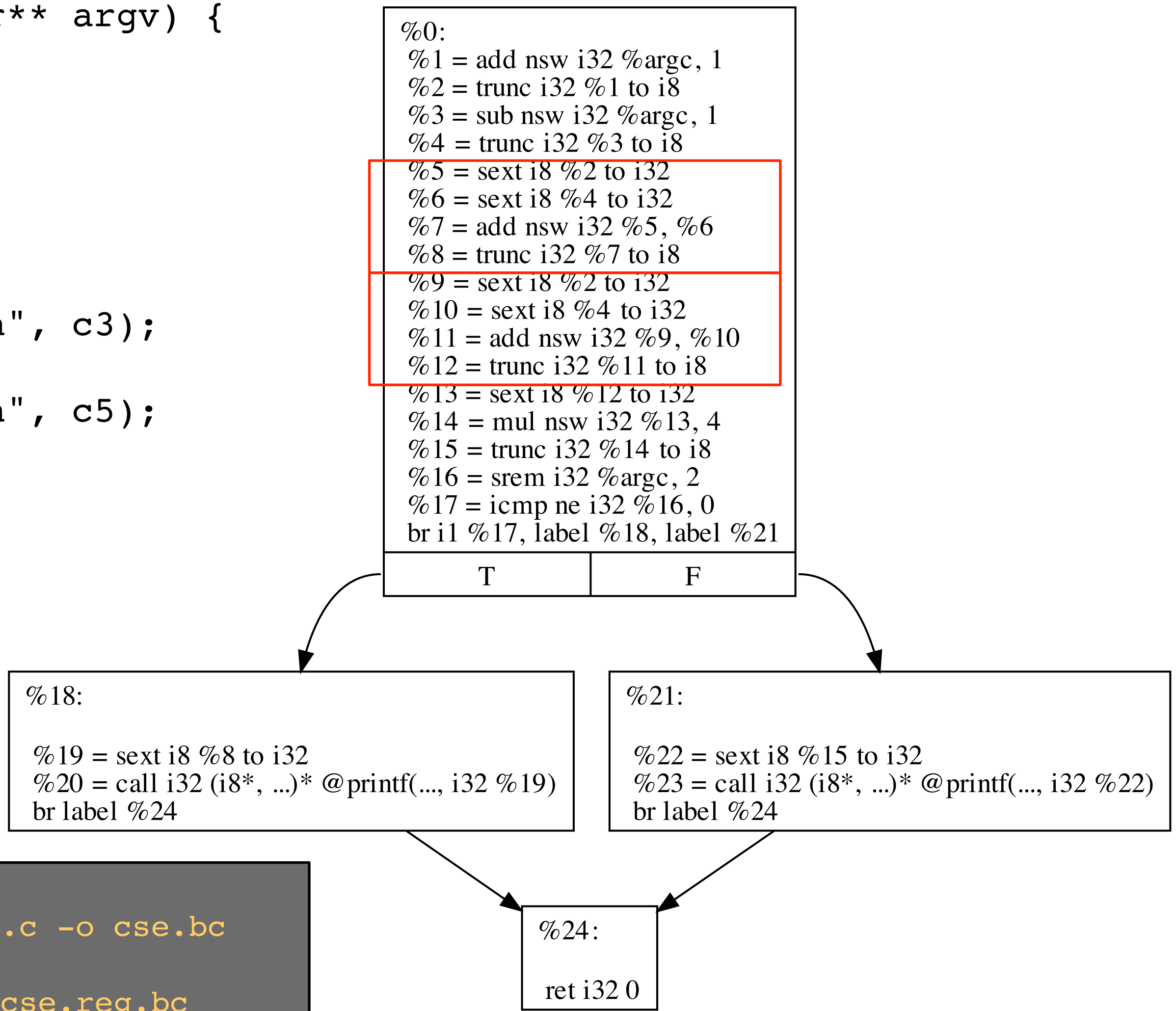
What is %1 in the left CFG? And what is i32 42 in the CFG on the right side?

One more: Common Subexpression Elimination

```
void main(int argc, char** argv) {
  char c1 = argc + 1;
  char c2 = argc - 1;
  char c3 = c1 + c2;
  char c4 = c1 + c2;
  char c5 = c4 * 4;
  if (argc % 2)
    printf("Value = %d\n", c3);
  else
    printf("Value = %d\n", c5);
}
```

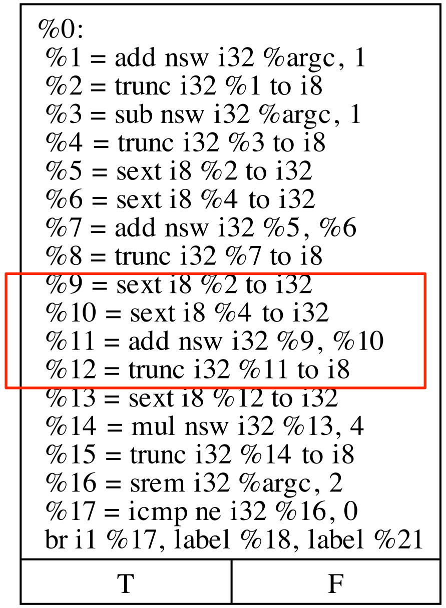
How could we optimize this program?

```
$> clang -c -emit-llvm cse.c -o cse.bc

$> opt –mem2reg cse.bc -o cse.reg.bc

$> opt –view-cfg cse.reg.bc
```

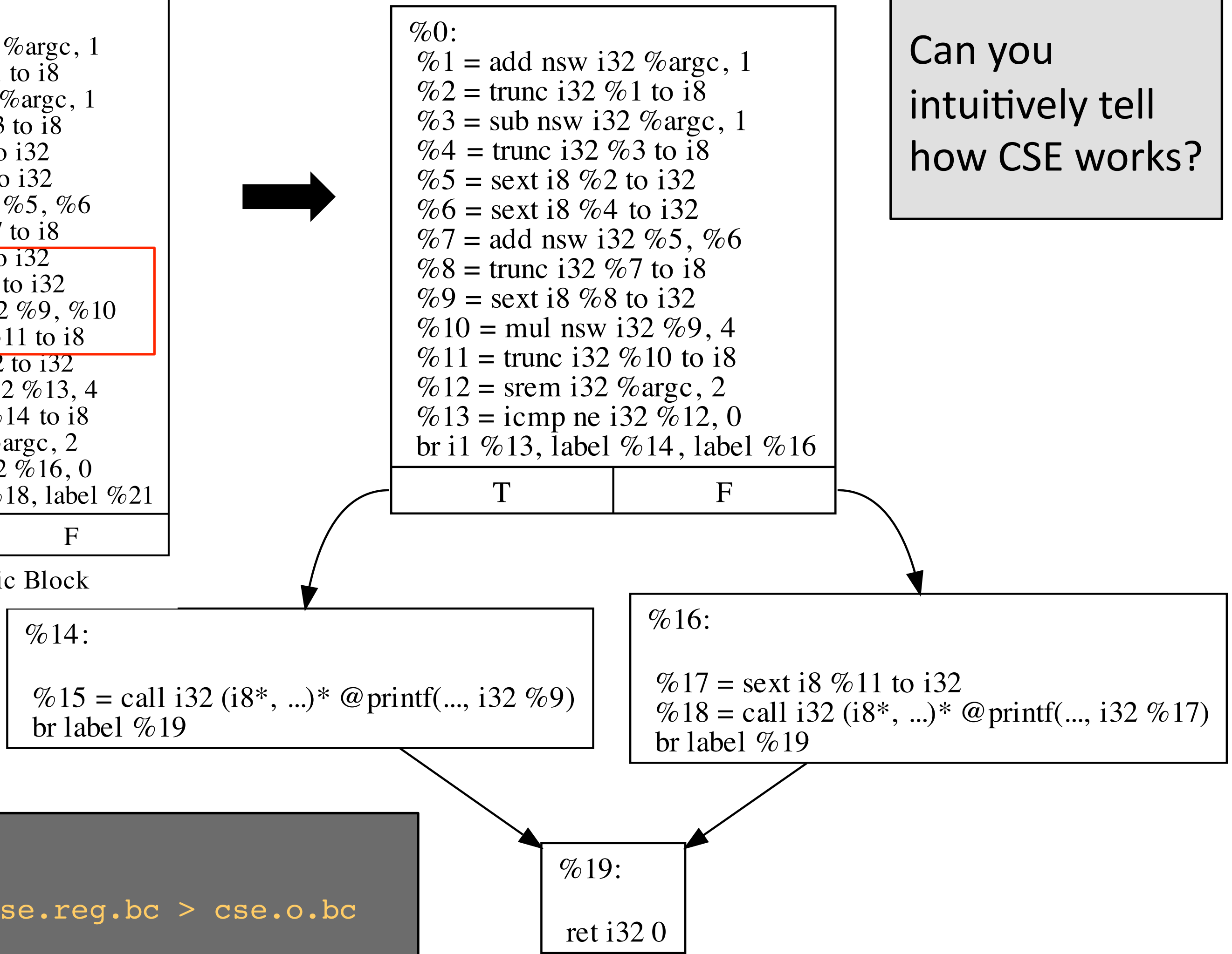


CFG for 'main' function

One more: Common Subexpression Elimination



Original Basic Block



CFG for 'main' function

Can you intuitively tell how CSE works?

```
$> opt -early-cse cse.reg.bc > cse.o.bc

$> opt –view-cfg cse.o.bc
```

LLVM Provides an Intermediate Representation

- LLVM represents programs, internally, via its own instruction set.
  - The LLVM optimizations manipulate these bytecodes.
  - We can program directly on them.
  - We can also interpret them.

♤
```
int callee(const int* X) {
  return *X + 1;
}

int main() {
  int T = 4;
  return callee(&T);
}
```

```
$> clang –c –emit-llvm f.c –o f.bc

$> opt –mem2reg f.bc –o f.bc

$> llvm-dis f.bc

$> cat f.ll
```

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %0 = load i32* %X, align 4
  %add = add nsw i32 %0, 1
  ret i32 %add
}
```

♤: Example taken from the slides of Gennady Pekhimenko "The LLVM Compiler Framework and Infrastructure"

LLVM Bytecodes are Interpretable

- Bytecode is a form of instruction set designed for efficient execution by a software interpreter.
  - They are portable!
  - Example: Java bytecodes.
- The tool **lli** directly executes programs in LLVM bitcode format.
  - lli may compile these bytecodes just-in-time, if a JIT is available.

```
$> echo "int main() {printf(\"Oi\n\");}" > t.c

$> clang -c -emit-llvm t.c -o t.bc

$> lli t.bc
```

How Does the LLVM IR Look Like?

- RISC instruction set, with usual opcodes
  - add, mul, or, shift, branch, load, store, etc
- Typed representation.

```
%0 = load i32* %X, align 4
%add = add nsw i32 %0, 1
ret i32 %add
```

- Static Single Assignment format
  - Each variable noun has only one definition in the program code.
- Control flow is represented explicitly.

This is C

```
switch(argc) {
  case 1: x = 2;
  case 2: x = 3;
  case 3: x = 5;
  case 4: x = 7;
  case 5: x = 11;
  default: x = 1;
}
```

This is LLVM

```
switch i32 %0, label %sw.default [
  i32 1, label %sw.bb
  i32 2, label %sw.bb1
  i32 3, label %sw.bb2
  i32 4, label %sw.bb3
  i32 5, label %sw.bb4
]
```

We can program directly on the IR♧

This is C

```
int callee(const int* X) {
  return *X + 1;
}

int main() {
  int T = 4;
  return callee(&T);
}
```

```
$> clang -c -emit-llvm ex0.c -o
ex0.bc

$> opt -mem2reg -instnamer
ex0.bc -o ex0.bc

$> llvm-dis < ex0.bc
```

This is LLVM

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %tmp = load i32* %X, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %T = alloca i32, align 4
  store i32 4, i32* %T, align 4
  %call = call i32 @callee(i32* %T)
  ret i32 %call
}
```

Which optimization could we apply on this code?

♧: although this is not something to the faint of heart.

Hacking the Bytecode File

<u>This is the original bytecode</u>

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %tmp = load i32* %X, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %T = alloca i32, align 4
  store i32 4, i32* %T, align 4
  %call = call i32 @callee(i32* %T)
  ret i32 %call
}
```

<u>This is the optimized bytecode</u>

```
; Function Attrs: nounwind ssp
define i32 @callee(i32 %X) #0 {
entry:
  %add = add nsw i32 %X, 1
  ret i32 %add
}

; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %call = call i32 @callee(i32 4)
  ret i32 %call
}
```

Can you point out the differences between the files?

We can compile and execute the bytecode file:

```
$> clang ex0.hack.ll
$> ./a.out
$> echo $?
$> 5
```

Understanding our Hand Optimization

```
int callee(const int* X){
  return *X + 1;
}
int main() {
  int T = 4;
  return callee(&T);
}
```

```
; Function Attrs: nounwind ssp
define i32 @callee(i32* %X) #0 {
entry:
  %tmp = load i32* %X, align 4
  %add = add nsw i32 %tmp, 1
  ret i32 %add
}
```

```
; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %T = alloca i32, align 4
  store i32 4, i32* %T, align 4
  %call = call i32 @callee(i32* %T)
  ret i32 %call
}
```

```
int callee(int X) {
  return X + 1;
}

int main() {
  int T = 4;
  return callee(T);
}
```

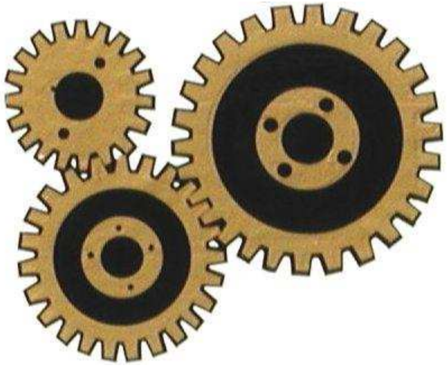
```
; Function Attrs: nounwind ssp
define i32 @callee(i32 %X) #0 {
entry:
  %add = add nsw i32 %X, 1
  ret i32 %add
}
```

```
; Function Attrs: nounwind ssp
define i32 @main() #0 {
entry:
  %call = call i32 @callee(i32 4)
  ret i32 %call
}
```

We did, by hand, some sort of scalarization, i.e., we are replacing pointers with scalars. Scalars are, in compiler jargon, the variables whose values we can keep in registers.

Generating Machine Code

- Once we have optimized the intermediate program, we can translate it to machine code.
- In LLVM, we use the llc tool to perform this translation. This tool is able to target many different architectures.



```
$> llc --version

  Registered Targets:
    alpha      - Alpha [experimental]
    arm        - ARM
    bfin       - Analog Devices Blackfin
    c          - C backend
    cellspu    - STI CBEA Cell SPU
    cpp        - C++ backend
    mblaze     - MBlaze
    mips       - Mips
    mips64     - Mips64 [experimental]
    mips64el   - Mips64el [experimental]
    mipsel     - Mipsel
    msp430     - MSP430 [experimental]
    ppc32      - PowerPC 32
    ppc64      - PowerPC 64
    ptx32      - PTX (32-bit) [Experimental]
    ptx64      - PTX (64-bit) [Experimental]
    sparc      - Sparc
    sparcv9    - Sparc V9
    systemz    - SystemZ
    thumb      - Thumb
    x86        - 32-bit X86: Pentium-Pro
    x86-64     - 64-bit X86: EM64T and AMD64
    xcore      - XCore
```

Generating Machine Code

- Once we have optimized the intermediate program, we can translate it to machine code.
- In LLVM, we use the llc tool to perform this translation. This tool is able to target many different architectures.

```
$> clang -c -emit-llvm identity.c -o identity.bc

$> opt -mem2reg identity.bc -o identity.opt.bc

$> llc -march=x86 identity.opt.bc -o identity.x86
```

```
      .globl      _identity
      .align      4, 0x90
_identity:
      pushl %ebx
      pushl %edi
      pushl %esi
      xorl  %eax, %eax
      movl  20(%esp), %ecx
      movl  16(%esp), %edx
      movl  %eax, %esi
      jmp   LBB1_1
      .align      4, 0x90
LBB1_3:
      movl  (%edx,%esi,4), %ebx
      movl  $0, (%ebx,%edi,4)
      incl  %edi
LBB1_2:
      cmpl  %ecx, %edi
      jl    LBB1_3
      incl  %esi
LBB1_1:
      cmpl  %ecx, %esi
      movl  %eax, %edi
      jl    LBB1_2
      jmp   LBB1_5
LBB1_6:
      movl  (%edx,%eax,4), %esi
      movl  $1, (%esi,%eax,4)
      incl  %eax
LBB1_5:
      cmpl  %ecx, %eax
      jl    LBB1_6
      popl  %esi
      popl  %edi
      popl  %ebx
      ret
```

Final Remarks

- LLVM implements the entire compilation flow.
  - Front-end, e.g., clang & clang++
  - Middle-end, e.g., analyses and optimizations
  - Back-end, e.g., different computer architectures
- LLVM has a high-level intermediate representation.
  - Types, explicit control flow