Compiler-Aided Optimization

Andrey Kamaev
Itseez UNN 2016

What is optimization?

*Finding an **alternative** with the most cost effective or highest achievable **performance** under the given **constraints**, by maximizing desired factors and minimizing undesired ones.*

Where get the performance?

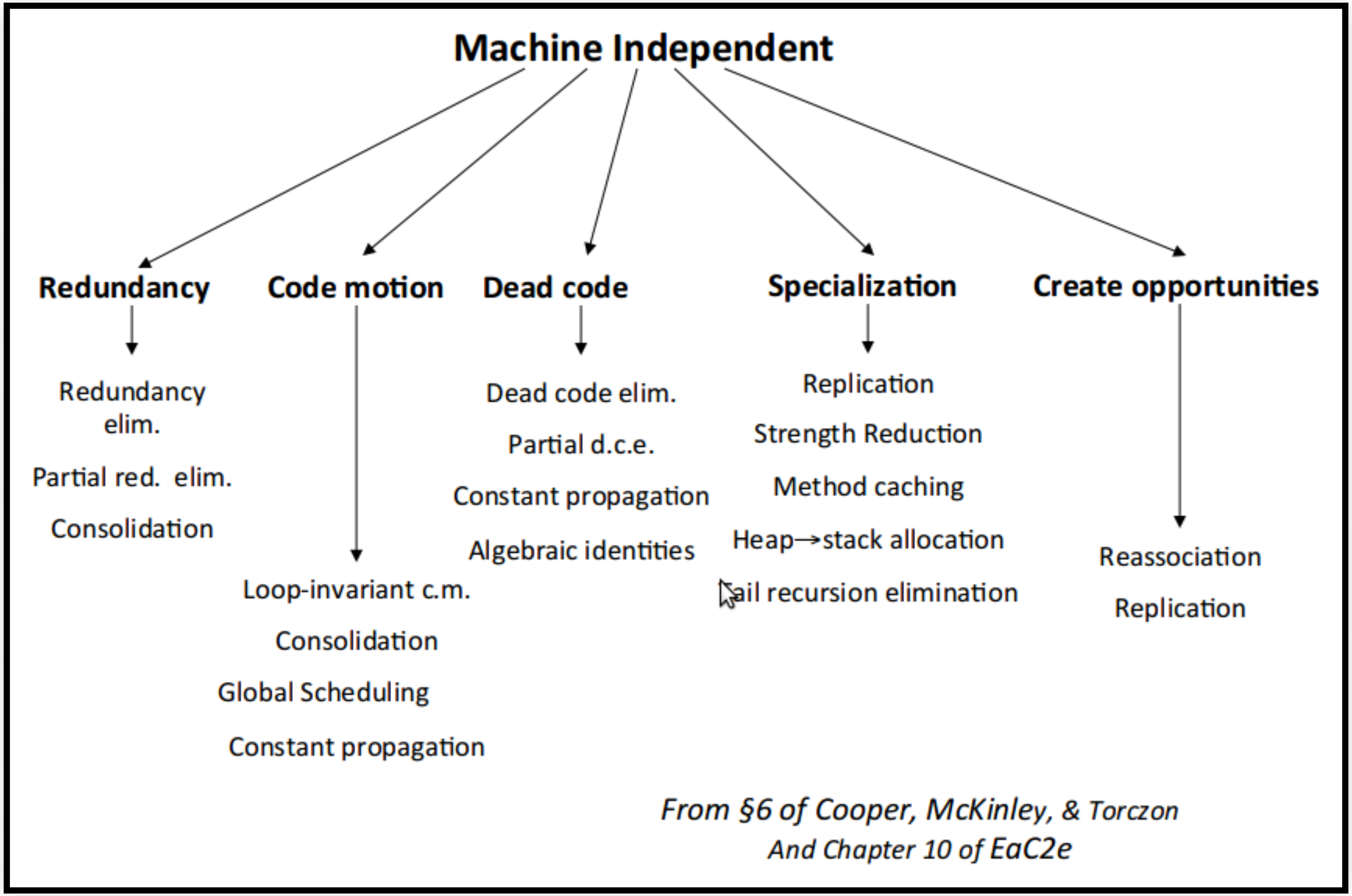
High-level	Programmer
Middle-level	**Compiler**
Low-level	Hardware

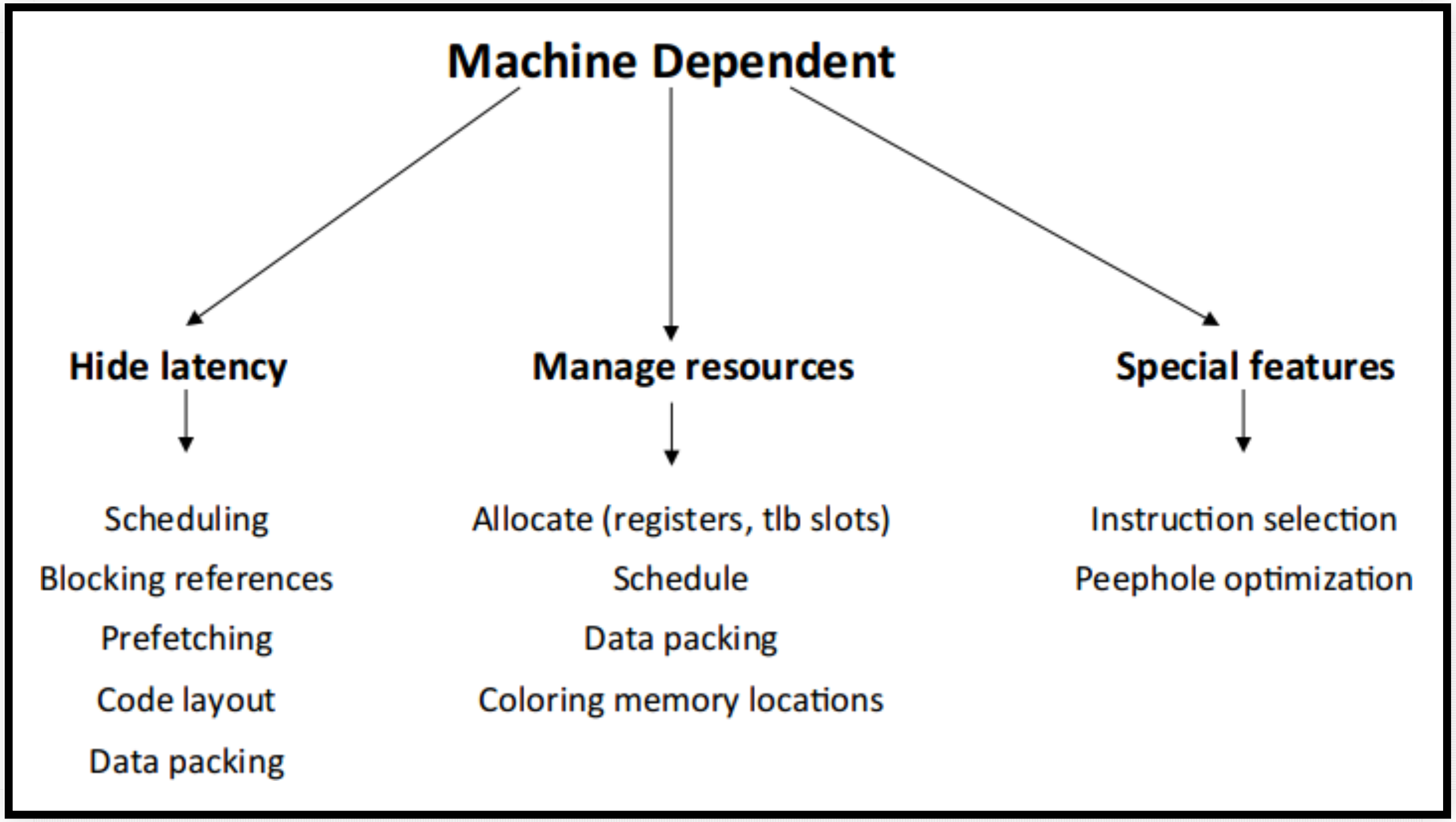
Understanding Compilers

- Compiler has limited understanding of the program's behavior and the environment in which it will be used
  - Most analysis is performed only within procedures
  - Most analysis is based only on static information
- Compilers emphasize **correctness** rather than **performance**
  - Must not cause any change in program behavior under any possible condition
- On well recognized constructs, compilers will usually do better than the developer
- Often, by simply slightly reorganizing existing code, it is possible to improve both code size and speed

Classes of compiler optimizations

- High-level optimizations
  - Loop optimizations
  - Interprocedural optimization
- Global optimizations
- Local optimizations
- Processor dependent optimizations
  - Register allocation
  - Peephole optimizations


Machine Independent
Redundancy
Code motion
Dead code
Specialization
Create opportunities
Redundancy elim.
Partial red. elim.
Consolidation
Loop-invariant c.m.
Consolidation
Global Scheduling
Constant propagation
Dead code elim.
Partial d.c.e.
Constant propagation
Algebraic identities
Replication
Strength Reduction
Method caching
Heap→stack allocation
Tail recursion elimination
Reassociation
Replication
From §6 of Cooper, McKinley, & Torczon
And Chapter 10 of EaC2e


Machine Dependent
Hide latency
Manage resources
Special features
Scheduling
Blocking references
Prefetching
Code layout
Data packing
Allocate (registers, tlb slots)
Schedule
Data packing
Coloring memory locations
Instruction selection
Peephole optimization

GCC -O3 [-mtune=generic -march=x86-64] (ver. 4.7.1)

-falign-labels -fasynchronous-unwind-tables -fauto-inc-dec -fbranch-count-reg -fcaller-saves -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers -fcrossjumping -fcse-follow-jumps -fdebug-types-section -fdefer-pop -fdelete-null-pointer-checks -fdevirtualize -fdwarf2-cfi-asm -fearly-inlining -feliminate-unused-debug-types -fexpensive-optimizations -fforward-propagate -ffunction-cse -fgcse -fgcse-after-reload -fgcse-lm -fgnu-runtime -fguess-branch-probability -fident -fif-conversion -fif-conversion2 -findirect-inlining -finline -finline-atomics -finline-functions -finline-functions-called-once -finline-small-functions -fipa-cp -fipa-cp-clone -fipa-profile -fipa-pure-const -fipa-reference -fipa-sra -fira-share-save-slots -fira-share-spill-slots -fivopts -fkeep-static-consts -fleading-underscore -fmath-errno -fmerge-constants -fmerge-debug-strings -fmove-loop-invariants -fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeephole -fpeephole2 -fpredictive-commoning -fprefetch-loop-arrays -free -freg-struct-return -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-critical-path-heuristic -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-interblock -fsched-last-insn-heuristic -fsched-rank-heuristic -fsched-spec -fsched-spec-insn-heuristic -fsched-stalled-insns-dep -fschedule-insns2 -fshow-column -fshrink-wrap -fsigned-zeros -fsplit-ivs-in-unroller -fsplit-wide-types -fstrict-aliasing -fstrict-overflow -fstrict-volatile-bitfields -fthread-jumps -ftoplevel-reorder -ftrapping-math -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-cselim -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-loop-distribute-patterns -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon -ftree-loop-optimize -ftree-parallelize-loops= -ftree-phiprop -ftree-pre -ftree-pta -ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slp-vectorize -ftree-sra -ftree-switch-conversion -ftree-tail-merge -ftree-ter -ftree-vect-loop-version -ftree-vectorize -ftree-vrp -funit-at-a-time -funswitch-loops -funwind-tables -fvar-tracking -fvar-tracking-assignments -fvect-cost-model -fzero-initialized-in-bss -m128bit-long-double -m64 -m80387 -maccumulate-outgoing-args -malign-stringops -mfancy-math-387 -mfp-ret-in-387 -mglibc -mieee-fp -mmmx -mno-sse4 -mpush-args -mred-zone -msse -msse2 -mtls-direct-seg-refs

Optimization rule #1

The First Rule of Program Optimization:

Don't do it.

The Second Rule (for experts only!):

Don't do it yet.

False optimizations

Don’t second-guess the compiler

There are things that the compiler can easily optimize by itself, for example, writing **a = a >> 1** should never replace **a = a / 2** if your intention is to divide the variable “**a**” by **2** and not shifting it. By doing so you are reducing the readability of your code without really improving anything, modern compilers are perfectly able to do that optimization by themselves.

Outsmarting the Compiler

```c
/* original code */
unsigned foo(unsigned char i)
{ // 3*shl, 3*add
        return i + (i<<8) + (i<<16) + (i<<24);
}
```

```c
/* attempt to improve foo */
unsigned bar(unsigned char i)
{ // 2*shl, 2*add
        unsigned int j = i + (i << 8);
        return j + (j<<16);
}
```

```c
/* "let the compiler do it" */
unsigned baz(unsigned char i)
{ // 1*imul
        return i*0x01010101;
}
```

Compiler Optimization Obstacles

- Pointer aliasing
  - Pointer arithmetic
  - Global variables
  - Dynamic memory allocation
- Control dependencies
  - Indirect addressing
  - Floating point
- Function calls
  - Side effects
  - External functions
  - Virtual functions
- Optimization barriers
  - `volatile` variables
  - `printf`
  - Intrinsic functions
  - inline assembly
- Correctness overhead
  - Sign-extend
  - Stack overusage
  - Exception handling
  - Hardware bugs

C/C++ Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program.

- `auto`
- `register`
- `static`
- `extern`
- `mutable`

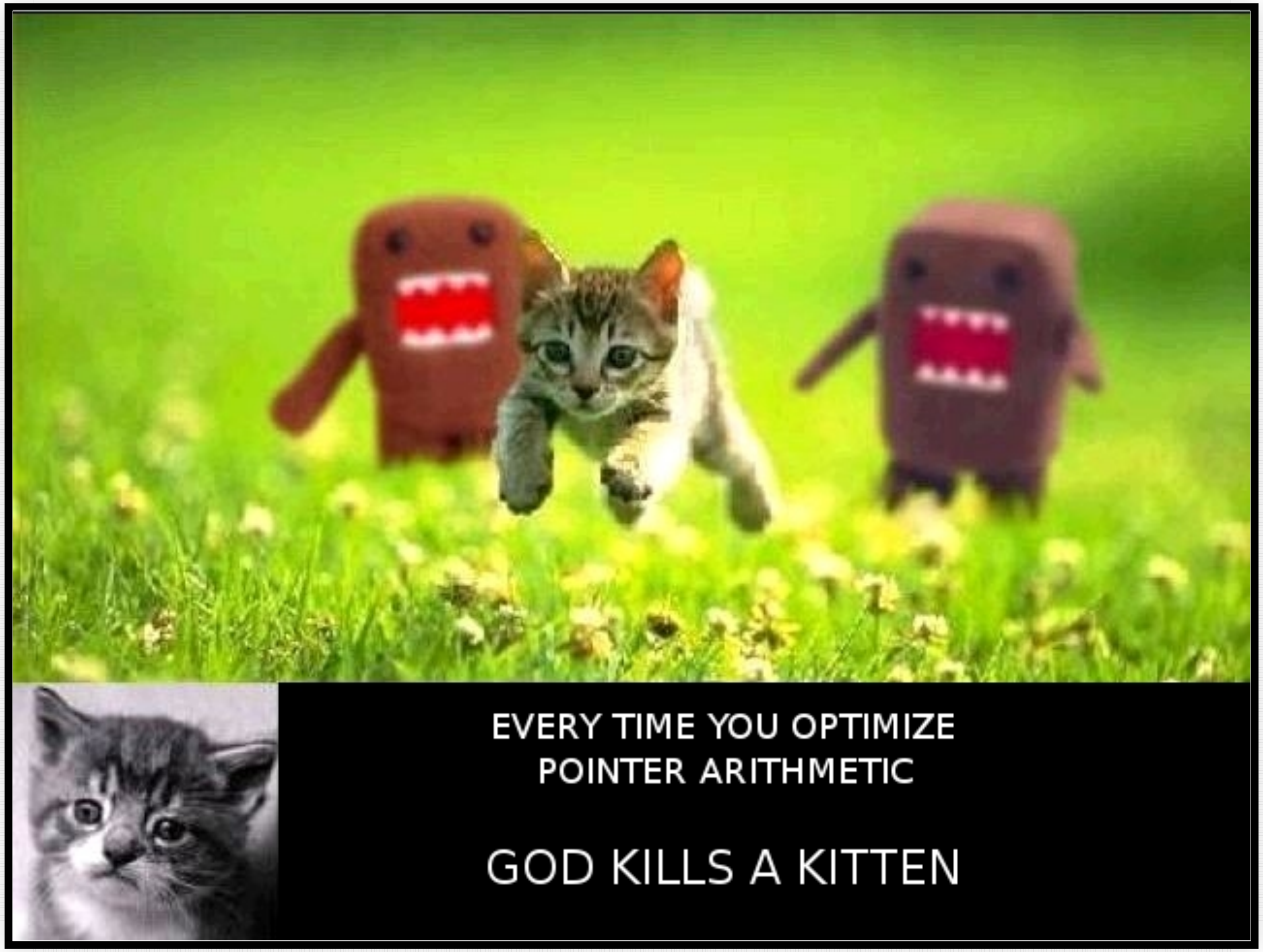
Storage Classes for Optimizators

- `register` - the fastest, the most limited
- `stack` - compiler knows all about them
- `memory` - subject of aliasing
- `extern` - compiler knows nothing about them
- `volatile` - strict barrier

Pointer Aliasing

- C/C++ language has pointers, C/C++ compiler has its worst problem - **pointer aliasing**
- Pointers **alias** when they point to the same address
  - Writing via one pointer will change the value read through another
- The compiler often doesn't know which pointers alias
  - The compiler must assume that any write through a pointer may affect the value read from any another pointer!
  - This can significantly reduce code efficiency

```
/* what if: timers(t1, t1, t1) */
void timers(int *t1, int *t2, int *step)
{
  *t1 += *step;
  *t2 += *step;
}
```


EVERY TIME YOU OPTIMIZE
POINTER ARITHMETIC
GOD KILLS A KITTEN

Taking a Variable's Address

```
int N;

getlimit(&N);
for (i = 0; i < N; i++)
    ...
```

- Taking the address of a variable means that it must live in memory
  - The variable is then subject to pointer aliasing
  - Heavy use of the variable will be costly
- Even if, as in example above, you only take the address once then use it later it's still a memory-bound variable
- Solution: make a second, non-memory-bound, copy of the variable for intensive use
- Global variables are always memory-bound and always subject of aliasing

More about Variables

Single responsibility principle

- Every violation of this principle produce false dependency
- Declare variables closer to usage
- You can't improve stack usage by early declaration of varable

More about Variables

Variable Storage Classes

- Register storage
  - In the best case compiler will be able to assign variable to some register
- Stack storage
  - Planned by compiler at compile time
    - No memory allocation happens
    - If you have more variables than registers then part of them are SPILLED to stack
  - Thread-local storage is a kind of stack storage
- Extern storage
  - For global and static variables

More about Variables

Certain data types are more efficient to use for local variables than others. So

Use machine's NATURAL WORD TYPE

- Shorter types require sign-extend or truncation
- Larger types require greater alignment and occupy more registers
- There is only a minimal difference between the efficiency of 32-bit integers and 64-bit integers, as long as you are not doing divisions
- So use 32-bit integers for intermediate values even if input/output has smaller precision

More about Variables

Signed vs Unsigned integers

In most cases, there is no difference in speed between using signed and unsigned integers.
But there are a few cases where it matters:

- **Division** by a constant: Unsigned is faster than signed when you divide an integer with a constant. This also applies to the modulo operator %
- **Overflow** behaves differently on signed and unsigned variables
- **Conversion** to floating point might be faster with signed than with unsigned integers

More about Variables

Pointers vs References

REFERENCES ARE POINTERS

with syntactic sugar from the point of compiler view

More about Variables

Floating Point

- The main rule: don't mix single and double precision (explicitly add `f` suffix to constants e.g. `1.0f`)
- Single precision float has 2 times smaller memory footprint
- Single precision float better compatible with various coprocessors
- Fixed-point is rarely efficient as replacement of true floating-point calculations

C/C++ Keywords

- **inline** - language overhead
- **const** - means nothing for optimizer (by default)
- **register** - complete placebo
- **restrict** - ~~C only;~~ mostly useless
- **volatile** - optimization barrier

#define for numeric constants

- ```
  #define CONSTANT 23
  ```
- ```
  const int Constant=23;
  ```
- ```
  static const int Constant=23;
  ```
- ```
  enum { constant=23 };
  ```

No memory references and additions in generated code:

```
void foo(void)
{
        a(constant+3);
        a(CONSTANT+4);
        a(Constant+5);
}
```

```
foo:
```

```
        subq $8, %rsp
        movl $26, %edi
        call a
        movl $27, %edi
        call a
        movl $28, %edi
        addq $8, %rsp
        jmp  a
```

macro vs inline

```c
#define abs(x) ((x)>0?(x):-(x))
static long abs2(int x) { return x>=0?x:-x; }

int foo(int a) { return abs(a);  }
int bar(int a) { return abs2(a); }
```

Compiler emits inlined branchless code:

```
foo:
bar:
        mov edx,edi
        sar edx,31     @ int tmp = x >> (sizeof(x) * 8 - 1);
        mov eax,edx
        xor eax,edi    @ return (tmp ^ x) - tmp;
        sub eax,edx
        ret
```

“static” keyword

- Mark constants **static**
  - compiler will not reserve memory slots for static constants
- Mark helper functions **static**
  - **static** gives you internal linkage: compiler will know that it sees all usages of that function
  - compiler will inline static functions used only once
- In C++11 anonymous namespace has exactly the same effect as **static** keyword (it does not in C++98)
- If you have to use global variables then make them **static**

THE END