The LLVM compiler infrastructure

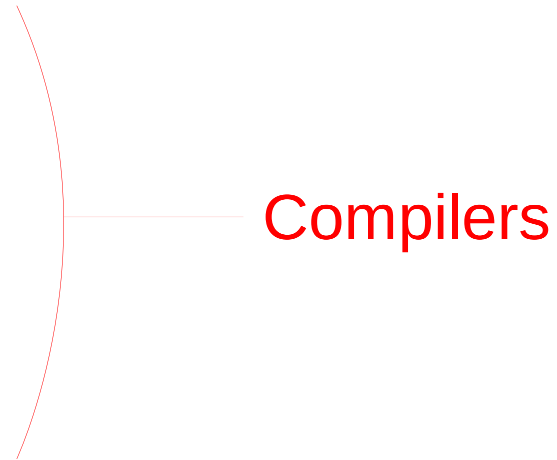
Duncan Sands

DeepBlueCapital / CNRS

Toolchain view

- clang
- gcc + dragonegg
- GHC
- Mono
- Pure
- RenderScript

...



Toolchain view

- clang
- gcc + dragonegg
- GHC
- Mono
- Pure
- RenderScript

...

Hosted by llvm.org

Toolchain view

- clang
- gcc + dragonegg

...

- lldb

Compilers

Debugger

Toolchain view

- clang
- gcc + dragonegg

...

- lldb

- llvm-mc

Toolchain view

- clang
- gcc + dragonegg

...

Compilers

- lldb ← Debugger

- llvm-mc ← Assembler / disassembler

- ? ← Linker (work in progress)

Toolchain view

- clang
- gcc + dragonegg

...

- lldb ← Debugger
- llvm-mc ← Assembler / disassembler
- ? ← Linker (work in progress)

Compilers

BSD style free software license

Toolchain view

- clang
- gcc + dragonegg

...

Compilers

- lldb ← Debugger
- llvm-mc ← Assembler / disassembler
- ? ← Linker (work in progress)
- static analyzers, IR manipulation and visualization tools, link time optimizers, JIT, C++ standard library, ...

Clang

GCC compatible C, C++, Obj-C, Obj-C++, OpenCL, ... compiler

```
$ gcc -c -O2 -mtune=native -fomit-frame-pointer
-msse2 -ffast-math oggenc.c
```

Drop-in replacement

```
$ clang -c -O2 -mtune=native -fomit-frame-pointer
-msse2 -ffast-math oggenc.c
```

Clang advantages

- Excellent error messages and warnings

```
$ cat t.c
struct S { int a; }
void foo(struct S *s);
$ gcc -c t.c
t.c:2:1: error: expected ‘;’, identifier or
‘(’ before ‘void’
$ clang -c t.c
t.c:1:20: error: expected ';' after struct
struct S { int a; }
                   ^
                   ;
1 error generated.
```

Missing semicolon

Clang advantages

- Excellent error messages and warnings

```
$ cat t.c
struct S { int a; }
void foo(struct S *s);
$ gcc -c t.c
t.c:2:1: error: expected ‘;’, identifier or
‘(’ before ‘void’
$ clang -c t.c
t.c:1:20: error: expected ';' after struct
struct S { int a; }
                   ^
                   ;
1 error generated.
```

Missing semicolon

Clang advantages

- Excellent error messages and warnings

```
$ cat t.c
struct S { int a; }
void foo(struct S *s);
$ gcc -c t.c
t.c:2:1: error: expected ‘;’, identifier or
‘(’ before ‘void’
$ clang -c t.c
t.c:1:20: error: expected ';' after struct
struct S { int a; }
                   ^
                   ;
1 error generated.
```

Missing semicolon

Clang advantages

- Excellent error messages and warnings
- Compiles code faster than GCC
- Strict standards conformance
- BSD license

Clang disadvantages

- Generated code usually slower than GCC's
- Strict standards conformance
- License not copyleft

Dragonegg

Plugin for GCC, supports same languages as GCC.

```
$ gcc -S -O2 fatigue.f90 -o -
        .file   "fatigue.f90"
        .section
.rodata.str1.1,"aMS",@progbits,1
...
```

Dragonegg

Plugin for GCC, supports same languages as GCC.

```
$ gcc -S -O2 fatigue.f90 -o -
        .file   "fatigue.f90"
        .section
.rodata.str1.1,"aMS",@progbits,1
...

$ gcc -S -O2 fatigue.f90 -o - -fplugin=./dragonegg.so
        .file   "fatigue.f90"
        .ident  "GCC: (Debian 4.6.2-4) 4.6.2 LLVM:
145237M"
        .text
        .globl  __free_input_MOD_check_number
...
```

Dragonegg

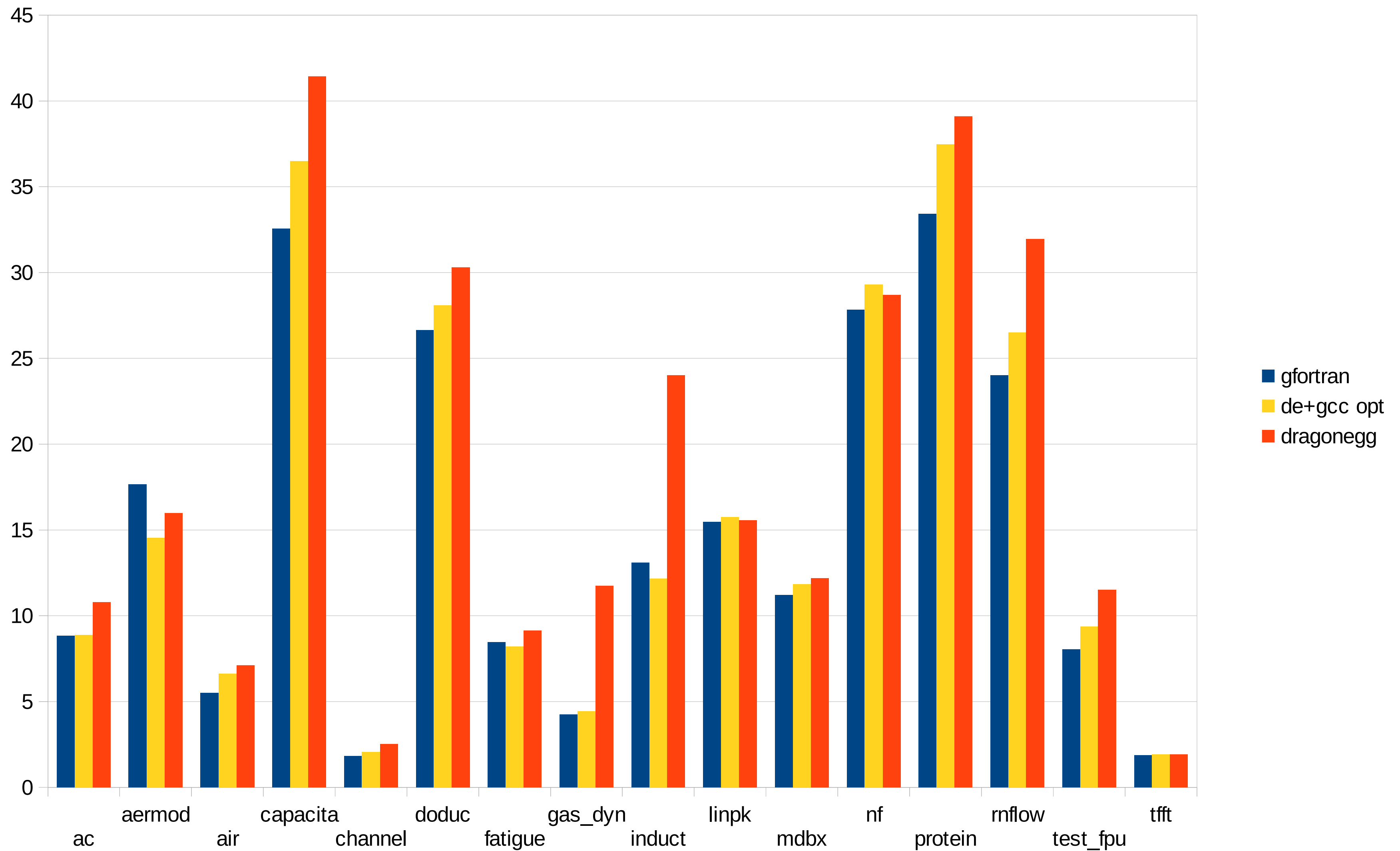
- Replaces GCC's optimizers with LLVM's (optional)
- Uses LLVM to do code generation
- Can produce LLVM IR instead of target assembler

Dragonegg

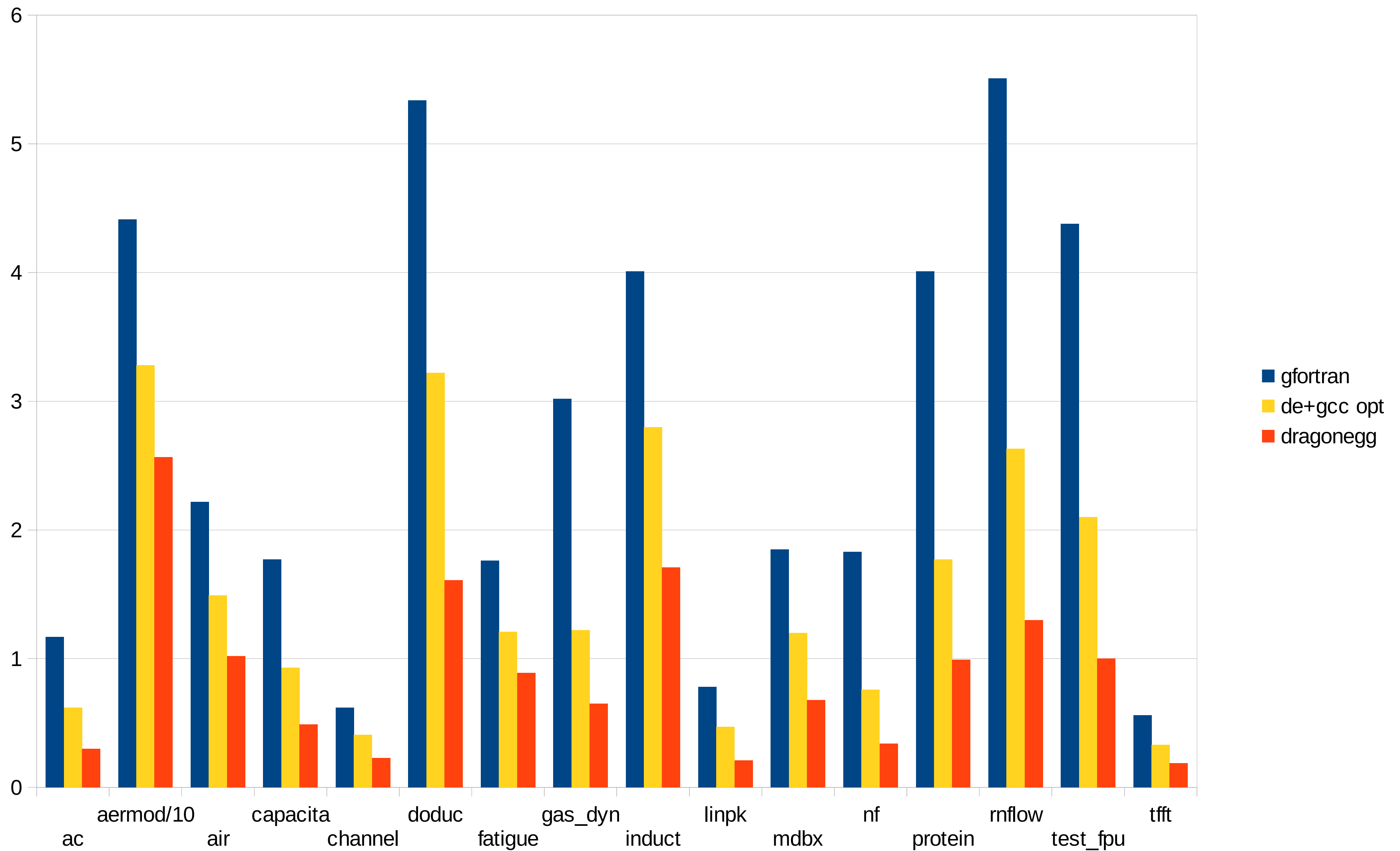
- Replaces GCC's optimizers with LLVM's (optional)
- Uses LLVM to do code generation
- Can produce LLVM IR instead of target assembler

```
$ gcc -S -O2 fatigue.f90 -o - -fplugin=./dragonegg.so
-fplugin-arg-dragonegg-emit-ir
; ModuleID = 'fatigue.f90'
target datalayout = "e-p:64:64:64-S128-i1:8:8-i8:8:8-
i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-
v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-
f128:128:128-n8:16:32:64"
target triple = "x86_64--linux-gnu"
...
```

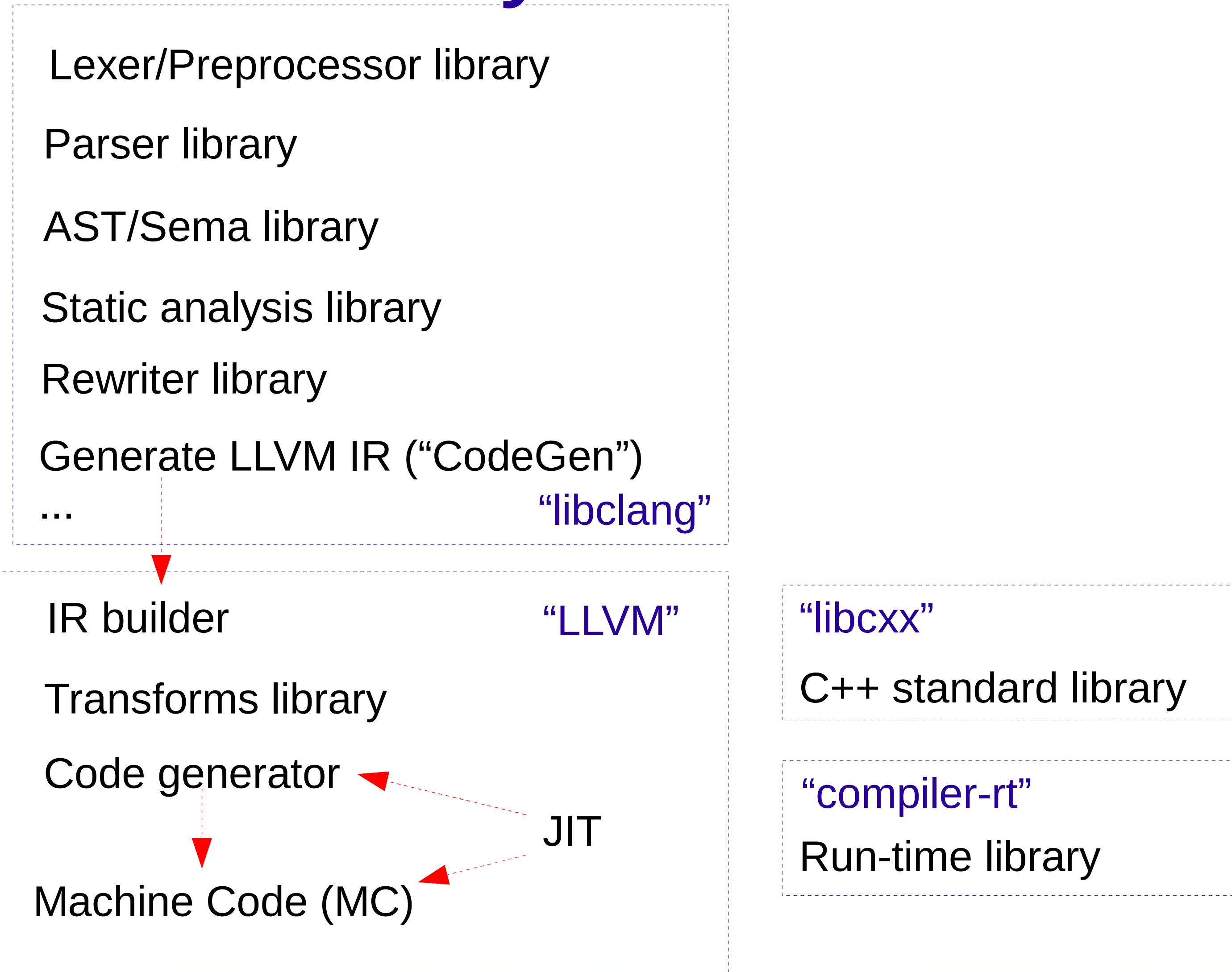
Execution time



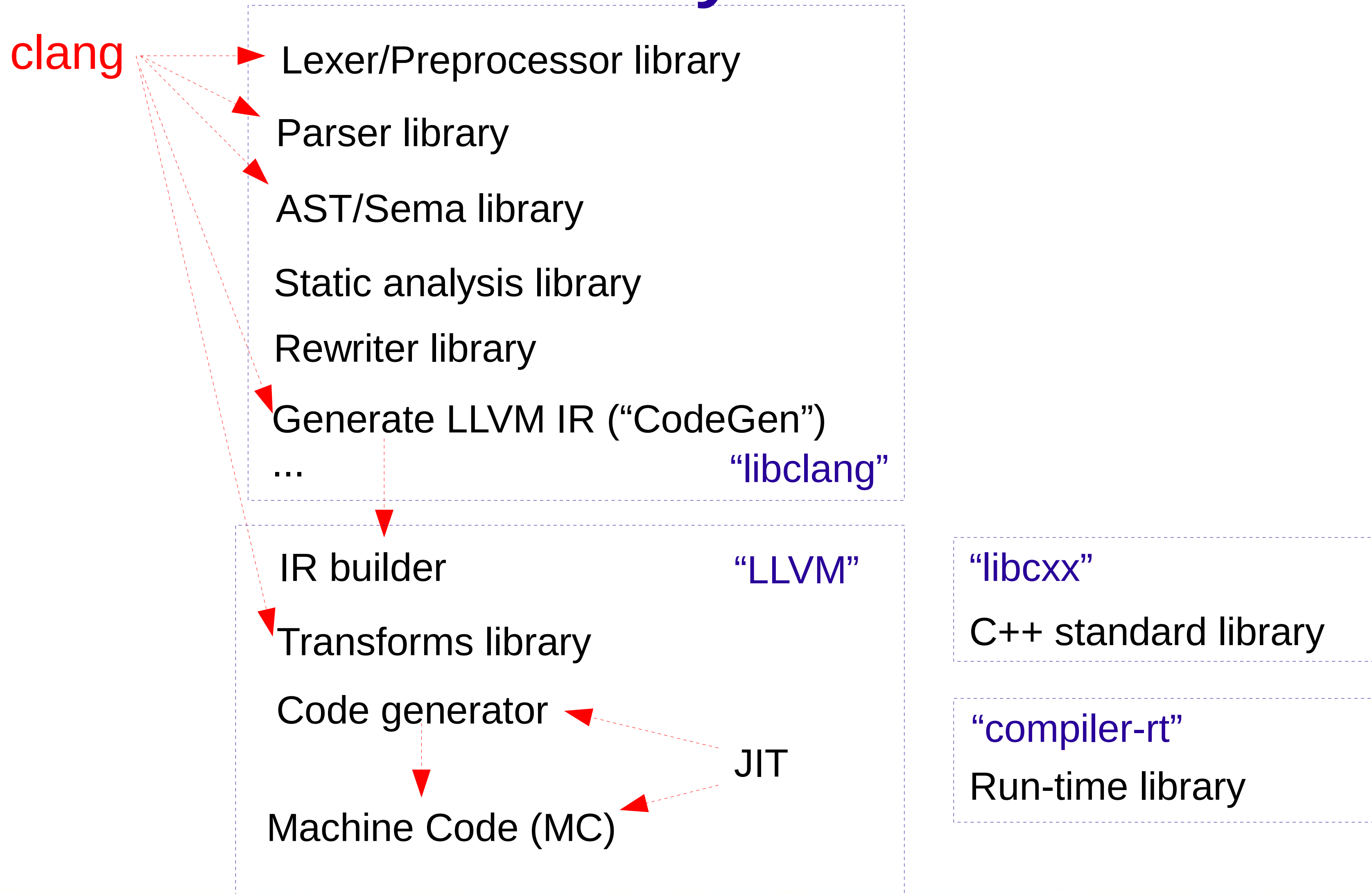
Compile time



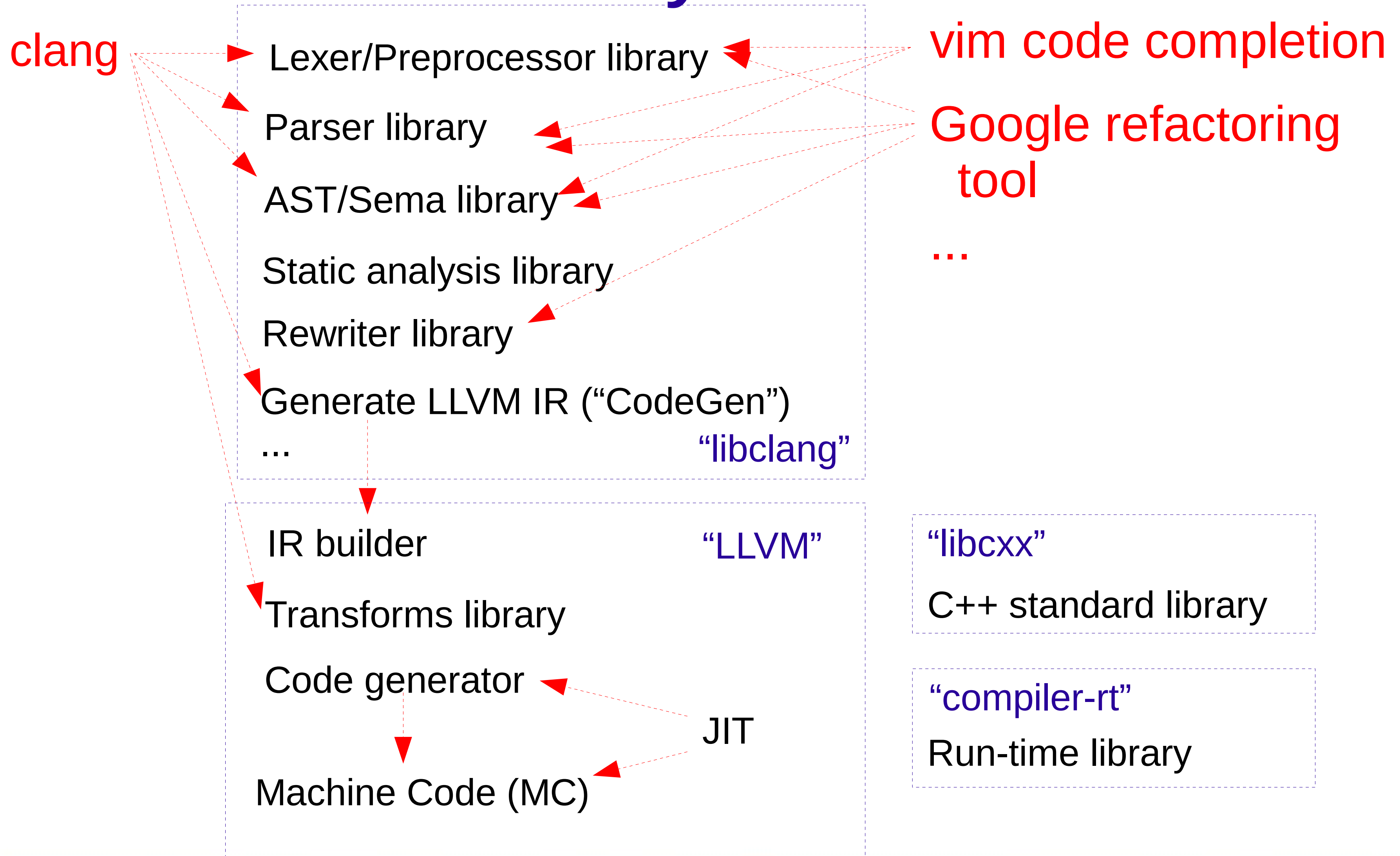
Library view



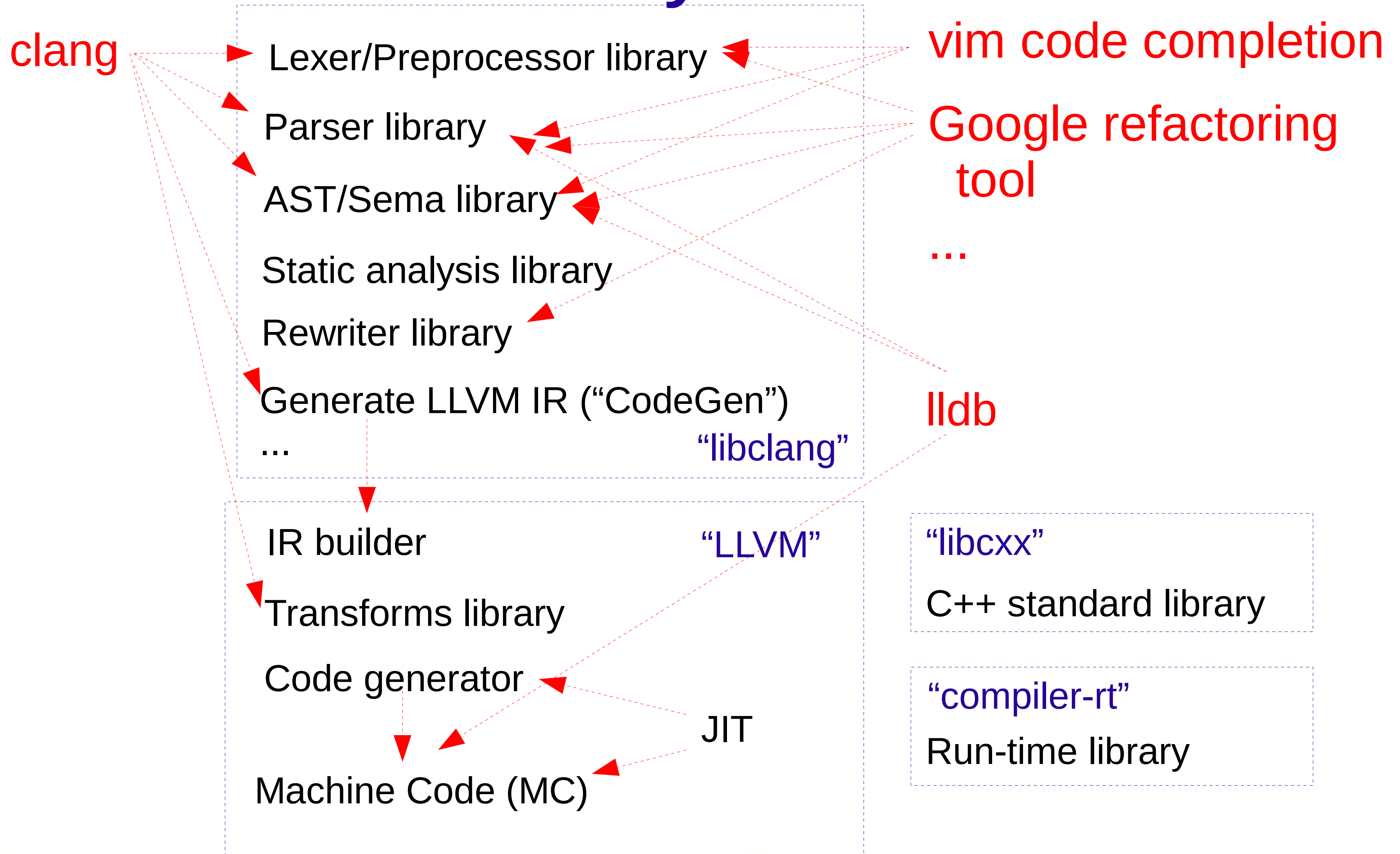
Library view



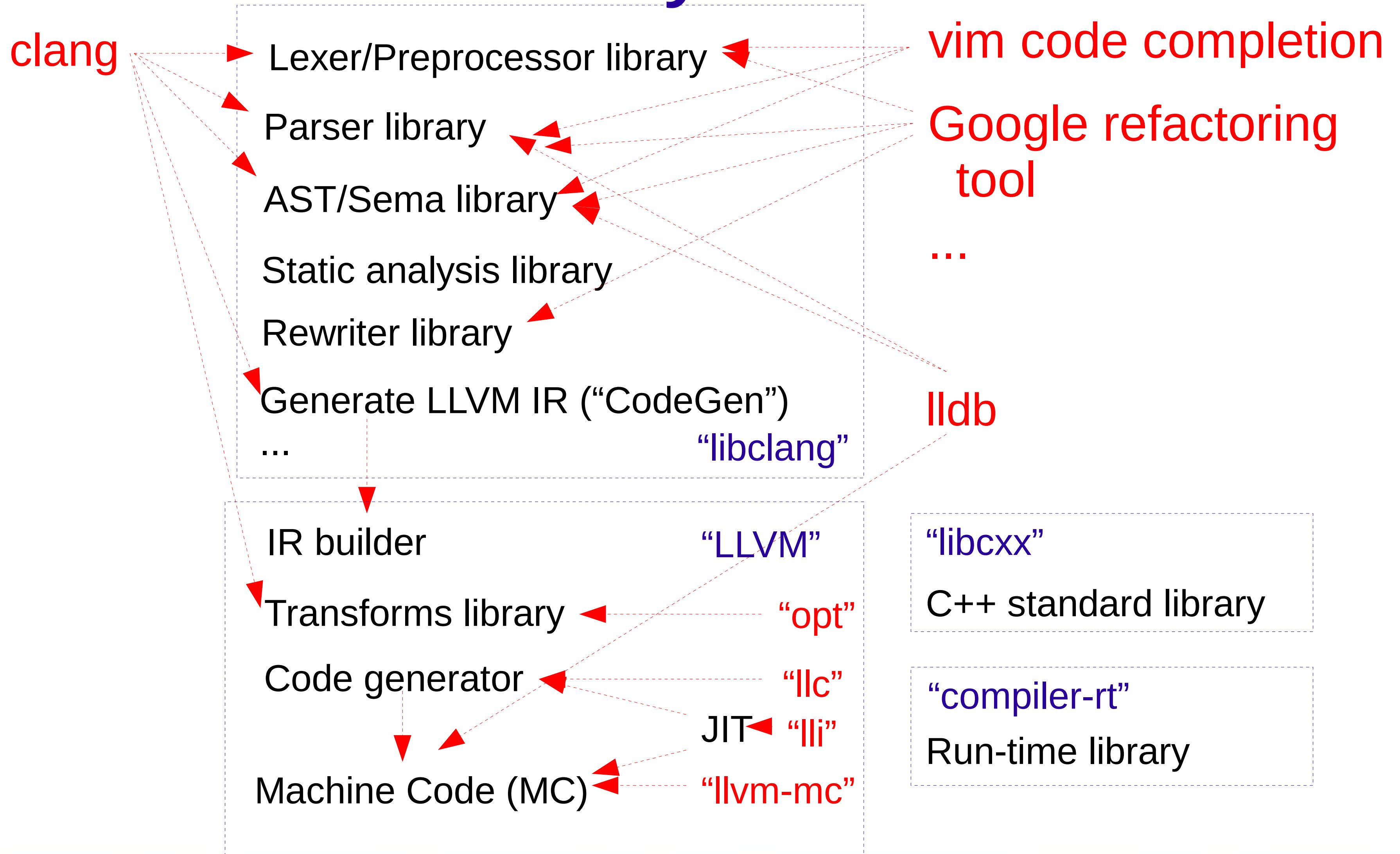
Library view



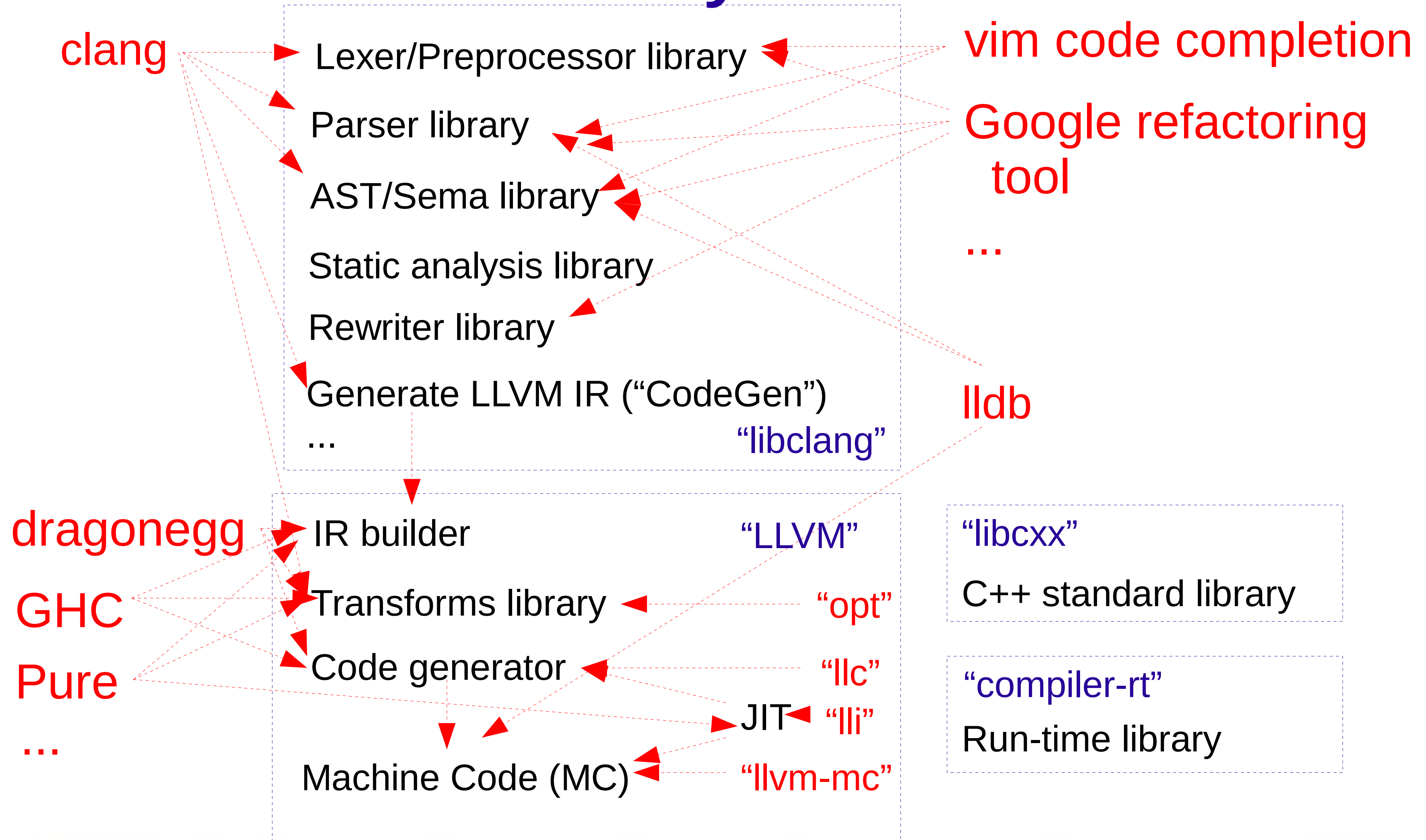
Library view



Library view



Library view



LLVM IR

IR = intermediate representation
  - how code is represented internally by LLVM

Three forms:

- In memory (how front-ends build it)
- On disk bitcode representation
- Human readable assembly

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

LLVM IR

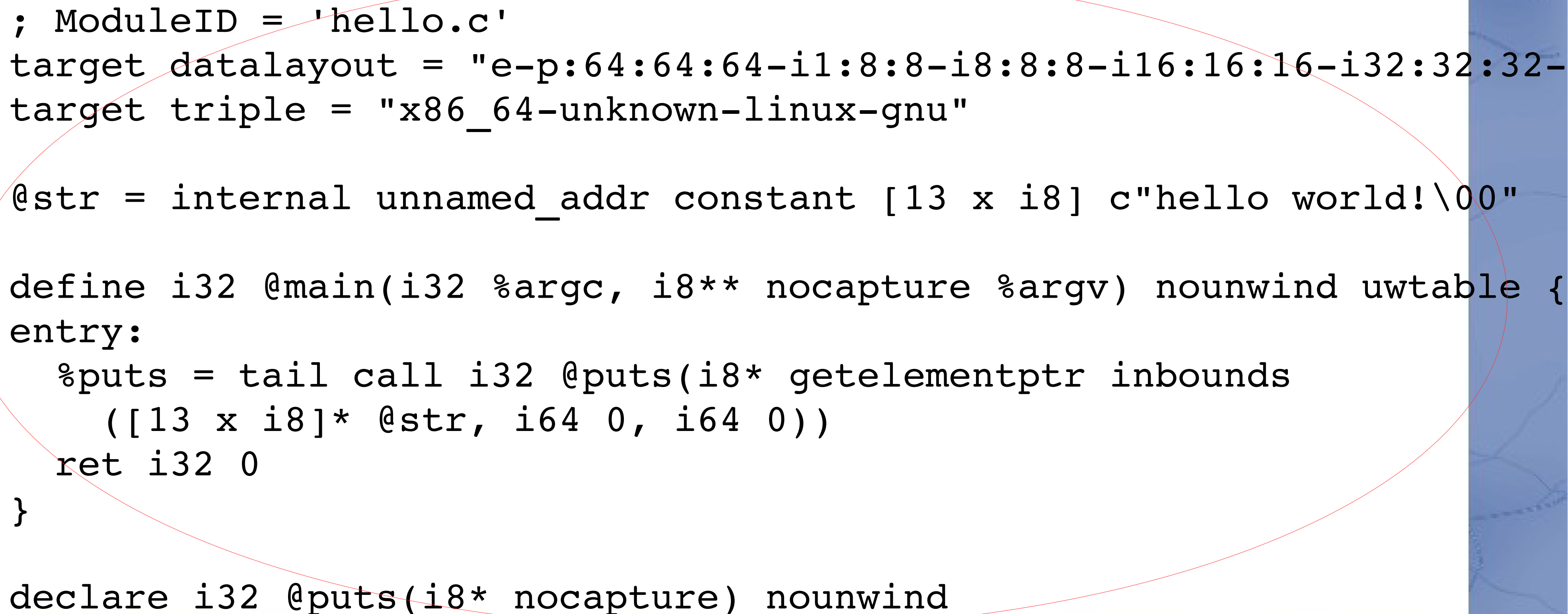
```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

Human readable IR



```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

Semicolon starts comment

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Comment

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

Little endian

Information about the target

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"
```

Pointers are 64 bits wide, 64 bits aligned

```
@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

Global variable definition

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```c
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

It's a constant!

Variable name

Global variable definition

```llvm
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"
```

Type

```llvm
@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}
```

Initial value

Not visible outside this module

```llvm
declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"
```

Function definition

```
@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Function declaration

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}
```

Function return type

```
declare i32 @puts(i8* nocapture) nounwind
```

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Parameter names

Parameter attributes

Function attributes

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Basic block name

Basic block

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Call instruction

Return instruction

LLVM IR

```
int main(int argc, char* argv[]) {
  printf("hello world!\n");
  return 0;
}
```

```
; ModuleID = 'hello.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
target triple = "x86_64-unknown-linux-gnu"

@str = internal unnamed_addr constant [13 x i8] c"hello world!\00"

define i32 @main(i32 %argc, i8** nocapture %argv) nounwind uwtable {
entry:
  %puts = tail call i32 @puts(i8* getelementptr inbounds
    ([13 x i8]* @str, i64 0, i64 0))
  ret i32 0
}

declare i32 @puts(i8* nocapture) nounwind
```

Constant expression

LLVM IR

- LLVM IR is always in SSA form

```
...
"<bb 428>":             ; preds = %"<bb 427>", %"<bb 426>"
  %n.465_1508 = phi i32 [ %n.465_1508.pre, %"<bb 427>" ],
                        [ %n.465_1484, %"<bb 426>" ]
  %D.2937_1509 = icmp eq i32 %n.465_1508, %D.2932_1481
  %n.466_1511 = add nsw i32 %n.465_1508, 1
  store i32 %n.466_1511, i32* %memtmp45, align 4
  br i1 %D.2937_1509, label %"<bb 431>", label %"<bb 426>"

"<bb 429>":             ; preds = %"<bb 413>", %"<bb 424>"
...
```

Basic block

LLVM IR

- LLVM IR is always in SSA form

Phi node (PHINode instruction)

```
...
"<bb 428>":                ; preds = %"<bb 427>", %"<bb 426>"
  %n.465_1508 = phi i32 [ %n.465_1508.pre, %"<bb 427>" ],
                      [ %n.465_1484, %"<bb 426>" ]
  %D.2937_1509 = icmp eq i32 %n.465_1508, %D.2932_1481
  %n.466_1511 = add nsw i32 %n.465_1508, 1
  store i32 %n.466_1511, i32* %memtmp45, align 4
  br i1 %D.2937_1509, label %"<bb 431>", label %"<bb 426>"

"<bb 429>":                ; preds = %"<bb 413>", %"<bb 424>"
...
```

Basic block

LLVM IR

- LLVM IR is always in SSA form

```
...
"<bb 428>":             ; preds = %"<bb 427>", %"<bb 426>"
  %n.465_1508 = phi i32 [ %n.465_1508.pre, %"<bb 427>" ],
                        [ %n.465_1484, %"<bb 426>" ]
  %D.2937_1509 = icmp eq i32 %n.465_1508, %D.2932_1481
  %n.466_1511 = add nsw i32 %n.465_1508, 1
  store i32 %n.466_1511, i32* %memtmp45, align 4
  br i1 %D.2937_1509, label %"<bb 431>", label %"<bb 426>"

"<bb 429>":             ; preds = %"<bb 413>", %"<bb 424>"
...
```

LLVM IR

- LLVM IR is always in SSA form
- Def-use chains are built into the IR

```
...
"<bb 428>":             ; preds = %"<bb 427>", %"<bb 426>"
  %n.465_1508 = phi i32 [ %n.465_1508.pre, %"<bb 427>" ],
                        [ %n.465_1484, %"<bb 426>" ]
  %D.2937_1509 = icmp eq i32 %n.465_1508, %D.2932_1481
  %n.466_1511 = add nsw i32 %n.465_1508, 1
  store i32 %n.466_1511, i32* %memtmp45, align 4
  br i1 %D.2937_1509, label %"<bb 431>", label %"<bb 426>"

"<bb 429>":             ; preds = %"<bb 413>", %"<bb 424>"
...
```

LLVM IR

- LLVM IR is always in SSA form
- Def-use chains are built into the IR
- The control flow graph is built into the IR

Control flow into basic block (predecessors)

```
...
"<bb 428>":            ; preds = %"<bb 427>", %"<bb 426>"
  %n.465_1508 = phi i32 [ %n.465_1508.pre, %"<bb 427>" ],
                        [ %n.465_1484, %"<bb 426>" ]
  %D.2937_1509 = icmp eq i32 %n.465_1508, %D.2932_1481
  %n.466_1511 = add nsw i32 %n.465_1508, 1
  store i32 %n.466_1511, i32* %memtmp45, align 4
  br i1 %D.2937_1509, label %"<bb 431>", label %"<bb 426>"

"<bb 429>":            ; preds = %"<bb 413>", %"<bb 424>"
...
```

Basic block

Explicit control flow out of basic block

Super optimization

- Optimization $\rightarrow$ Improve code

Super optimization

- Optimization $\rightarrow$ Improve code
- Super-optimization $\rightarrow$ Obtain perfect code

Super optimization

- Optimization $\rightarrow$ Improve code
- Super-optimization $\rightarrow$ Obtain perfect code

Super-optimization $\rightarrow$ automatically find code improvements

Super optimization

- ~~Optimization $\rightarrow$ Improve code~~
- ~~Super-optimization $\rightarrow$ Obtain perfect code~~

Super-optimization $\rightarrow$ automatically find code improvements

Idea from LLVM OpenProjects web-page
(suggested by John Regehr)

Goal

Automatically find simplifications missed by the LLVM optimizers

- And have a human implement them in LLVM

Goal

Automatically find simplifications missed by the LLVM optimizers

- And have a human implement them in LLVM

Non goal

~~Directly optimize programs~~

→ It doesn't matter if the simplifications found are sometimes wrong

Process

- Compile program to bitcode

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM

Repeat

Process

- Compile program to bitcode
- Run optimizers on bitcode
- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM
- Profit!

Repeat

Process

- Compile program to bitcode
- Run optimizers on bitcode

- Harvest interesting expressions
- Analyse them for missing simplifications
- Implement the simplifications in LLVM
- Profit!

Inspired by “Automatic Generation of Peephole Superoptimizers” by Bansal & Aiken (Computer Systems Lab, Stanford)

Harvesting

```
$ opt -load=./harvest.so -std-compile-opts -harvest \
    -disable-output bzip2.bc
@07:@09
07:@07:@3c:12:@3c:@06:@07:24:28:20:@29
...
```

Normalized & encoded form allows textual comparisons:

```
$ opt -load=./harvest.so -std-compile-opts -harvest \
  -disable-output bzip2.bc | sort | uniq -c | sort -r -n
    265 @00:07:@2b
    178 @01:07:@0f
    120 @00:@07:@2b
    ...
```

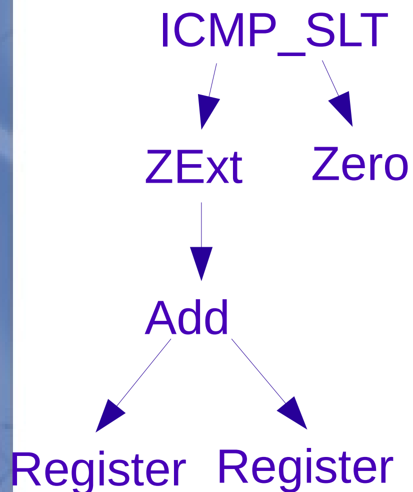
Ordered by frequency of occurrence

Harvesting

Most common expressions in unoptimized bitcode from the LLVM testsuite:

07:0a	→	sext X	sext = sign-extend
00:07:2c	→	X != 0	
07:09	→	zext X	zext = zero-extend
05:07:0f	→	X +nsw -1	+nsw = add with no-signed wrap
00:07:2b	→	X == 0	
07:07:13	→	X -nsw Y	-nsw = sub with no-signed wrap
07:07:32	→	X >=s Y	>=s = signed greater than or equal
01:07:0f	→	X +nsw 1	
06:07:0a:16	→	(sext X) * power-of-2	power-of-2 = constant that is a power of two

Expressions



- Directed acyclic graph - no loops!
- Integer operations only - no floating point!
- No memory operations (load/store)!
- No types!
- Limited set of constants (eg: Zero, One, SignBit)

Most integer operations supported (eg: ctlz, overflow intrinsics).
Doesn't support byteswap (because of lack of types).

Analysing expressions

Four modes:

- Constant folding
- Reduce to sub-expression
- Unused variables
- Rule reduction

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x /s y) * y) /s y → x /s y

- Unused variables

  x - (x + y) → 0 - y

  Result does not depend on x
  Can replace x with (eg) 0

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 → 0 (i.e. false)

- Reduce to sub-expression

  ((x /s y) * y) /s y → x /s y

- Unused variables

  x - (x + y) → 0 - y

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

Analysing expressions

Four modes:

- Constant folding

  zext x <s 0 $\rightarrow$ 0 (i.e. false)

  Fast! Always a win!

- Reduce to sub-expression

  ((x /s y) * y) /s y $\rightarrow$ x /s y

  Fast! Often a win!

- Unused variables

  x - (x + y) $\rightarrow$ 0 - y

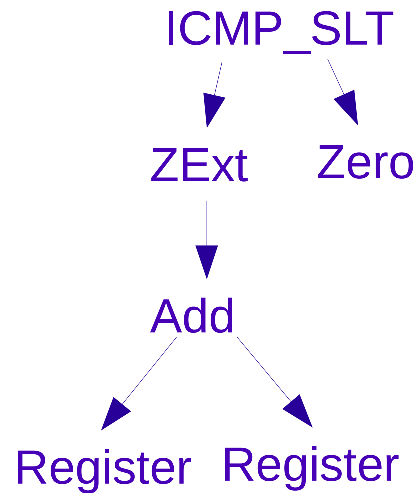
  Fast! Sometimes a win!

- Rule reduction

  Repeatedly apply rules from a list.
  Search minimum of cost function.

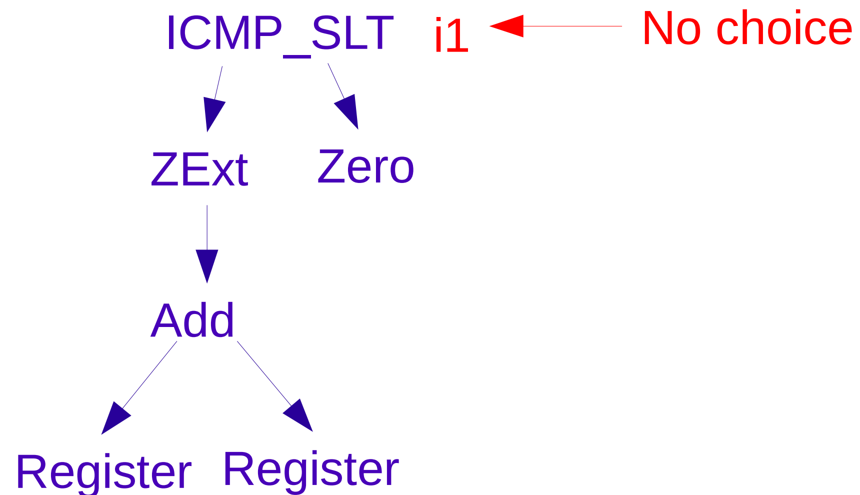
  Slow! Work in progress!

Constant folding



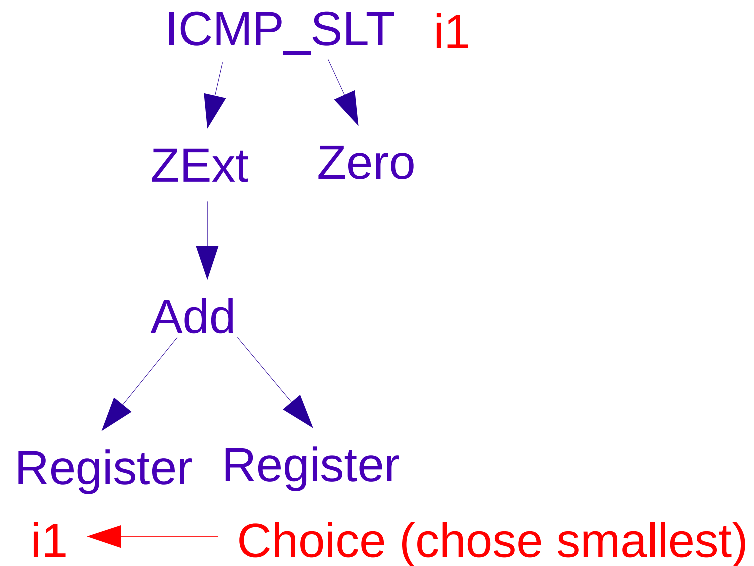
- Assign types to nodes

Constant folding



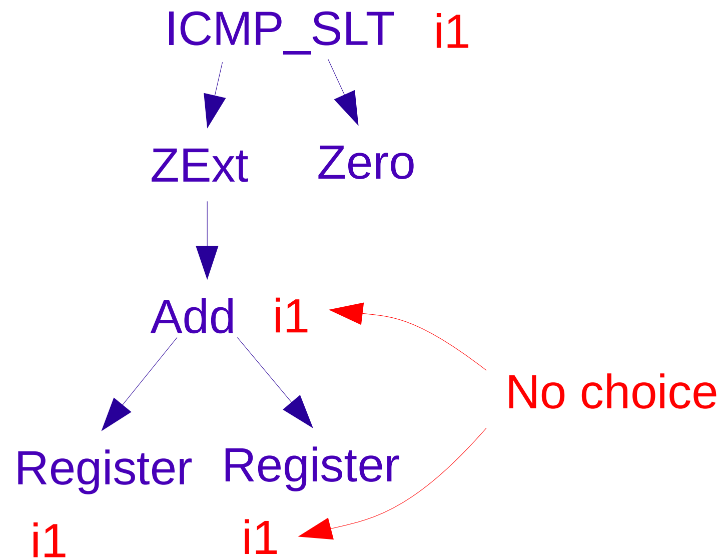
- Assign types to nodes

Constant folding



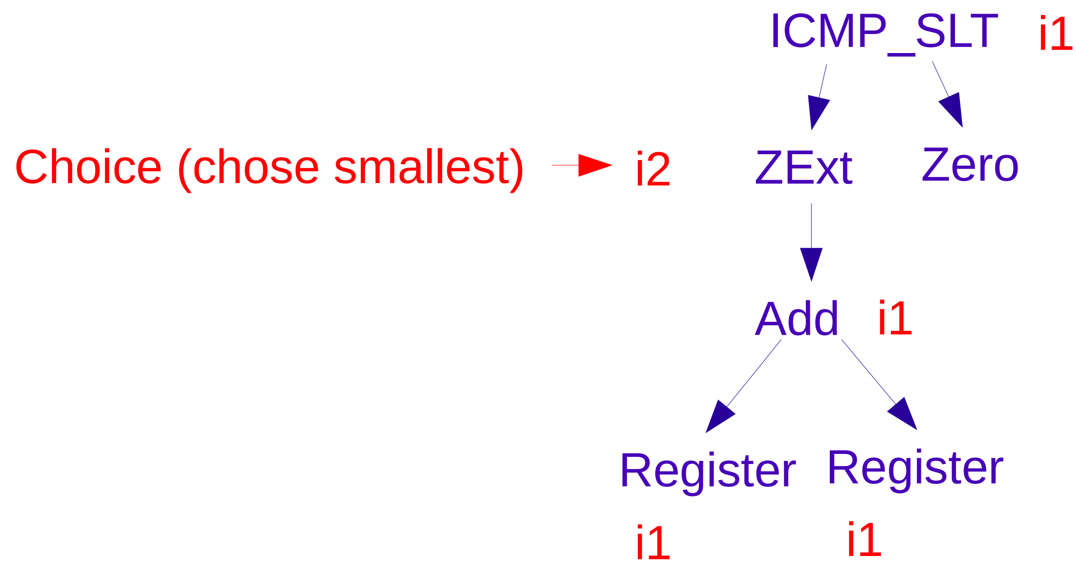
- Assign types to nodes

Constant folding



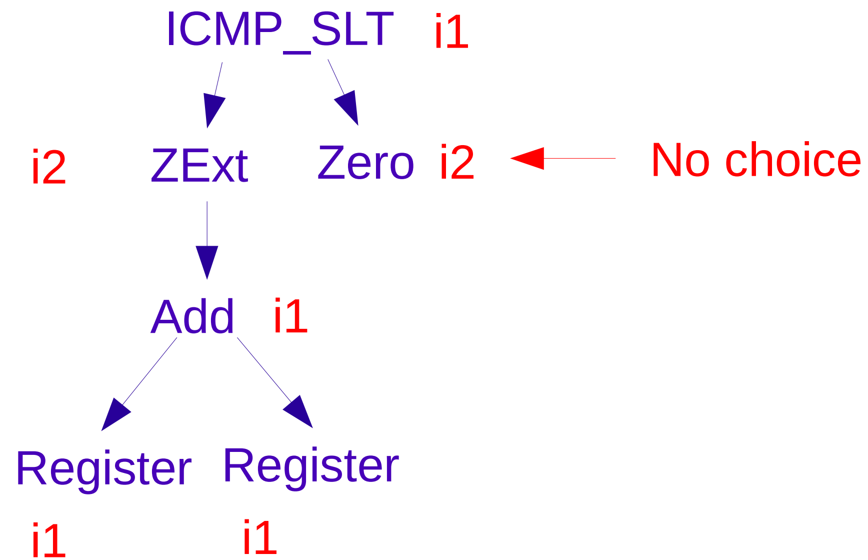
- Assign types to nodes

Constant folding



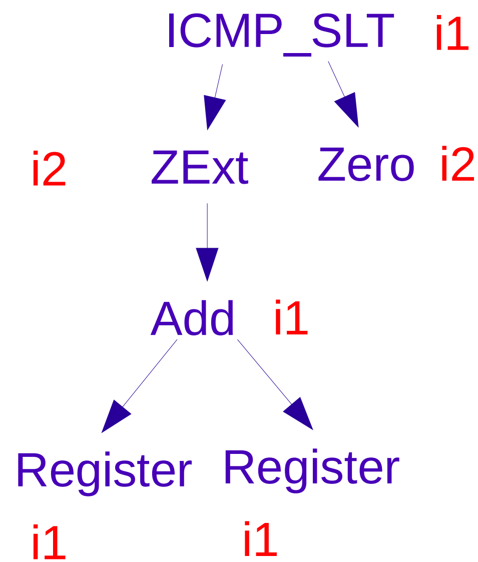
- Assign types to nodes

Constant folding



- Assign types to nodes

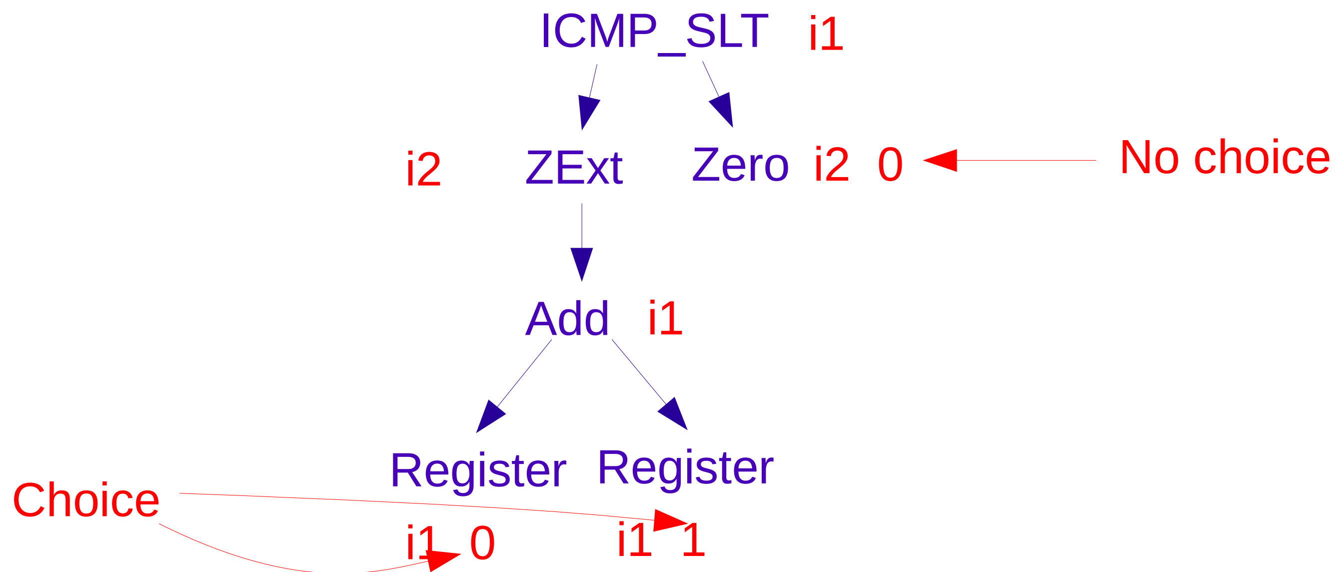
Constant folding



- Assign types to nodes

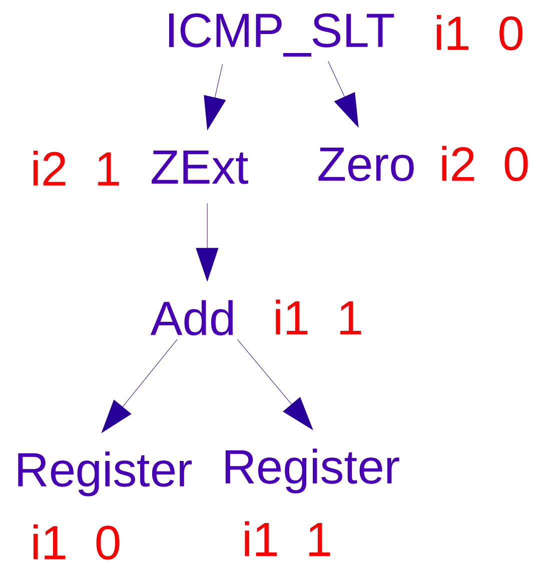
  Strategies: (1) Random choice; (2) All small types.

Constant folding



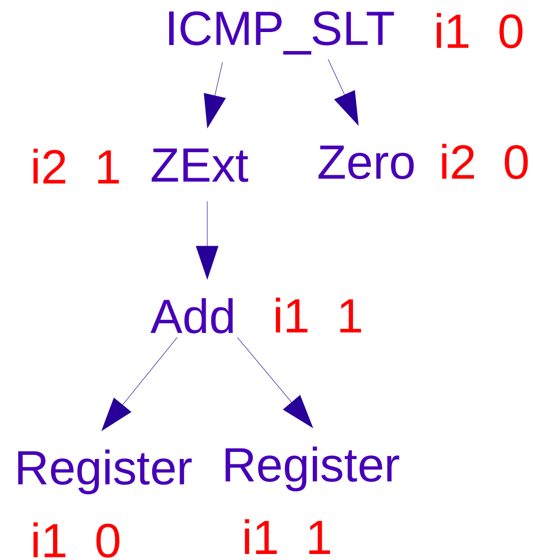
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up

Constant folding



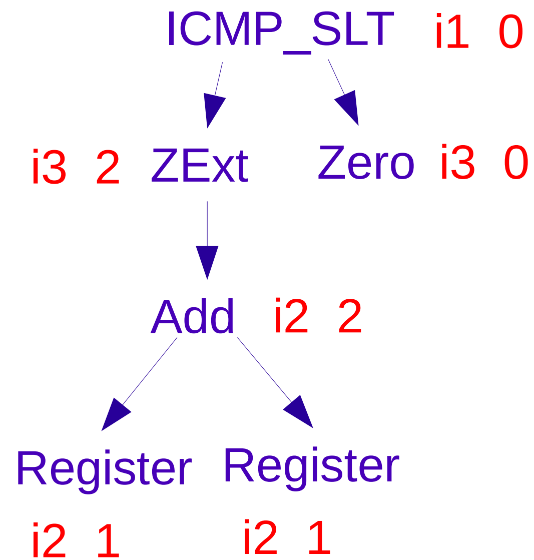
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up

Constant folding



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.

Constant folding



Repeat many times

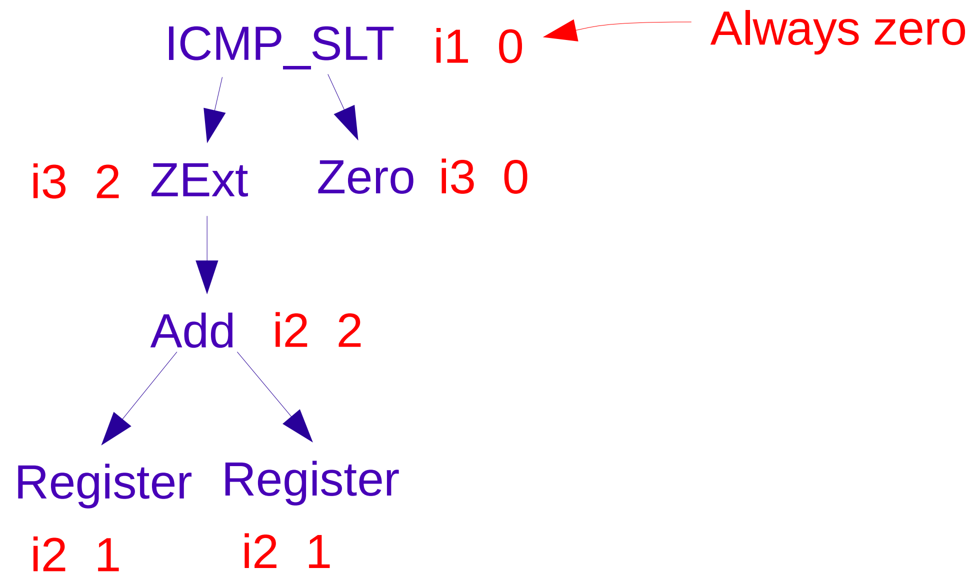
- Assign types to nodes

  Strategies: (1) Random choice; (2) All small types.

- Assign values to terminal nodes & propagate up

  Strategies: (1) Random inputs; (2) Every possible input.

Constant folding



- Assign types to nodes

  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up

  Strategies: (1) Random inputs; (2) Every possible input.
- Result at the root always the same

  → found a constant fold

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false

Examples

Constant folds found in "fully optimized" code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false

  Implemented as: "non-negative-number + power-of-two != 0"

Examples

Constant folds found in "fully optimized" code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true

  “max(X, Y) >= X”. Implemented several max/min folds.

Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true
- X rem (Y ? X : 1) $\rightarrow$ 0
- (Y /u X) >u Y $\rightarrow$ false

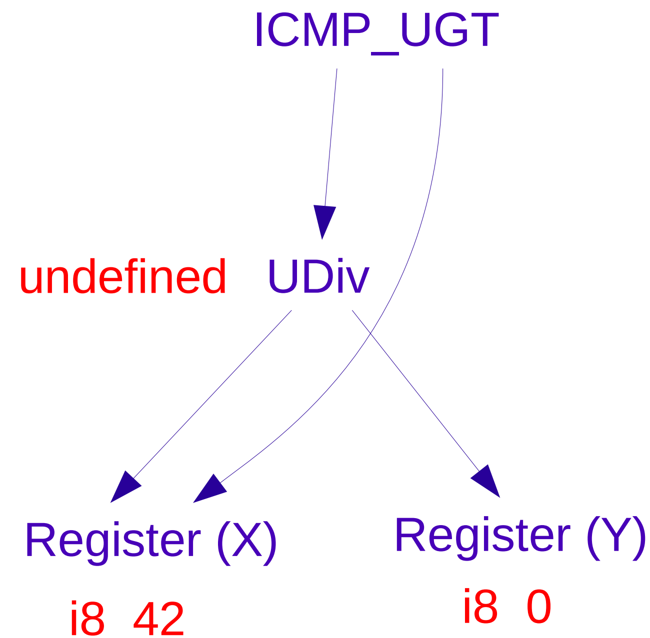
Examples

Constant folds found in “fully optimized” code:

- (((X + Y) >>L power-of-two) & Z) + power-of-two == 0 $\rightarrow$ false
- ((X >s Y) ? X : Y) >=s X $\rightarrow$ true
- X rem (Y ? X : 1) $\rightarrow$ 0
- (Y /u X) >u Y $\rightarrow$ false

Undefined behaviour

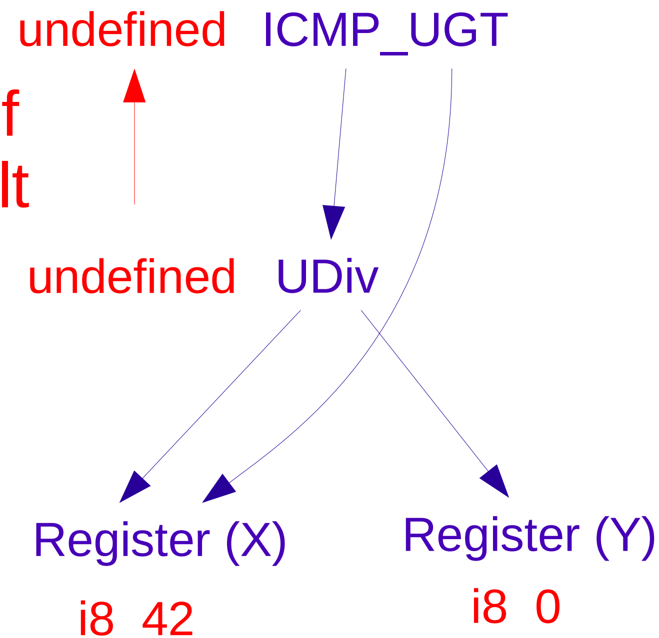
(X /u Y) >u X $\rightarrow$ false



Undefined behaviour

(X /u Y) >u X $\rightarrow$ false

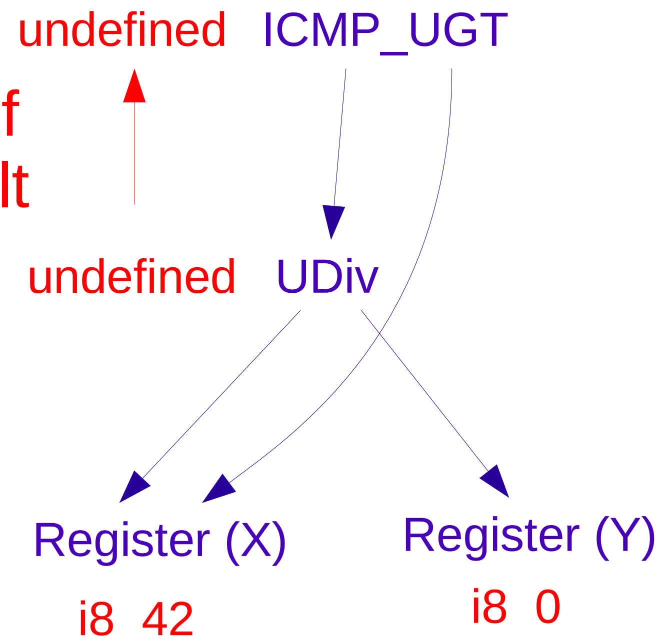
Any operation with an undef operand gets an undef result



Undefined behaviour

(X /u Y) >u X $\rightarrow$ false

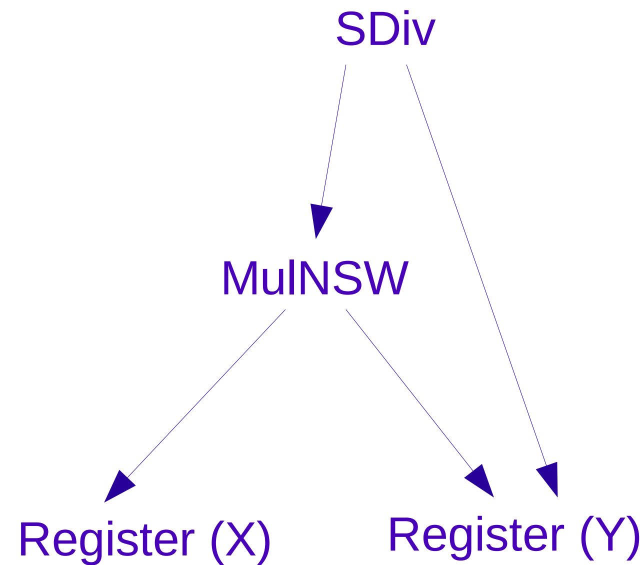
Any operation with an undef operand gets an undef result



- Avoids false negatives
- May result in subtle false positives

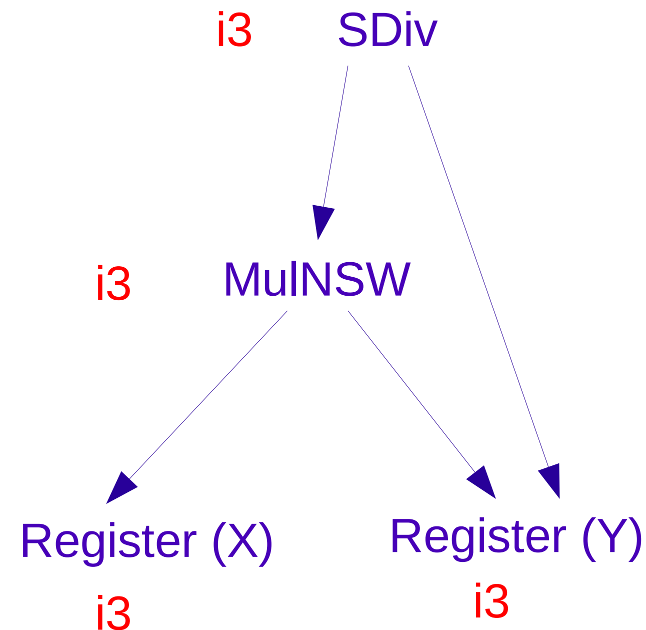
Reduce to subexpression

(X *nsw Y) /s Y $\rightarrow$ X



Reduce to subexpression

(X *nsw Y) /s Y $\rightarrow$ X

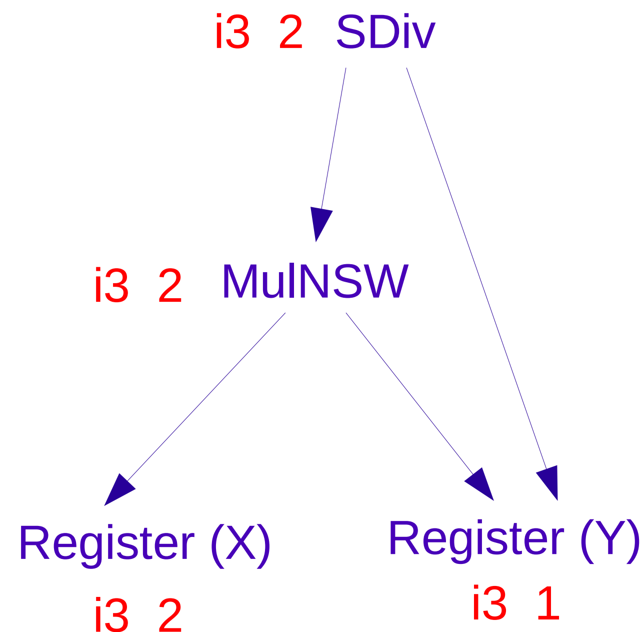


- Assign types to nodes

  Strategies: (1) Random choice; (2) All small types.

Reduce to subexpression

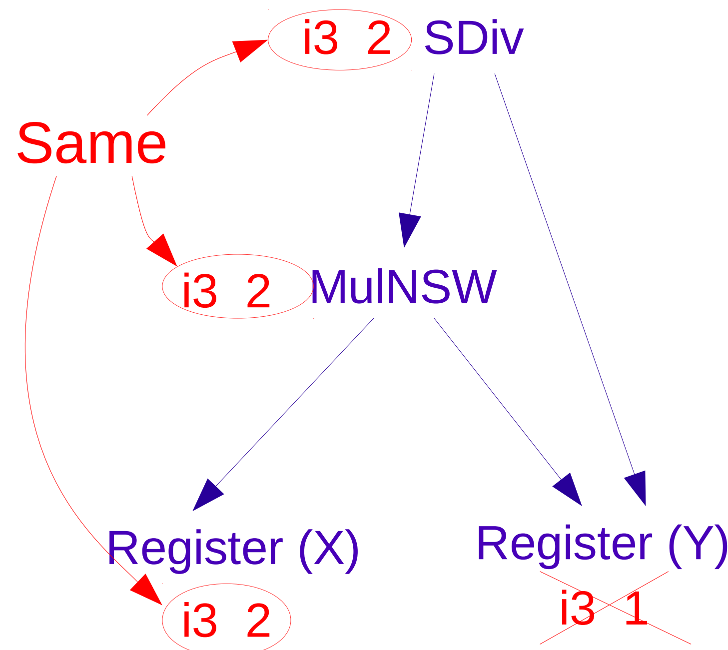
(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.

Reduce to subexpression

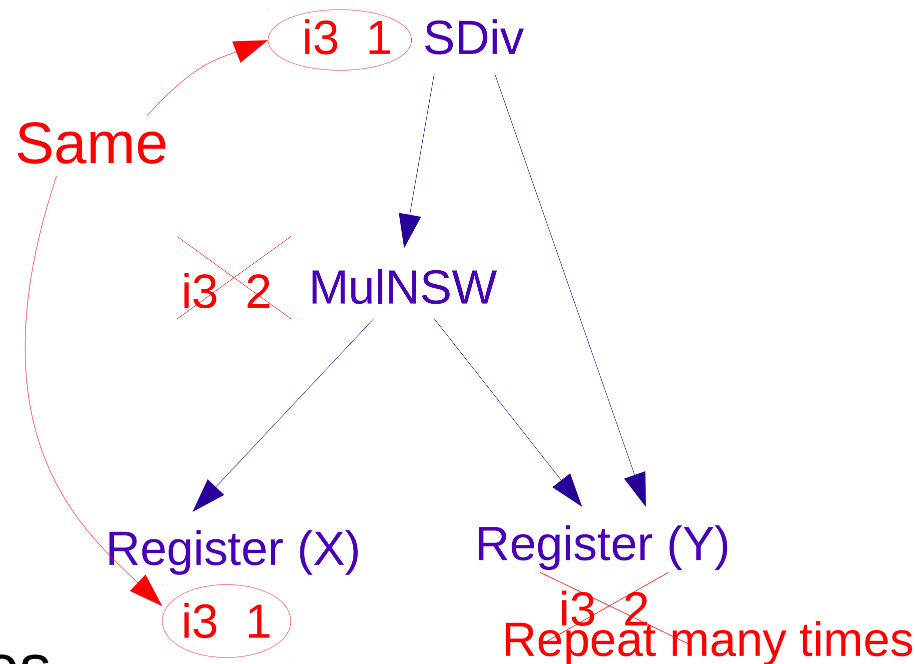
(X *nsw Y) /s Y → X



- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- See if some node always has same value as root (or undef)

Reduce to subexpression

(X *nsw Y) /s Y $\rightarrow$ X



- Assign types to nodes

  Strategies: (1) Random choice; (2) All small types.

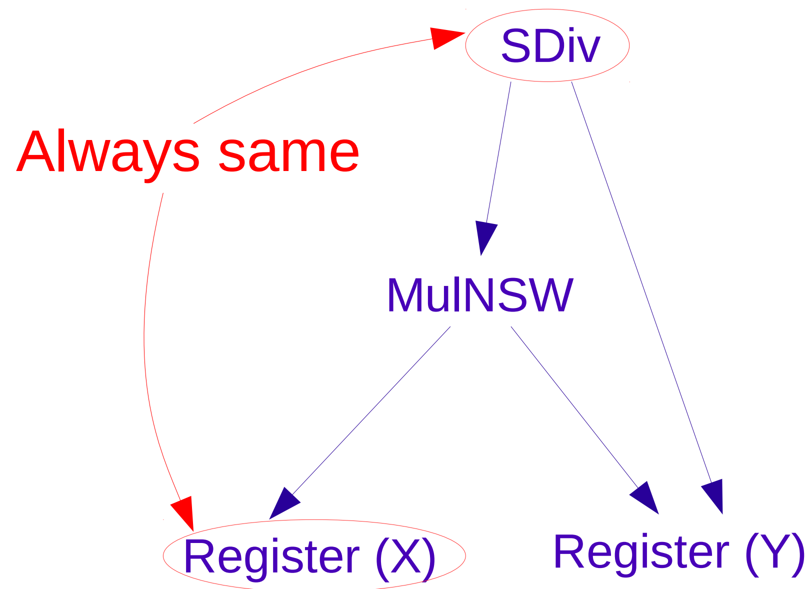
- Assign values to terminal nodes & propagate up

  Strategies: (1) Random inputs; (2) Every possible input.

- See if some node always has same value as root (or undef)

Reduce to subexpression

(X *nsw Y) /s Y → X

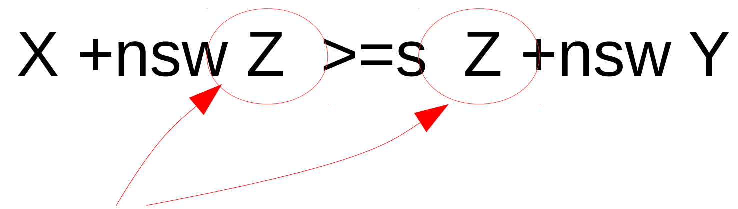


Repeat many times

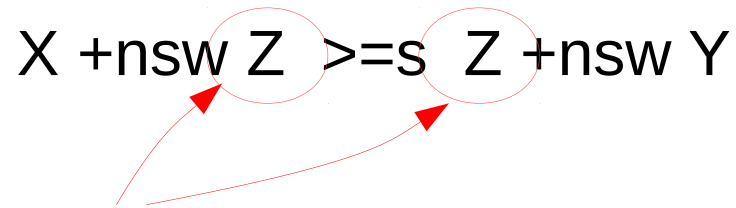
- Assign types to nodes
  Strategies: (1) Random choice; (2) All small types.
- Assign values to terminal nodes & propagate up
  Strategies: (1) Random inputs; (2) Every possible input.
- See if some node always has same value as root (or undef)

→ found a subexpression reduction

Unused variables

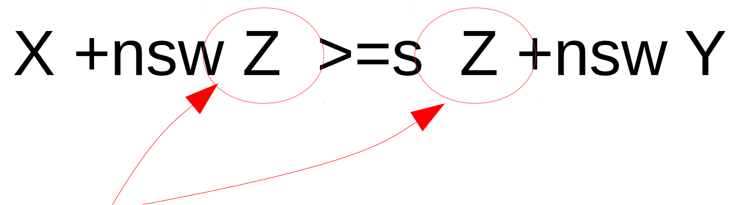


Unused variables



For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Unused variables



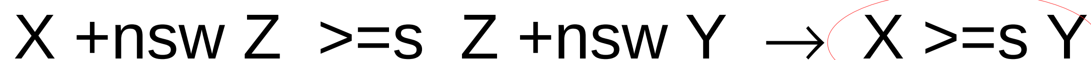
X +nsw Z >=s Z +nsw Y

Z is an “unused variable”

For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Replaced Z with 0

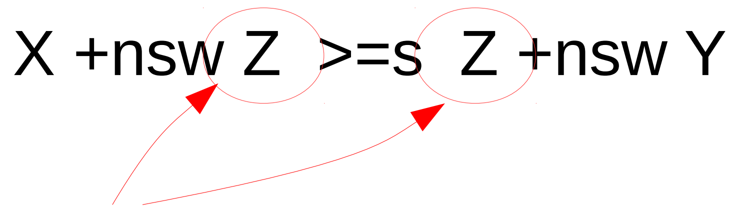
Transform:



X +nsw Z >=s Z +nsw Y → X >=s Y

Unused variables

X +nsw Z >=s Z +nsw Y



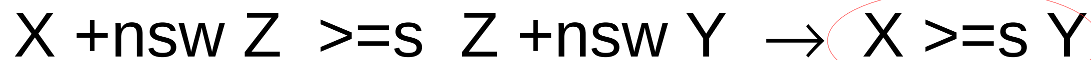
Z is an “unused variable”

For every choice of the other variables (X, Y) the result of the expression does not depend on the value of Z (or is undefined)

Replaced Z with 0

Transform:

X +nsw Z >=s Z +nsw Y → X >=s Y



Detect similarly to constant folding etc.

Examples

Unused variables found in “fully optimized” code:

- X >=s X +nsw Y
- ((X + Y) + -1) == X
- Y >>exact X == 0
- Y <<nsw X == 0

X is unused

Examples

Unused variables found in “fully optimized” code:

- X >=s X +nsw Y → 0 >=s Y
- ((X + Y) + -1) == X → Y + -1 == 0
- Y >>exact X == 0 → Y == 0
- Y <<nsw X == 0 → Y == 0

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y

Cost: 22

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30

Rule reduction

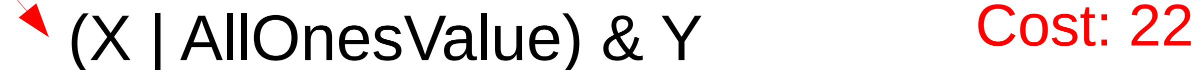
Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);             // Commutativity
rule (0 And AllBitsSet) <=> 0;           // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

(X & Y) | Y — Cost: 22

(X & Y) | (Y & AllOnesValue) — Cost: 30

(X & Y) | (AllOnesValue & Y) — Cost: 30



(X | AllOnesValue) & Y — Cost: 22

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Expression	Cost
(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Expression	Cost
(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Expression	Cost
(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);              // Commutativity
rule (0 And AllBitsSet) <=> 0;            // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Time: 1 minute	Expression	Cost
	(X & Y) | Y	Cost: 22
	(X & Y) | (Y & AllOnesValue)	Cost: 30
	(X & Y) | (AllOnesValue & Y)	Cost: 30
	(X | AllOnesValue) & Y	Cost: 22
	AllOnesValue & Y	Cost: 11
	Y	Cost: 3

Rule reduction

Requires a list of rules, eg:

```
rule (0 And 1) => (1 And 0);            // Commutativity
rule (0 And AllBitsSet) <=> 0;          // AllBitsSet is And-identity
rule ((0 Or 1) And 2) <=> ((0 And 2) Or (1 And 2)); // Distributivity
rule (0 Or AllBitsSet) => AllBitsSet; // AllBitsSet is Or-annihilator.
```

Expression	Cost
(X & Y) | Y	Cost: 22
(X & Y) | (Y & AllOnesValue)	Cost: 30
(X & Y) | (AllOnesValue & Y)	Cost: 30
(X | AllOnesValue) & Y	Cost: 22
AllOnesValue & Y	Cost: 11
Y	Cost: 3

Time: 1 minute

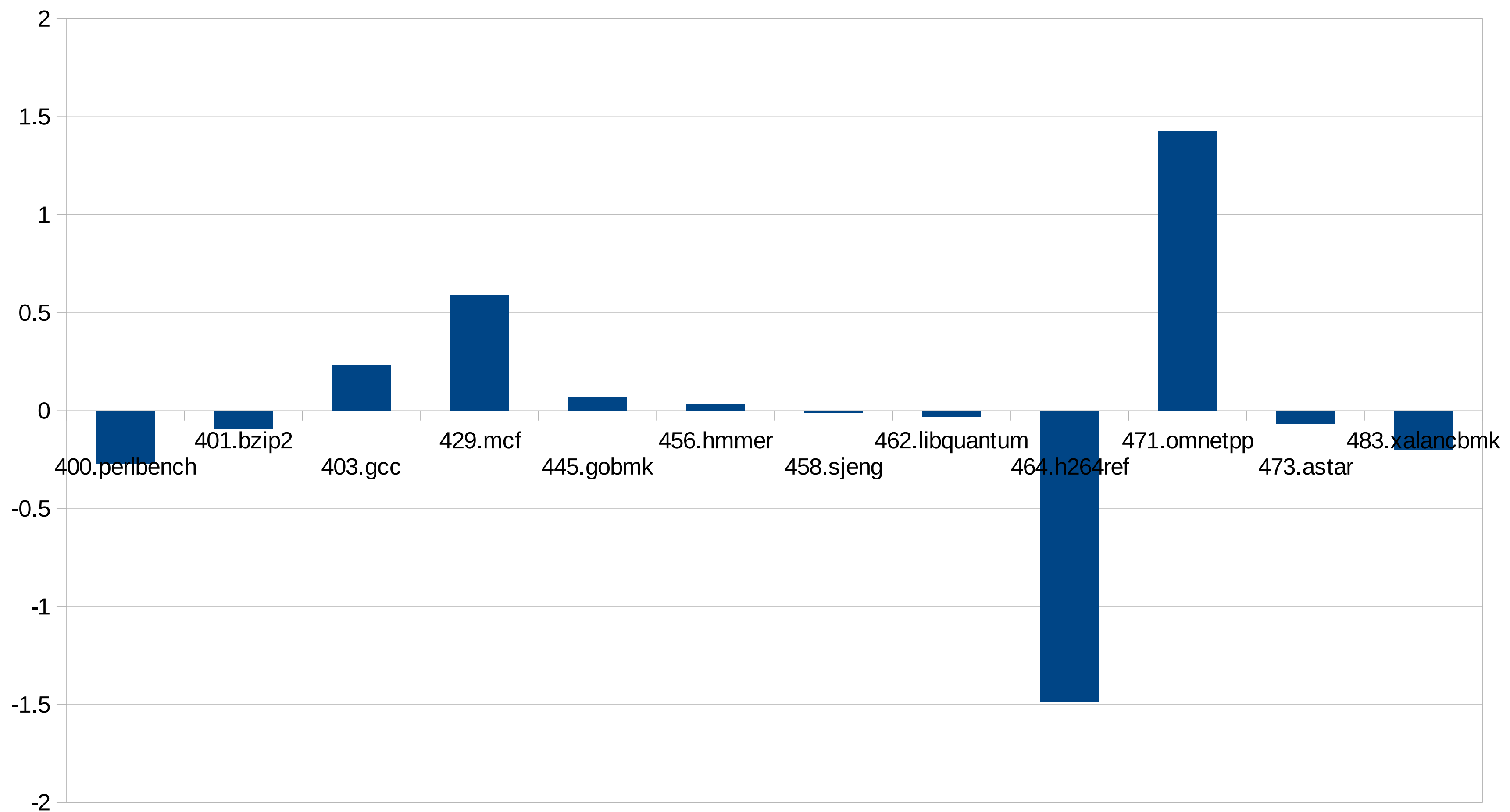
SubExpr: 0.05 secs

UnusedVar: 0.08 secs

Profit!

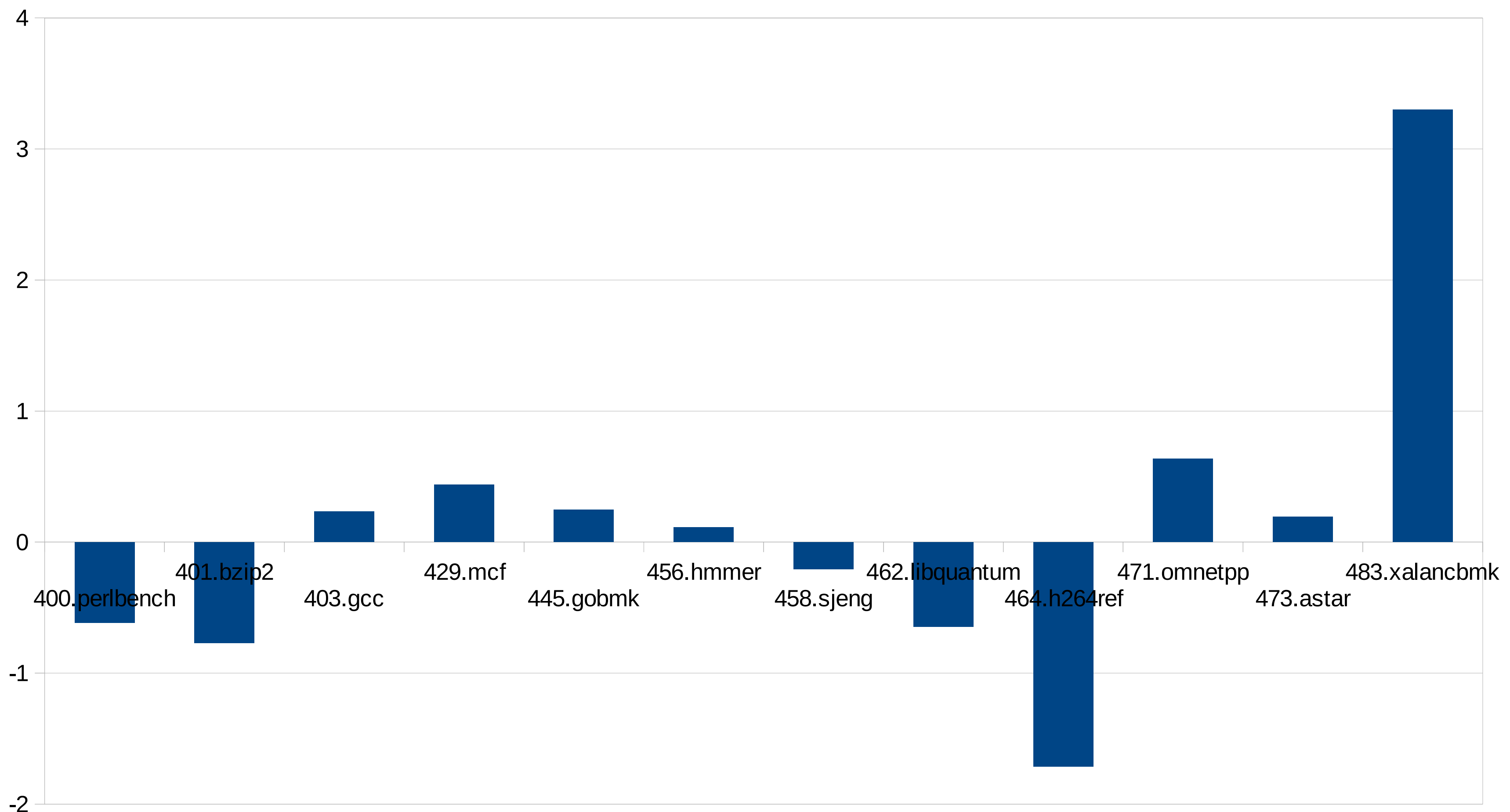
Profit?

Approximate % speed-up: constant folds



Profit?!

Approximate % speed-up: constant folds & reduce to sub-expr:



Getting it

LLVM / clang / dragonegg:

http://llvm.org

Super-optimizer:

svn://topo.math.u-psud.fr/harvest

LLVM IR

Translation unit → LLVM IR module

A module holds lists of

- Types
- Global variables
- Functions
- Aliases
- Named metadata

LLVM IR

Translation unit → LLVM IR module

A module holds lists of

- Types

```
%"struct.array2_real(kind=8)" = type
  { i8*, i64, i64, [2 x %struct.descriptor_dimension] }
%struct.descriptor_dimension = type { i64, i64, i64 }
```

- Global variables
- Functions
- Aliases
- Named metadata

LLVM IR

Translation unit → LLVM IR module

A module holds lists of

- Types
- Global variables

```
@str = internal unnamed_addr constant
    [13 x i8] c"hello world!\00"
```

- Functions
- Aliases
- Named metadata

LLVM IR

Translation unit → LLVM IR module

A module holds lists of

- Types
- Global variables
- Functions

```
define i32 @main(i32 %argc, i8** nocapture %argv)
    nounwind uwtable { ... }
```

- Aliases

```
declare i32 @puts(i8* nocapture) nounwind
```

- Named metadata

LLVM IR

Translation unit → LLVM IR module

A module holds lists of

- Types
- Global variables
- Functions
- Aliases

```
@g0 = common unnamed_addr global i32 0
@g1 = alias i32* @g0
```

- Named metadata

LLVM IR

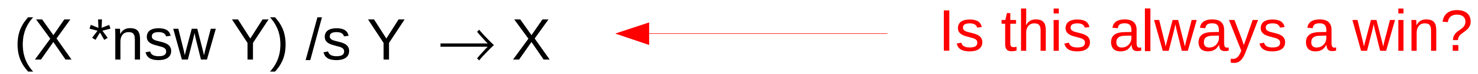
Translation unit → LLVM IR module

A module holds lists of

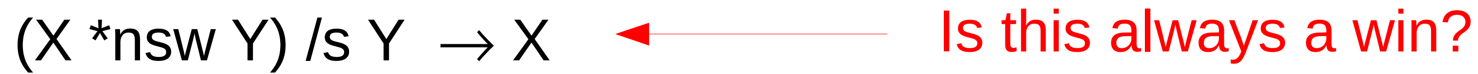
- Types
- Global variables
- Functions
- Aliases
- Named metadata

```
!0 = metadata !{metadata !"zero"}
!1 = metadata !{metadata !"one"}
!2 = metadata !{metadata !"two"}
!name = !{!0, !1, !2}
```

Register pressure

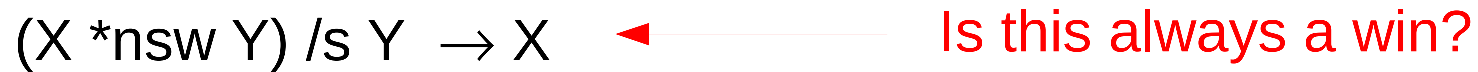


Register pressure




```
Z = X *nsw Y

...

W = Z /s Y
call @foo(W, Y, Z)
```

Register pressure



Z = X *nsw Y

X not used again

...

W = Z /s Y
call @foo(W, Y, Z)

Register pressure

(X *nsw Y) /s Y → X   Is this always a win?

Z = X *nsw Y

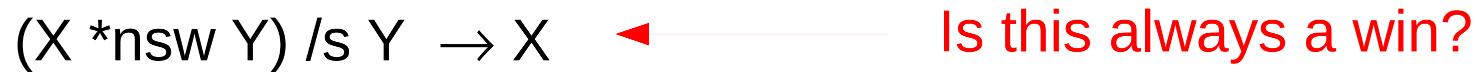
X not used again

Two registers needed (for Y, Z)

...

W = Z /s Y
call @foo(W, Y, Z)

Register pressure



Z = X *nsw Y

...

W = Z /s Y
call @foo(W, Y, Z)

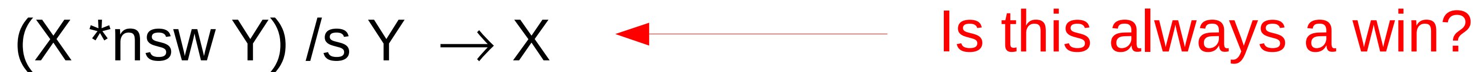
Transform: W → X

Z = X *nsw Y

...

... W not computed ...
call @foo(X, Y, Z)

Register pressure



Three registers needed (for X, Y, Z)

Z = X *nsw Y

...

... W not computed ...
call @foo(X, Y, Z)

Register pressure

(X *nsw Y) /s Y $\rightarrow$ X ← Is this always a win?

Transform increases the number of long lived registers by one.
May require spilling to the stack.