PROGRAMMING LANGUAGES LABORATORY

Universidade Federal de Minas Gerais - Department of Computer Science

CONTROL FLOW GRAPHS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

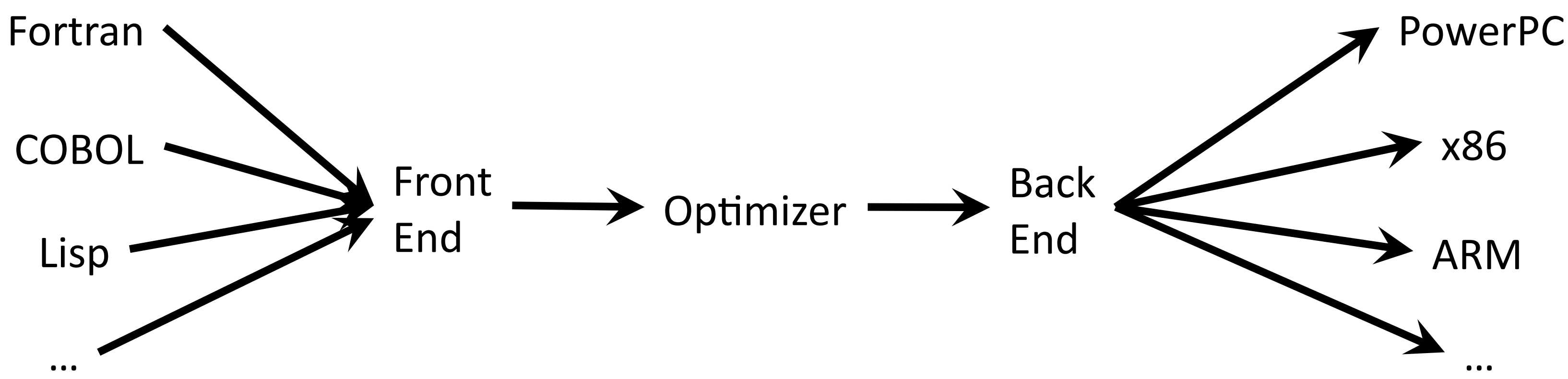
Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from the "Dragon Book", Secs 8.5 – 8.7, by Aho *et al*.

Intermediate Program Representations

- Optimizing compilers and human beings do not see the program in the same way.
  - We are more interested in source code.
  - But, source code is too different from machine code.
  - Besides, from an engineering point of view, it is better to have a common way to represent programs in different languages, and target different architectures.



Basic Blocks and Flow Graphs

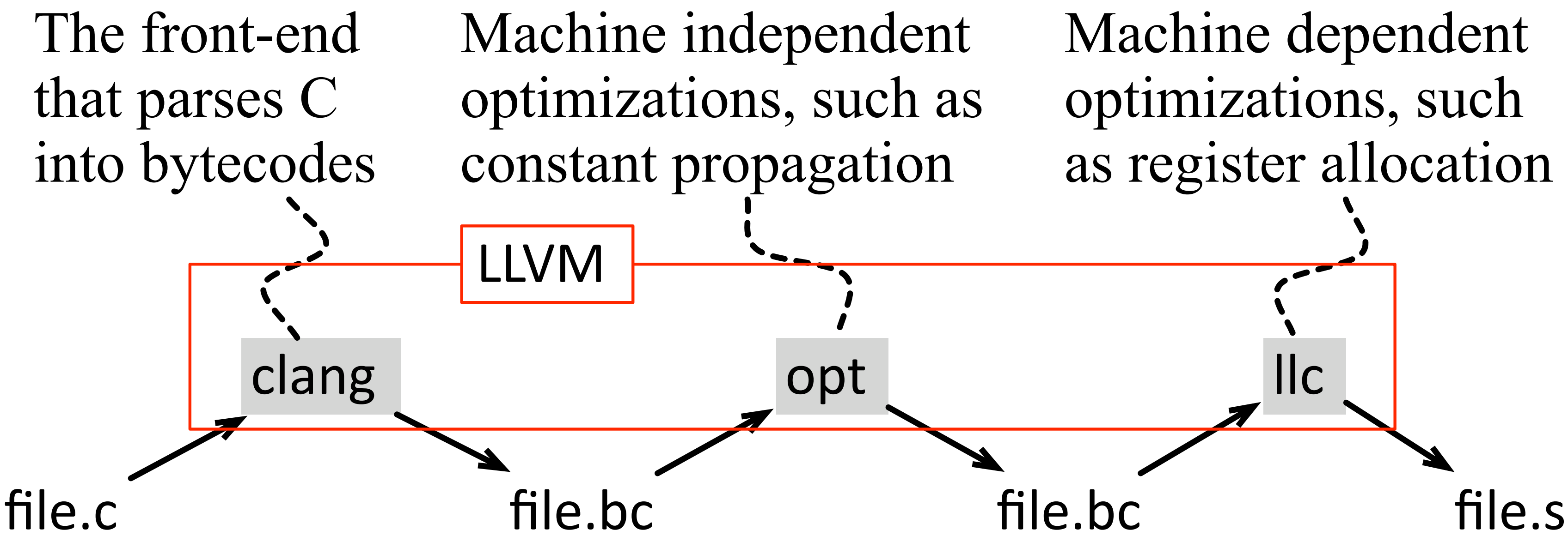
- Usually compilers represent programs as *control flow graphs* (CFG).
- A control flow graph is a directed graph.
  - Nodes are *basic blocks*.
  - There is an edge from basic block B_1 to basic block B_2 if program execution can flow from B_1 to B_2.
- Before defining basic block, we will illustrate this notion by showing the CFG of the function on the right.

```
void identity(int** a, int N) {
  int i, j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      a[i][j] = 0;
    }
  }
  for (i = 0; i < N; i++) {
    a[i][i] = 1;
  }
}
```

What does this program do?

The Low Level Virtual Machine

- We will be working with a compilation framework called The *Low Level Virtual Machine*, or LLVM, for short.
- LLVM is today the most used compiler in research.
- Additionally, this compiler is used in many important companies: Apple, Cray, Google, etc.



Using LLVM to visualize a CFG

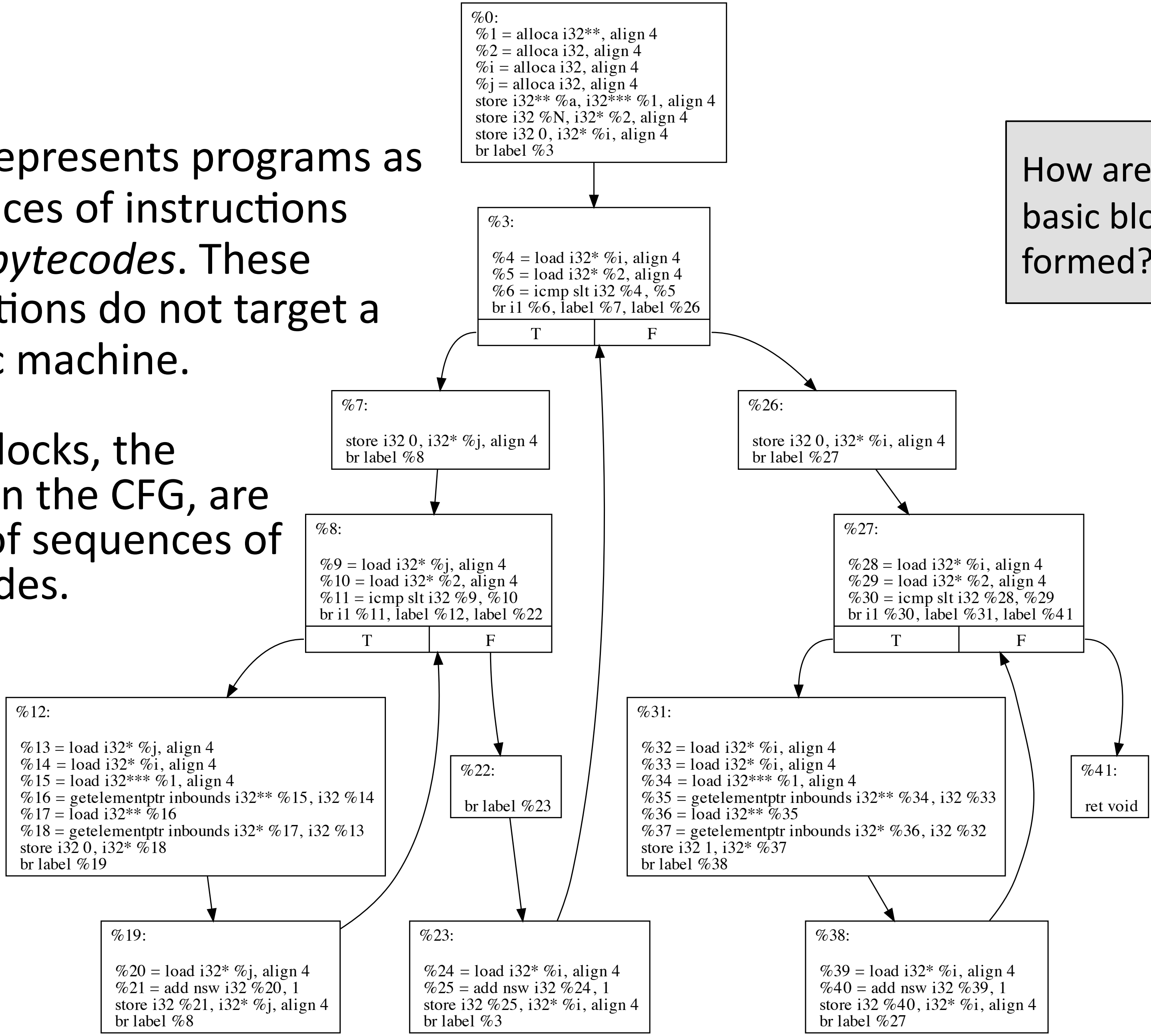
- We can use the opt tool, the LLVM machine independent optimizer, to visualize the control flow graph of a given function

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

$> opt –view-cfg identity.bc
```

- We will be able to see the CFG of our target program, as long as we have the tool DOT installed in our system.

- LLVM represents programs as sequences of instructions called *bytecodes*. These instructions do not target a specific machine.
- Basic blocks, the nodes in the CFG, are made of sequences of bytecodes.



CFG for 'identity' function

How are the basic blocks formed?

Basic Blocks

How can we identify basic blocks from linear sequences of instructions?

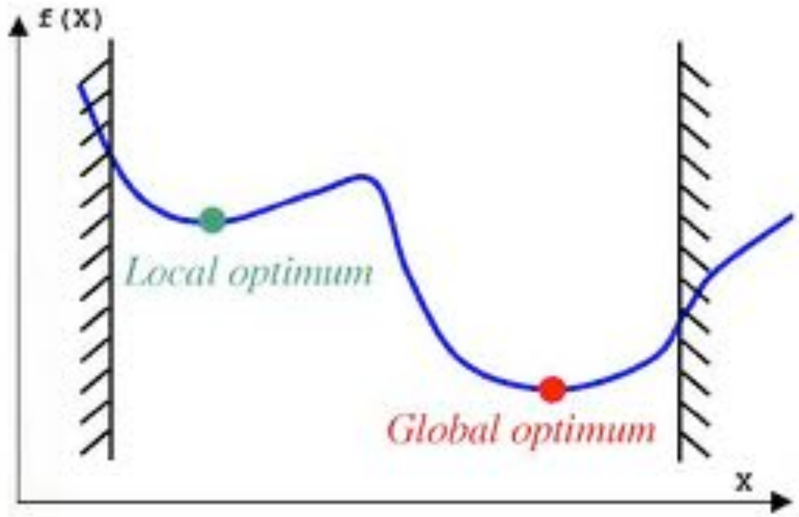
- Basic blocks are maximal sequences of consecutive instructions with the following properties:
  - The flow of control can only enter the basic block through the first instruction in the block.
    - There are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.

```
%8 = load i32* %2, align 4
store i32 %8, i32* %sum, align 4
%9 = load i32* %sum, align 4
%10 = add nsw i32 %9, 1
store i32 %10, i32* %sum, align 4
%11 = load i32* %3, align 4
%12 = load i32* %2, align 4
%13 = mul nsw i32 %11, %12
%14 = load i32* %1, align 4
%15 = sub nsw i32 %13, %14
%16 = load i32* %2, align 4
%17 = load i32* %2, align 4
%18 = mul nsw i32 %16, %17
%19 = add nsw i32 %15, %18
%20 = load i32* %sum, align 4
%21 = add nsw i32 %20, %19
store i32 %21, i32* %sum, align 4
%22 = load i32* %sum, align 4
%23 = add nsw i32 %22, -1
store i32 %23, i32* %sum, align 4
br label %24
```

Identifying Basic Blocks

- The first instruction of a basic block is called a *leader*.
- We can identity leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straightforward to find the basic blocks:
  – For each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

LOCAL OPTIMIZATIONS



DCC 888

Local Optimization

- Code optimization techniques that work in the scope of a basic block are called *local optimizations*.
  - DAG based optimizations
  - Peephole optimizations
  - Local register allocation
- Code optimization techniques that need to analyze the entire control flow graph of a program are called *global optimizations*.
  - Most of the optimizations that we will see in this course are global optimizations.

1) Can you think about a concrete local optimization?

2) Can you think about any global optimization?

3) Can you think about any larger scope of optimization?

DAG-Based Optimizations

- Some of the earliest code optimizations that compilers have performed would rely on a directed acyclic representation of the instructions in the basic block. These DAGS were constructed as follows:
  - There is a node in the DAG for each *input value* appearing in the basic block.
  - There is a node associated with each instruction in the basic block.
  - If instruction S uses variables defined in statements S_1, …, S_2, then we have edges from each S_i to S.
  - If a variable is defined in the basic block, but is not used inside it, then we mark it as an *output value*.

Example of DAG Representation

1: $a = b + c$

2: $b = a - d$

3: $c = b + c$

4: $d = a - d$

- In the program on the left, we have that b, c and d are input values, because they are used in the basic block, but are not defined before the first use.
- We say that the definitions of c and d, at lines 3 and 4, are output values, because these definitions are not used in the basic block.

What is the DAG representation of this basic block?

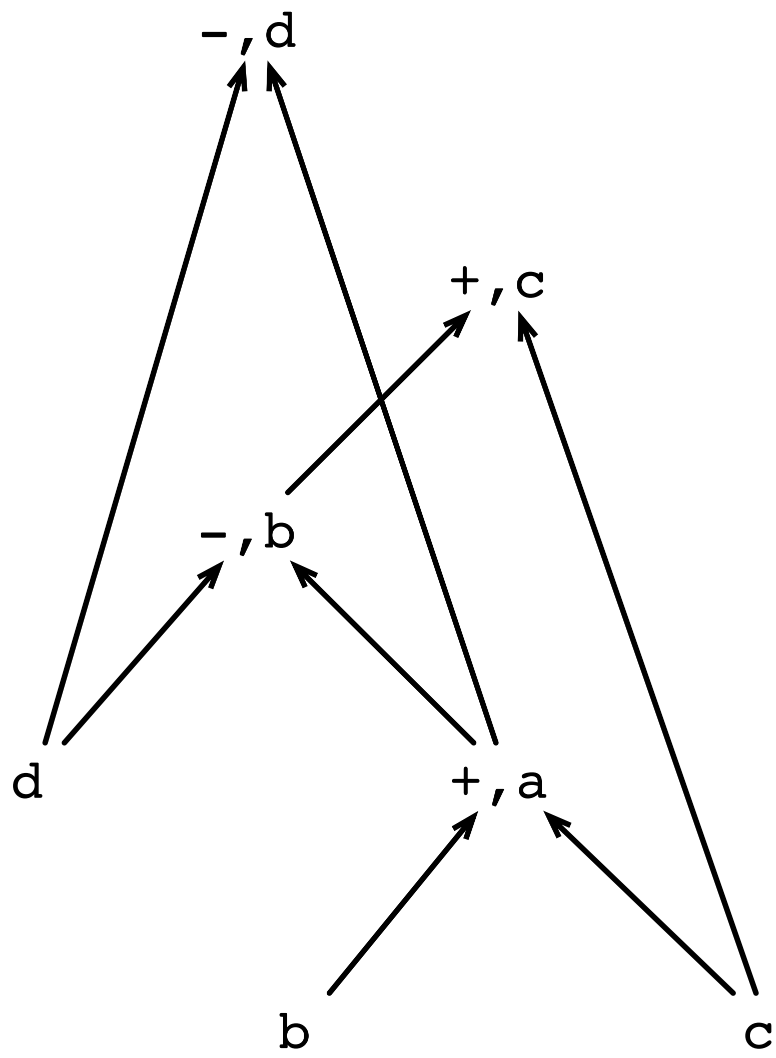
Example of DAG Representation

1: a = b + c

2: b = a – d

3: c = b + c

4: d = a – d



Could you design a simple algorithm to create DAGs from basic blocks?

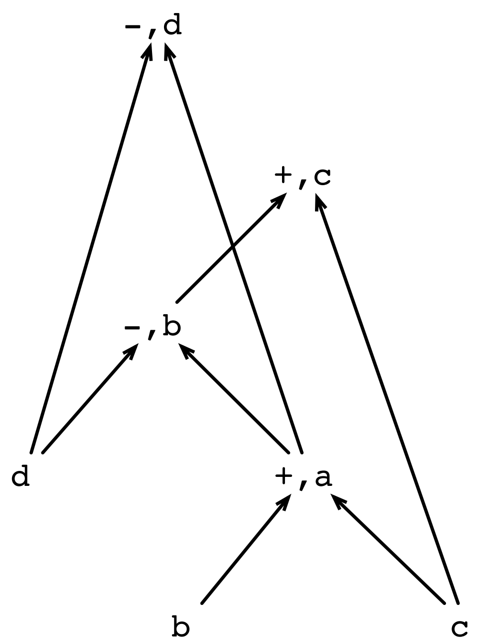
Example of DAG Representation

1: a = b + c

2: b = a - d

3: c = b + c

4: d = a - d



- For each input value v_i:
  - create a node v_i in the DAG
  - label this node with the tag ***in***
- For each statement $v = \boldsymbol{f}(v_1, ..., v_n)$, in the sequence defined by the basic block:
  - create a node v in the DAG
  - create an edge (v_1, v) for each i, $1 \leq i \leq n$
  - label this node with the tag $\boldsymbol{f}$

1) Could you optimize this DAG in particular?

2) Can you think about any optimization that we could perform in general?

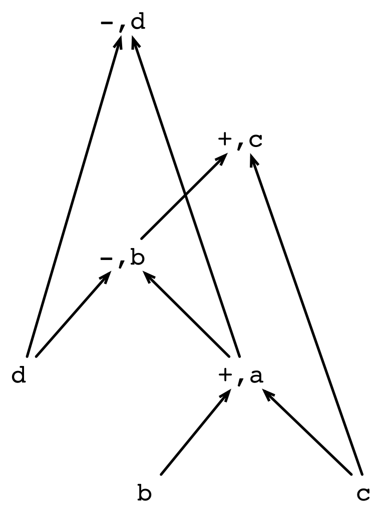
Finding Local Common Subexpressions

1: a = b + c

2: b = a - d

3: c = b + c

4: d = a - d



- For each input value v_i:
  - create a node v_i in the DAG
  - label this node with the tag ***in***
- For each statement $v = \boldsymbol{f}(v_1, ..., v_n)$, in the sequence defined by the basic block:
  - create a node v in the DAG
  - create an edge (v_1, v) for each i, $1 \leq i \leq n$
  - label this node with the tag $\boldsymbol{f}$

How could we adapt our algorithm to reuse expressions that have already been created?

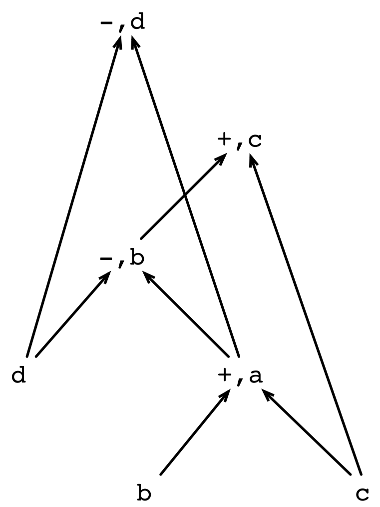
Finding Local Common Subexpressions

1: a = b + c

2: b = a - d

3: c = b + c

4: d = a - d



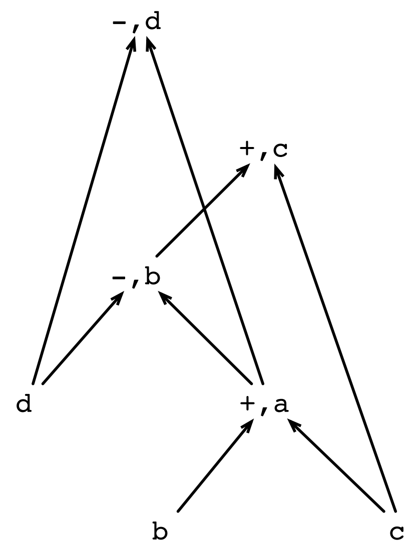
1. For each input value v_i:
    1. create a node v_i in the DAG
    2. label this node with the tag ***in***

2. For each statement $v = \boldsymbol{f}(v_1, ..., v_n)$, in the sequence defined by the basic block:
    1. If the DAG already **contains** a node v' labeled f, with all the children $(v_1, ..., v_n)$ in the order given by i, $1 \leq i \leq n$
        1. Let v' be an alias of v in the DAG
    2. else:
        1. create a node v in the DAG
        2. create an edge (v_1, v) for each i, $1 \leq i \leq n$
        3. label this node with the tag $\boldsymbol{f}$

How can we check **this** property?

Value Numbers

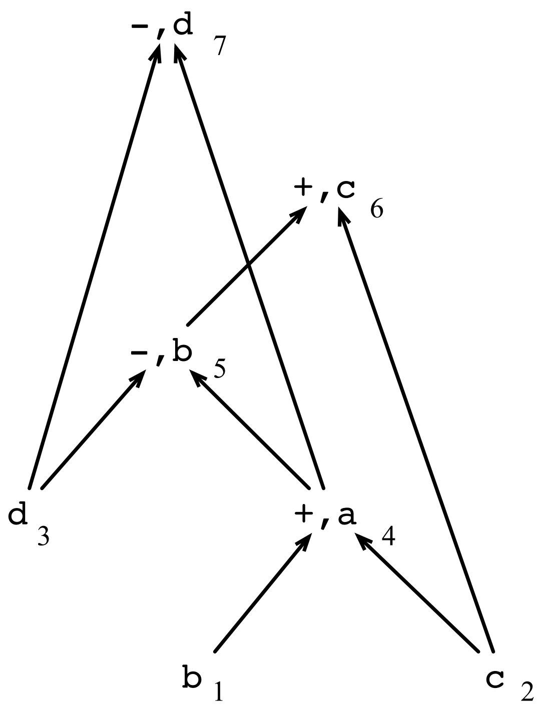
- We can associate each node in the DAG with a signature (lb, v_1, …., v_n), where lb is the label of the node, and each v_i, $1 \leq i \leq n$ *is* a child of the node
  - We can use this signature as the key of a *hash-function*
  - The value produced by the hash-function is called the *value-number* of that variable
- Thus, whenever we build a new node to our DAG, e.g., step 2.1 of our algorithm, we perform a search in the hash-table. If the node is already there, we simply return a reference to it, instead of creating a new node.



What is the result of optimizing this DAG?

Value Numbers

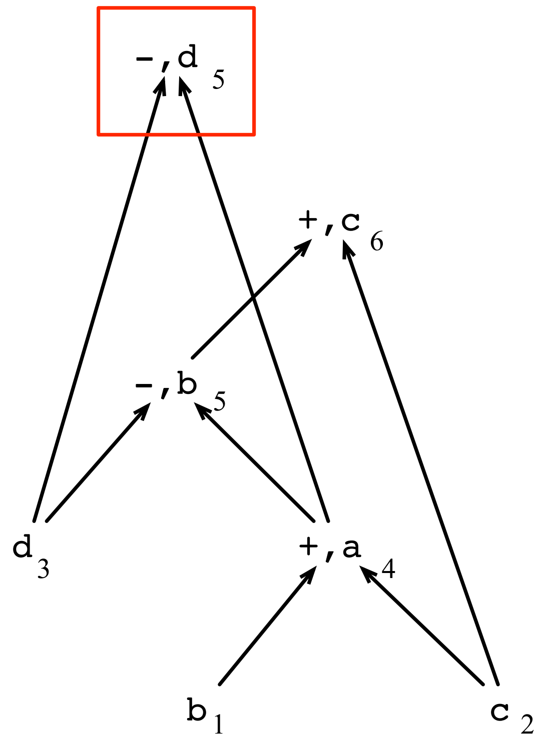
- Value numbers allows us to refer to nodes in our DAG by their semantics, instead of by their textual program representation.



Original DAG

1 (b)	(in, _)
2 (c)	(in, _)
3 (d)	(in, _)
4 (a = b + c)	(+, 1, 2)
5 (b = d - a)	(-, 3, 4)
6 (c = b + c)	(+, 5, 2)
7 (d = d - a)	(-, 3, 4)

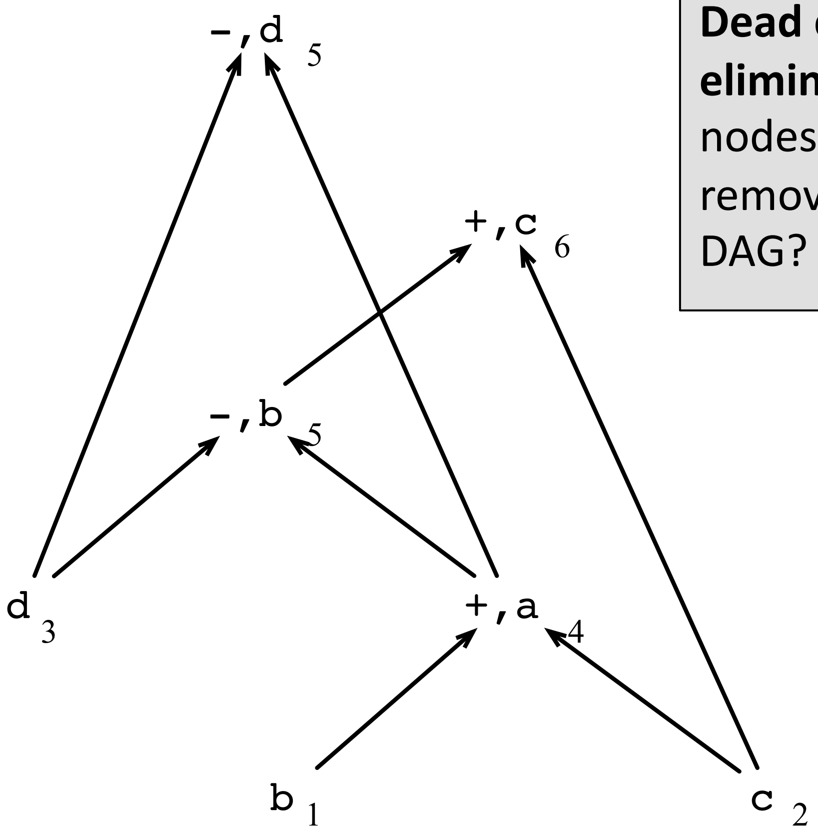
Value-Number Table



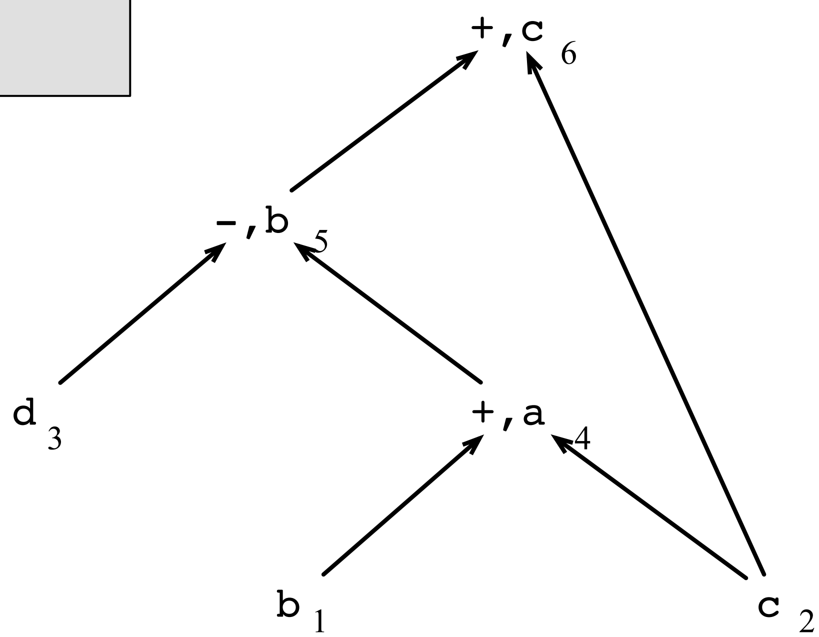
Optimized DAG

Value Numbers

Dead code elimination: which nodes can be safely removed from the DAG?



Original DAG



Optimized DAG

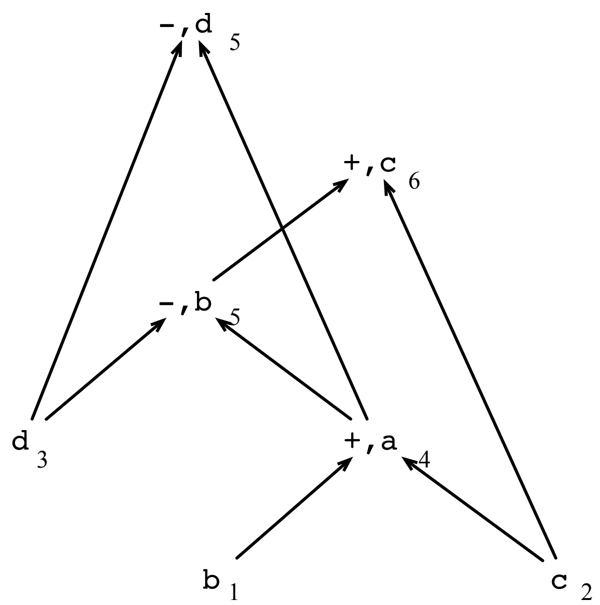
Finding more identities

- We can use several tricks to find common subexpressions in DAGs
  - Commutativity: the value number of $x + y$ and $y + x$ should be the same.
  - Identities: the comparison $x < y$ can often be implemented by $t = x - y; t < 0$
  - Associativity:

$$
\begin{array}{l}
a = b + c \\
t = c + d \\
e = t + b
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
a = b + c \\
e = a + d
\end{array}
$$

Dead Code Elimination

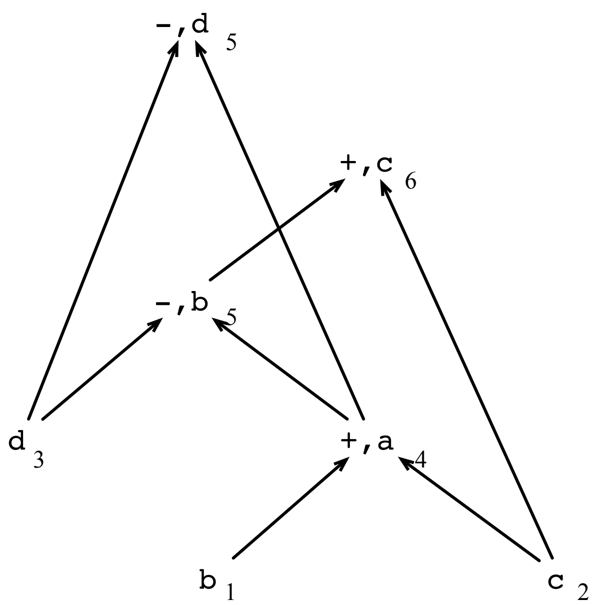
- We can eliminate any node if:
  - This node has no ancestors, e.g., it is a *root node*.
  - This node is not marked as an *output node*.
- We can iterate this pattern of eliminations until we have no more nodes that we can remove.

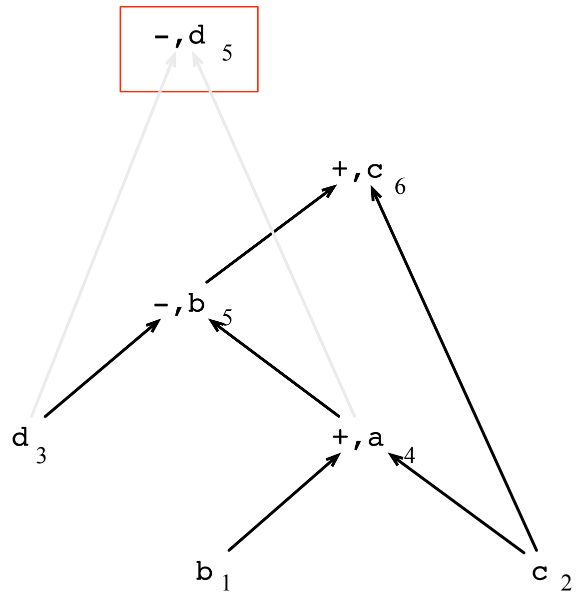


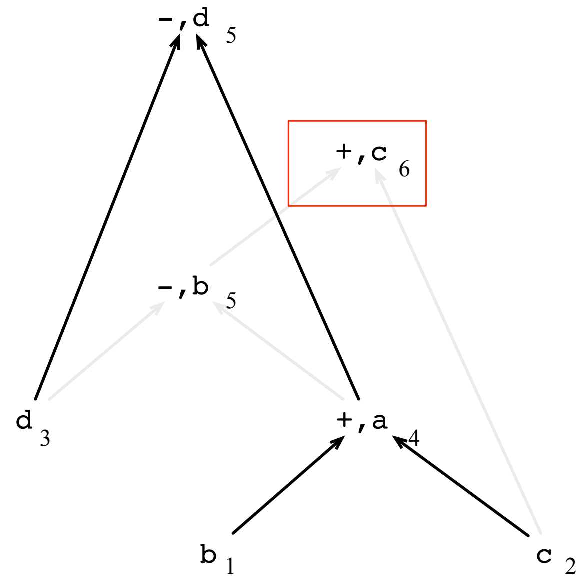
- Which nodes could we eliminate if (-, d) were not an output node?
- Which nodes could we eliminate if (+, c) were not an output node?

Dead Code Elimination

- We can eliminate any node if:
  – This node has no ancestors, e.g., it is a *root node*.
  – This node is not marked as an *output node*.
- We can iterate this pattern of eliminations until we have no more nodes that we can remove.







Algebraic Identities

- We can explore algebraic identities to optimize DAGs
  - Arithmetic Identities:
    - $x + 0 = 0 + x = x$; $x * 1 = 1 * x = x$; $x - 0 = x$; $x/1 = x$
  - Reduction in strength:
    - $x^2 = x * x$; $2 * x = x + x$; $x / 2 = x * 0.5$
  - Constant folding:
    - evaluate expressions at compilation time, replacing each expression by its value

Reduction in strength optimizes code by replacing some sequences of instructions by others, which can be computed more efficiently.

Could you provide a rational for each of these reductions in strength?

Peephole Optimizations

How to find the best window size?

- Peephole optimizations are a category of local code optimizations.
- The principle is very simple:
  – the optimizer analyzes sequences of instructions.
  – only code that is within a small window of instructions is analyzed each time.
  – this window slides over the code.
  – once patterns are discovered inside this window, optimizations are applied.

```
%8 = load i32* %2, align 4
store i32 %8, i32* %sum, align 4
%9 = load i32* %sum, align 4
%10 = add nsw i32 %9, 1
store i32 %10, i32* %sum, align 4
%11 = load i32* %3, align 4
%12 = load i32* %2, align 4
%13 = mul nsw i32 %11, %12
%14 = load i32* %1, align 4
%15 = sub nsw i32 %13, %14
%16 = load i32* %2, align 4
%17 = load i32* %2, align 4
%18 = mul nsw i32 %16, %17
%19 = add nsw i32 %15, %18
%20 = load i32* %sum, align 4
%21 = add nsw i32 %20, %19
store i32 %21, i32* %sum, align 4
%22 = load i32* %sum, align 4
%23 = add nsw i32 %22, -1
store i32 %23, i32* %sum, align 4
br label %24
```

Redundant Loads and Stores

- Some memory access patterns are clearly redundant:

```
load R0, m
store m, RO
```

1) This code is very naïve. How could it appear in actual assembly programs?

- Patterns like this can be easily eliminated by a peephole optimizer.

2) Why is this optimization only safe inside a basic block?

Branch Transformations

- Some branches can be rewritten into faster code. For instance:

```
if debug == 1 goto L1
goto L2
L1: …
L2: …
```

```
if debug != 1 goto L2
L1: …
L2: …
```

- This optimization crosses the boundaries of basic blocks, but it is still performed on a small sliding window.

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

➡

```
    goto L2
    ...
L1: goto L2
```

Why is the code on the right faster?

Could we remove the third line altogether?

Jumps to Jumps

- How could we optimize this code sequence?

```
    goto L1
    ...
L1: goto L2
```

➡

```
    goto L2
    ...
L1: goto L2
```

➡

```
goto L2
...
```

- We can eliminate the second jump, provided that we know that there is no jump to L1 in the rest of the program.
  - Why?
- Notice that this peephole optimization requires some previous information gathering: we must know which instructions are targets of jumps.

Jumps to Jumps

- How could we optimize this code sequence?

We saw how to optimize the sequence on the right. Does the same optimization applies on the code below?

```
    goto L1
    ...
L1: goto L2
```

```
    if a < b goto L1
    ...
L1: goto L2
```

Jumps to Jumps

- How could we optimize this code sequence?
  - Under which assumptions is your optimization valid?

```
     goto L1
     ...
L1:  if a < b goto L2
L3:
```

Jumps to Jumps

- How could we optimize this code sequence?
  – Under which assumptions is your optimization valid?

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

```
    if a < b goto L2
    goto L3
    ...
L3:
```

- In order to apply this optimization, we need to make sure that:
  - There is no jump to L1
  - L1 is preceded by an unconditional goto.

Why are these assumptions necessary?

Reduction in Strength

- Instead of performing reduction in strength at the DAG level, many compilers do it via a peephole optimizer.
  – This optimizer allows us to replace sequences such as 4 * x by, x << 2, for instance.

4 * x is a pattern that is pretty common in loops that range over 32-bit words. Why?

Machine Idioms

- Many computer architectures have efficient instructions to implement some common operations.
  - A typical example is found in the increment and decrement operators:

```
addl $1, %edi    ➡    incl %edi
```

Is there any other machine idioms that we could think about?

Local Register Allocation

- Registers are memory locations which have very fast memory access.
- However, registers exist in small number. For instance, the 32-bits x86 processor has eight visible registers.
- A key optimization consists in mapping the most used variables in a program onto registers.
- Compilers usually need to see the entire function (or even the whole program) to perform a good register allocation.
- But, if we need to do it quickly, we can only look into a basic block. In this case, we are doing *local register allocation*.

1) What are the most used variables in a program?
2) Why is it better to see the entire function, instead of a basic block, to do better register allocation?
3) Why would we be willing to do register allocation quickly?
4) How could we find registers for the variables in a basic block?

Registers vs Memory

```
int arith(int a1, int an, int N) {
  return (a1 + an) * N / 2;
}
```

```
_arith_all_mem:
## BB#0:
subl    $12, %esp
movl    16(%esp), %eax
movl    %eax, 8(%esp)
movl    20(%esp), %eax
movl    %eax, 4(%esp)
movl    24(%esp), %eax
movl    %eax, (%esp)
movl    8(%esp), %ecx
addl    4(%esp), %ecx
imull   %eax, %ecx
movl    %ecx, %eax
shrl    $31, %eax
addl    %ecx, %eax
sarl    %eax
addl    $12, %esp
ret
```

```
_arith_reg_alloc:
## BB#0:
movl    4(%esp), %ecx
addl    8(%esp), %ecx
imull   12(%esp), %ecx
movl    %ecx, %eax
shrl    $31, %eax
addl    %ecx, %eax
sarl    %eax
ret
```

The program that uses registers is substantially faster than the program that maps all the variables to memory. It is shorter too, as we do not need so many loads and stores to move variables to and from memory.

1) Why we do not need to worry about "memory allocation"?

2) Can you think about an algorithm to find registers for the variables inside a basic block?

Local Register Allocation

```
allocate(Block b) {
 for (Inst i = b.first_inst; i != b.last_inst; i++) {
  for (Operand o = i.first_operand; o != i.last_operand; o++) {
   if (o is in memory m) {
    r = find_reg(i)
    assign r to o
    add "r = load m" before i
   }
  }
  v = i.definition
  r = find_reg(i)
  assign r to v
 }
 for (Operand o = i.first_operand; o != i.last_operand; o++) {
   if (i is the last use of o) {
    return the register bound to o to the list of free registers
   }
  }
}
```

1) Why do we need to insert this load instruction in our code?
2) When are registers mapped into memory?
3) How would be the implementation of `find_reg`?

Spilling

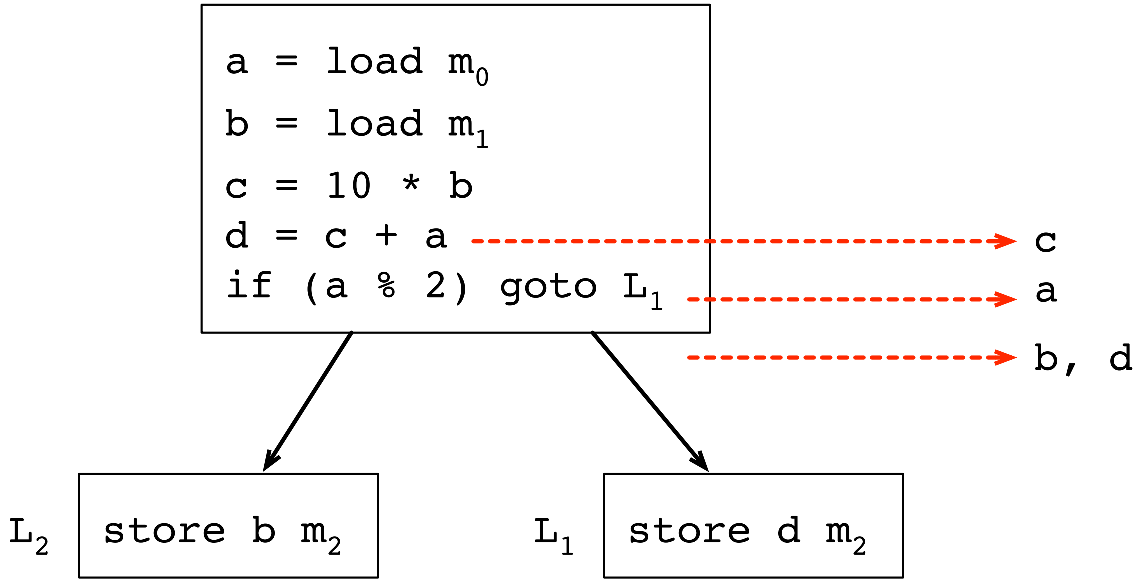
```
find_reg(i) {
  if there is free register r
    return r
  else
    let v be the last used variable after i, that is in a register
      if v is does not have a memory slot
        let m be a fresh memory slot
        add "store v m" right after the definition point of v
      else
        let m be the memory slot assigned to v
    return r
}
```

If the *register pressure* is too high, we may have to evict some variable to memory. This action is called *spilling*. If a variable is mapped into memory, then we call it a *spill*.

1) Why do we spill the variable that has the last use after the current instruction i?
2) Why do we need to insert this store in our code?
3) How many stores can we have per variable?

Spilling the Furthest Use

- Usually we spill the variable that has the furthest use from the spilling point.
- This strategy is called the Belady[♣] Algorithm, and it is also used in operating systems, when deciding which page to evict to disk in the virtual memory system.



But we must be careful when doing local register allocation, because if a variable is used after the basic block, we must assume that it has a use at the end of the block.

♣: A Study of Replacement algorithms for a Virtual Storage Computer (1966) – IBM Systems Journal

A QUICK INTRO TO LLVM

DCC 888

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

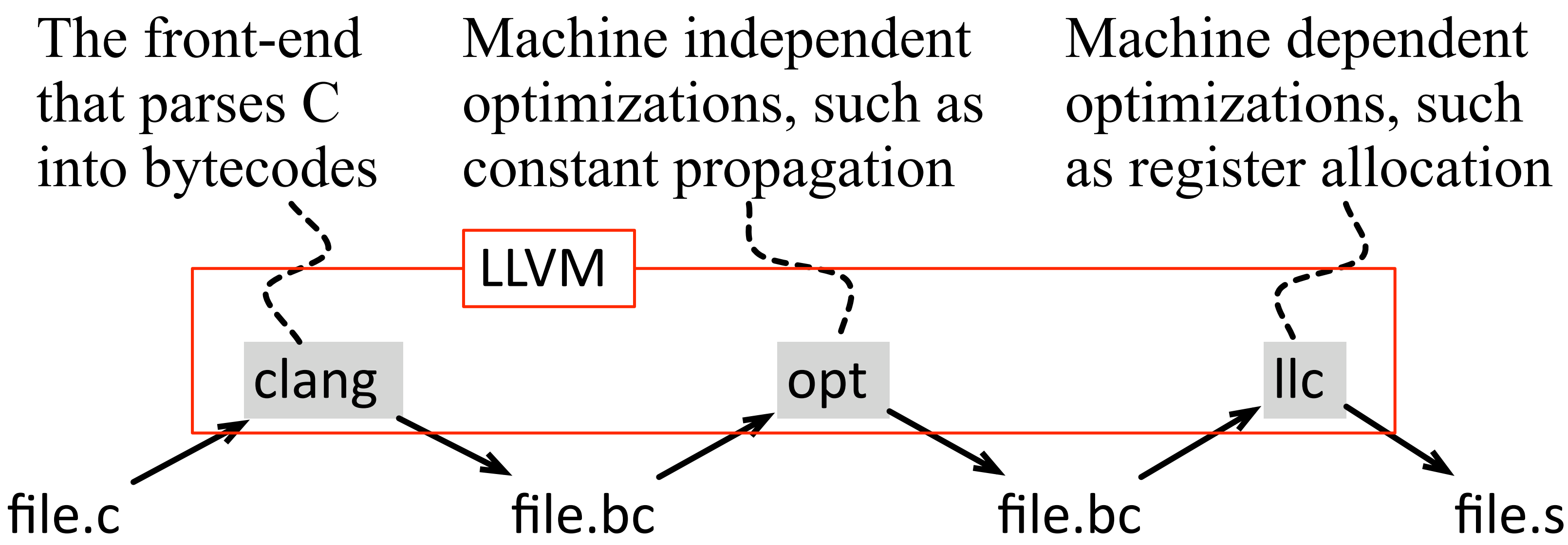
- Compile C/C++ programs:

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

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.

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

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:

```
int 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:

```
int 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

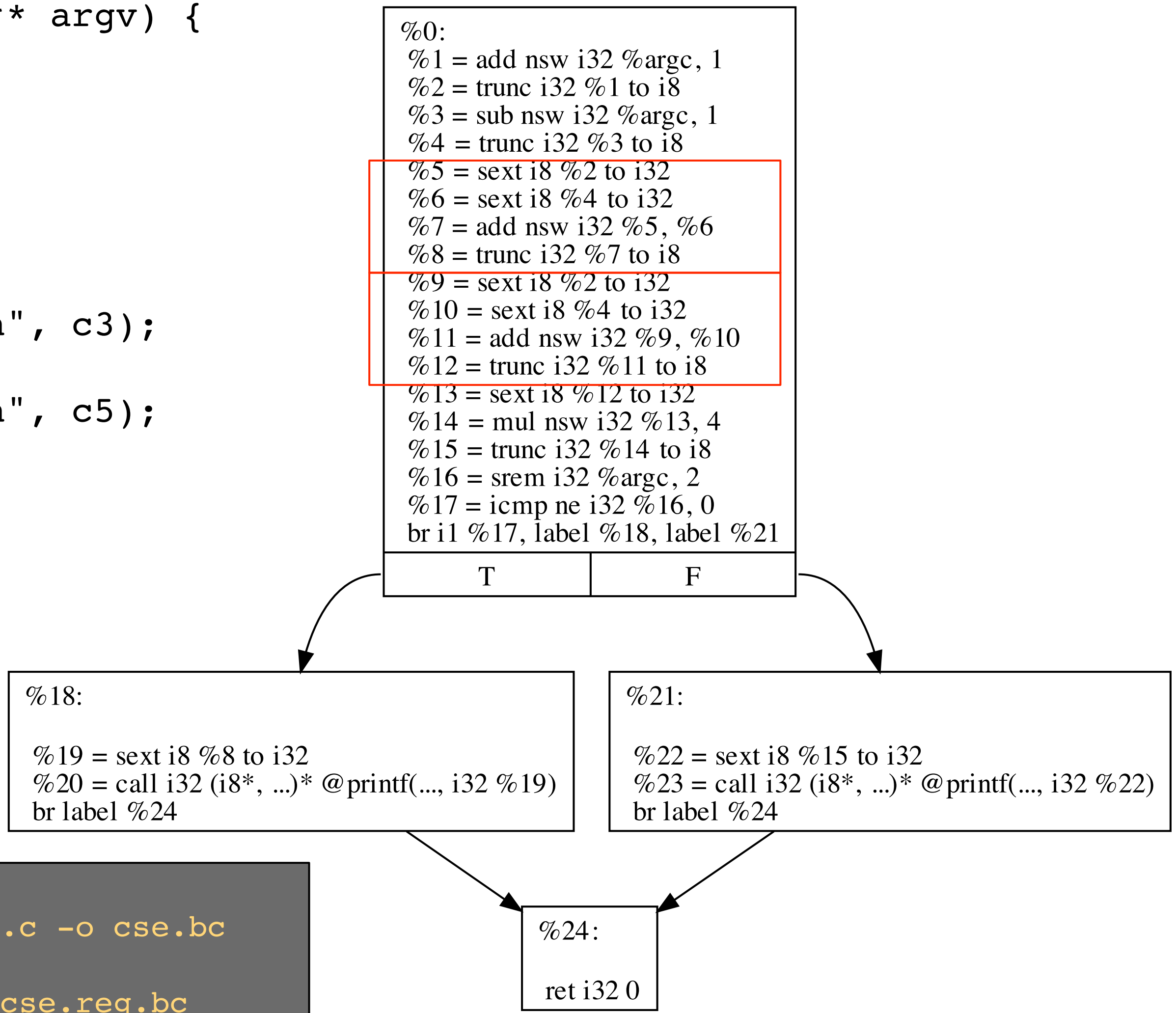
```
int main(int argc, char** argv) {
  char c1 = argc + 1;
  char c2 = argc - 1;
  char c3 = c1 + c2;
  char c4 = c1 + c2;
  char c5 = c4 * 4;
  if (argc % 2)
    printf("Value = %d\n", c3);
  else
    printf("Value = %d\n", c5);
}
```

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

```
%0:
 %1 = add nsw i32 %argc, 1
 %2 = trunc i32 %1 to i8
 %3 = sub nsw i32 %argc, 1
 %4 = trunc i32 %3 to i8
 %5 = sext i8 %2 to i32
 %6 = sext i8 %4 to i32
 %7 = add nsw i32 %5, %6
 %8 = trunc i32 %7 to i8
 %9 = sext i8 %2 to i32
 %10 = sext i8 %4 to i32
 %11 = add nsw i32 %9, %10
 %12 = trunc i32 %11 to i8
 %13 = sext i8 %12 to i32
 %14 = mul nsw i32 %13, 4
 %15 = trunc i32 %14 to i8
 %16 = srem i32 %argc, 2
 %17 = icmp ne i32 %16, 0
 br i1 %17, label %18, label %21
        T              F
```

Original Basic Block

```
%0:
 %1 = add nsw i32 %argc, 1
 %2 = trunc i32 %1 to i8
 %3 = sub nsw i32 %argc, 1
 %4 = trunc i32 %3 to i8
 %5 = sext i8 %2 to i32
 %6 = sext i8 %4 to i32
 %7 = add nsw i32 %5, %6
 %8 = trunc i32 %7 to i8
 %9 = sext i8 %8 to i32
 %10 = mul nsw i32 %9, 4
 %11 = trunc i32 %10 to i8
 %12 = srem i32 %argc, 2
 %13 = icmp ne i32 %12, 0
 br i1 %13, label %14, label %16
        T              F
```

```
%14:

 %15 = call i32 (i8*, ...)* @printf(..., i32 %9)
 br label %19
```

```
%16:

 %17 = sext i8 %11 to i32
 %18 = call i32 (i8*, ...)* @printf(..., i32 %17)
 br label %19
```

```
%19:

 ret i32 0
```

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
```

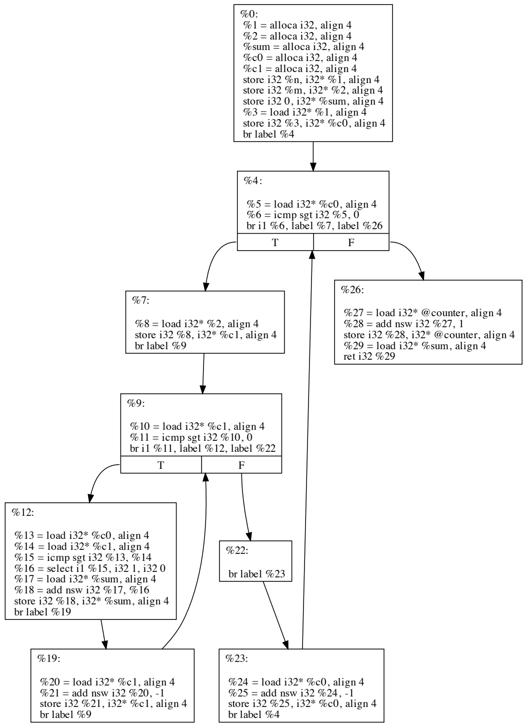
Writing a LLVM pass

- LLVM applies a chain of analyses and transformations on the target program.
- Each of these analyses or transformations is called a *pass*.
- We have seen a few passes already: `mem2reg`, `early-cse` and `constprop`, for instance.
- Some passes, which are machine independent, are invoked by the opt tool.
- Other passes, which are machine dependent, are invoked by the llc tool.
- A pass may require information provided by other passes. Such dependencies must be explicitly stated.
  - For instance: a common pattern is a transformation pass requiring an analysis pass.

Counting Number of Opcodes in Programs

Let's write a pass that counts the number of times that each opcode appears in a given function. This pass must print, for each function, a list with all the instructions that showed up in its code, followed by the number of times each of these opcodes has been used.

```
int foo(int n, int m) {
  int sum = 0;
  int c0;
  for (c0 = n; c0 > 0; c0--) {
    int c1 = m;
    for (; c1 > 0; c1--) {
      sum += c0 > c1 ? 1 : 0;
    }
  }
  counter++;
  return sum;
}
```




```
Function foo
add: 4
alloca: 5
br: 8
icmp: 3
load: 11
ret: 1
select: 1
store: 9
```

Counting Number of Opcodes in Programs

```
#define DEBUG_TYPE "opCounter"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;
namespace {
  struct CountOp : public FunctionPass {
    std::map<std::string, int> opCounter;
    static char ID;
    CountOp() : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
      errs() << "Function " << F.getName() << '\n';
      for (Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
        for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
          if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
            opCounter[i->getOpcodeName()] = 1;
          } else {
            opCounter[i->getOpcodeName()] += 1;
          }
        }
      }
      std::map <std::string, int>::iterator i = opCounter.begin();
      std::map <std::string, int>::iterator e = opCounter.end();
      while (i != e) {
        errs() << i->first << ": " << i->second << "\n";
        i++;
      }
      errs() << "\n";
      opCounter.clear();
      return false;
    }
  };
}
char CountOp::ID = 0;
static RegisterPass<CountOp> X("opCounter", "Counts opcodes per functions");
```

Our pass runs once for each function in the program; therefore, it is a `FunctionPass`. If we had to see the whole program, then we would implement a `ModulePass`.

This line defines the name of the pass, in the command line, e.g., opCounter, and the help string that opt provides to the user about the pass.

A Closer Look into our Pass

```
struct CountOp : public FunctionPass {
   std::map<std::string, int> opCounter;
   static char ID;
   CountOp() : FunctionPass(ID) {}
   virtual bool runOnFunction(Function &F) {
     errs() << "Function " << F.getName() << '\n';
     for (Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
       for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
         if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
           opCounter[i->getOpcodeName()] = 1;
         } else {
           opCounter[i->getOpcodeName()] += 1;
         }
       }
     }
     std::map <std::string, int>::iterator i = opCounter.begin();
     std::map <std::string, int>::iterator e = opCounter.end();
     while (i != e) {
       errs() << i->first << ": " << i->second << "\n";
       i++;
     }
     errs() << "\n";
     opCounter.clear();
     return false;
   }
  };
}
```

We will be recording the number of each opcode in **this** map, that binds opcode names to integer numbers.

This code collects the opcodes. We will look into it more closely soon.

This code prints our results. It is a standard loop on an STL data structure. We use iterators to go over the map. Each element in a map is a pair, where the first element is the key, and the second is the value.

Iterating Through Functions, Blocks and Insts

```
for(Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
  for(BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
    if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
      opCounter[i->getOpcodeName()] = 1;
    } else {
      opCounter[i->getOpcodeName()] += 1;
    }
  }
}
```

We go over LLVM data structures through iterators.

- An **iterator over a Module** gives us a list of Functions.
- An **iterator over a Function** gives us a list of basic blocks.
- An **iterator over a Block** gives us a list of instructions.
- And we can **iterate over the operands** of the instruction too.

```
for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F);

for (User::op_iterator O = I.op_begin(), E = I.op_end(); O != E; ++O);
```

Compiling the Pass

- To Compile the pass, we can follow these two steps:

1. Generally we save the pass into `/llvm/lib/Transforms/DirectoryName`, where `DirectoryName` can be, for instance, `CountOp`.

2. We build a Makefile for the project. If we invoke the LLVM standard Makefile, we save some time.

```
# Path to top level of LLVM hierarchy
LEVEL = ../../..

# Name of the library to build
LIBRARYNAME = CountOp

# Make the shared library become a
# loadable module so the tools can
# dlopen/dlsym on the resulting library.
LOADABLE_MODULE = 1

# Include the makefile implementation
include $(LEVEL)/Makefile.common
```

Running the Pass

- Our pass is now a shared library, in `llvm/Debug/lib`, if we have compiled our LLVM distribution with the –`Debug` directive, or in `llvm/Release/lib`, if we have not.
- We can invoke it using the opt tool:

Just to avoid printing the binary t.bc file

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

$> opt -load CountOp.dylib -opCounter -disable-output t.bc
```

- Remember, if we are running on Linux, then our shared library has the extension "`.so`", instead of "`.dylib`", as in the Mac OS.

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
  - That is something smart, which we will see later in the course.
- 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

```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 opts could we apply on this code?

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

Hacking the Bytecode File

This is the original bytecode

```
; 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
}
```

This is the optimized bytecode

```
; 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 all 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

```
; 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
}
```

```
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:
  %add = add nsw i32 %X, 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 @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
```

A Bit of History

- Some of the first code optimizations were implemented only locally
- Fortran was one of the first programming languages to be optimized by a compiler
- The LLVM infra-structure was implemented by Chris Lattner, during his PhD, at UIUC

- Kam, J. B. and J. D. Ullman, "Monotone Data Flow Analysis Frameworks", Actal Informatica 7:3 (1977), pp. 305-318
- Kildall, G. "A Unified Approach to Global Program Optimizations", ACM Symposium on Principles of Programming Languages (1973), pp. 194-206
- Lattner, C., and Adve, V., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", CGO, pp. 75-88 (2004)