

# Low-Level Virtual Machine

## Ein Überblick

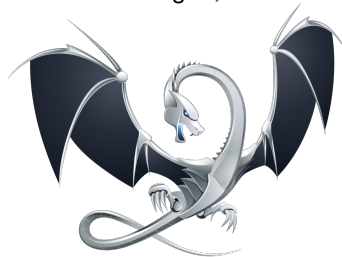


TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Compiler II: Fortgeschrittene Themen  
16. Juni 2015

Julian Oppermann

Eingebettete Systeme und Anwendungen, Technische Universität Darmstadt



- ▶ Was ist LLVM überhaupt genau?
- ▶ Entstehungsgeschichte
- ▶ Grober Überblick für die Tools
- ▶ C-Frontend
- ▶ **Einführung in die Zwischendarstellung**
- ▶ Fallstudie: Wie programmiert man eine Optimierung?
- ▶ Ein paar interessante Eigenschaften der Codeerzeugung

→ Exkurs in die echte Welt



- ▶ Grundlage für die Apple-Entwicklungswerkzeuge
  - ▶ Wenn Sie einen Mac oder ein iOS-Gerät dabei haben, läuft darauf Code, der mit LLVM übersetzt wurde.
- ▶ Safari bekommt einen LLVM-basierten Just-in-Time-Compiler für JavaScript
- ▶ NVIDIA nutzt LLVM als OpenCL-Compiler
- ▶ Google baut für PNaCl (Portable Native Client, Chrome) auf LLVM auf
- ▶ In der Forschung (einige Beispiele)
  - ▶ Nymble (High-level Synthese, von unserem Lehrstuhl)
  - ▶ LegUp (auch HLS, von der Univ. Toronto)
  - ▶ DeAliaser (Speicherabhängigkeiten ignorieren und durch Transactional Memory abfangen, Univ. Illinois, Urbana-Champaign)
  - ▶ ...



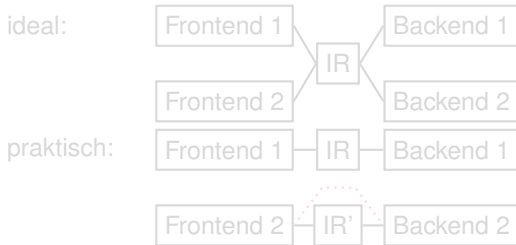
- ▶ LLVM ist **kein** Compiler!
- ▶ Die LLVM-IR ist eine Zwischendarstellung, auf die man eine Vielzahl von Quellsprachen abbilden kann.
- ▶ “The LLVM Compiler Infrastructure Project” koordiniert die Entwicklung der IR und vieler weiterer Unterprojekte.
- ▶ Typischer Einsatz als Compiler:  
`clang` liest Quelltext und *benutzt* die LLVM-Bibliotheken zum Aufbau der IR, Optimierung und Codeerzeugung.

- LLVM Core** LLVM-IR, Analysen, Optimierungen, Codeerzeugung
- clang** C/C++/Objective-C Frontend
- dragonegg** Schnittstelle zu den Frontends der GCC
- LLDB** Debugger
- libc++** C++-Standardbibliothek
- compiler-rt** Laufzeitumgebung für Architekturen, denen bestimmte Instruktionen fehlen
- vmkit** LLVM-basierte virtuelle Maschinen für Java und .NET
- polly** Schleifentransformationen zur Verbesserung der Cachelokalität und zur automatischen Parallelisierung
- ⋮

- ▶ LLVM stand ursprünglich für **Low Level Virtual Machine**.
  - ▶ Aktuell wird nur noch das Akronym verwendet, da das Projekt mittlerweile viel umfassender geworden ist.
- ▶ Begonnen Dezember 2000 von Chris Lattner und Vikram Adve an der University of Illinois.
- ▶ Heute ein erfolgreiches Open Source-Projekt unter BSD-kompatibler Lizenz.
- ▶ Gewinner des ACM Software System Award 2012.
  - ▶ In der Gesellschaft von Eclipse (2011), Java (2002), Apache (1999), WWW (1995), TCP/IP (1991), TeX (1986), UNIX (1983), ...
- ▶ Aktuelle Version: LLVM 3.6.1 (Ende Mai 2015)

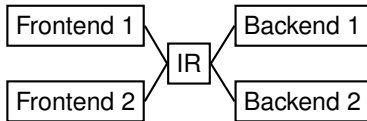


- ▶ LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- ▶ Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - ▶ Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!



- ▶ LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- ▶ Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - ▶ Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!

ideal:



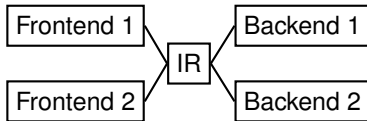
praktisch:



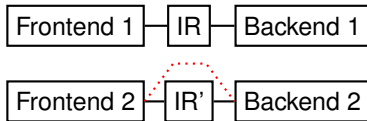


- ▶ LLVM-IR ist eigenständige Sprache, enthält alle Programminformationen.
- ▶ Ermöglicht echte Entkopplung von Frontend, Optimierungen und Codeerzeugung.
  - ▶ Im Gegensatz zu allen anderen Sprachimplementierungen zu Beginn der Entwicklung!

ideal:



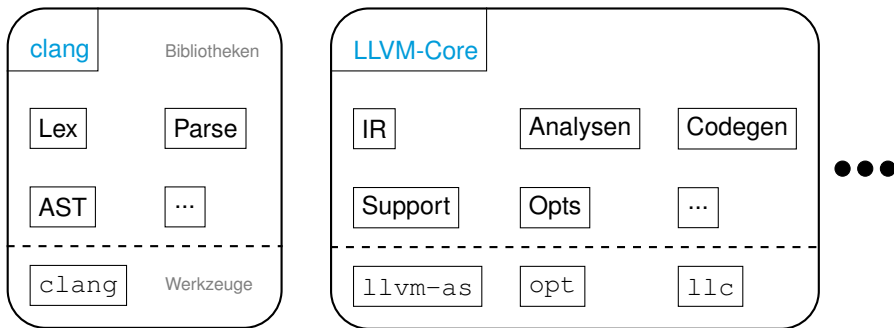
praktisch:



# LLVM-Projekt

## Was macht LLVM besonders?

- ▶ Modulare Architektur, bestehend aus wiederverwendbaren Bibliotheken





- ▶ Es gibt drei **äquivalente** Darstellungsformen:
  - ▶ textuell (Assembler-Format, siehe Beispiel) `prog.ll`
  - ▶ binär (Bitcode-Format) `prog.bc`
  - ▶ im Speicher (C++-Objekte)
- ▶ Jede Darstellungsform enthält stets alle Details des Programms  
→ klare Schnittstelle für Analysen und Transformationen.
- ▶ Für alle Analysen und Transformation wird ausschließlich diese IR verwendet.

# LLVM-Projekt

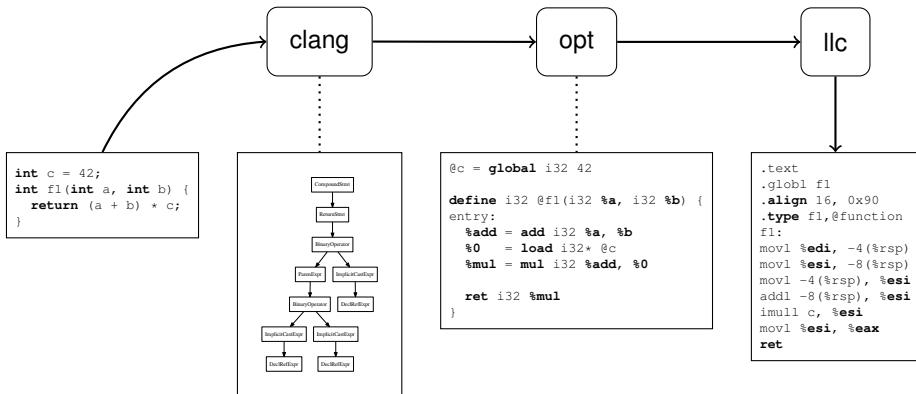
## Was macht LLVM besonders?

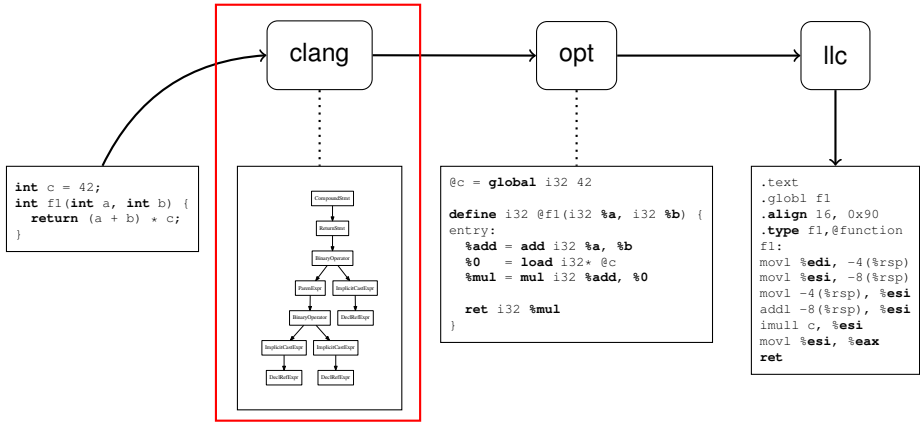


- ▶ clang
  - ▶ Schneller als gcc (compile time)
  - ▶ Bessere Fehlermeldungen

```
$ gcc-4.2 -fsyntax-only t.c
t.c:7: error: invalid operands to binary + (have 'int' and 'struct A')
$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
    return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                        ~~~~~ ^ ~~~~~
```

- ▶ LLVM Core
  - ▶ Aggressive skalare Optimierungen
  - ▶ Link-time optimization
  - ▶ “easily hackable”





- ▶ Benutzt einen handgeschriebenen Parser nach dem Prinzip des rekursiven Abstiegs.
- ▶ Beispiel: if-Statement (stark vereinfacht)

```
StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {
    SourceLocation IfLoc = ConsumeToken(); // eat the 'if'.

    if (Tok.isNot(tok::l_paren)) return StmtError();

    if (ParseParenExprOrCondition(CondExp, CondVar, IfLoc, true)) return StmtError();

    StmtResult ThenStmt (ParseStatement(&InnerStatementTrailingElseLoc));

    StmtResult ElseStmt;
    if (Tok.is(tok::kw_else))
        ElseStmt = ParseStatement();

    return Actions.ActOnIfStmt(...);
}
```

- ▶ Klassenhierarchien für Deklarationen (`Decl`), Anweisungen (`Stmt`) und Typen (`Type`).
  - ▶ Ausdrücke (`Expr`) sind Unterklassen von `Stmt`.
- ▶ Wurzelknoten ist `TranslationUnitDecl`.
- ▶ Keine gemeinsame Oberklasse, jeder Knotentyp spezifiziert seine eigenen Zugriffsmethoden:

```
class IfStmt : public Stmt {  
    ...  
    Expr *getCond() { return reinterpret_cast<Expr*>(SubExprs[COND]); }  
    Stmt *getThen() { return SubExprs[THEN]; }  
    Stmt *getElse() { return SubExprs[ELSE]; }  
    ...  
}
```

- ▶ Traversierung mittels `RecursiveASTVisitor` (“Makromonster”).



# clang

## AST (Beispiel)



```
int
```

```
addabs(int a,  
       int b)
```

```
{  
    int x;  
    if (a*b >= 0)  
        x = a+b;  
    else  
        x = a-b;  
    return x;  
}
```

```
TranslationUnitDecl 0x5ff6ba0 <<invalid sloc>>  
  '-FunctionDecl 0x5ff75d0 <../llvm-vortrag/cfg.c:1:1, line:8:1> addabs 'int (int, int)'  
    |-ParmVarDecl 0x5ff7490 <line:1:12, col:16> a 'int'  
    |-ParmVarDecl 0x5ff7500 <col:19, col:23> b 'int'  
    '-CompoundStmt 0x6023f10 <col:26, line:8:1>  
      |-DeclStmt 0x5ff76e8 <line:2:2, col:7>  
        |-VarDecl 0x5ff7690 <col:2, col:6> x 'int'  
        |-IfStmt 0x6023e80 <line:3:2, line:6:9>  
          | |-<<NULL>>>  
          | |-BinaryOperator 0x5ff77c8 <line:3:6, col:13> 'int' '>='  
          | | |-BinaryOperator 0x5ff7780 <col:6, col:8> 'int' '*'  
          | | | '-DeclRefExpr 0x5ff7700 <col:6> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'  
          | | | '-DeclRefExpr 0x5ff7728 <col:8> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'  
          | | '-IntegerLiteral 0x5ff77a8 <col:13> 'int' 0  
          | |-BinaryOperator 0x6023d60 <line:4:3, col:9> 'int' '='  
          | | |-DeclRefExpr 0x5ff77f0 <col:3> 'int' lvalue Var 0x5ff7690 'x' 'int'  
          | | '-BinaryOperator 0x5ff7898 <col:7, col:9> 'int' '+'  
          | | | '-DeclRefExpr 0x5ff7818 <col:7> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'  
          | | '-DeclRefExpr 0x5ff7840 <col:9> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'  
          | '-BinaryOperator 0x6023e58 <line:6:3, col:9> 'int' '='  
          | | |-DeclRefExpr 0x6023d88 <col:3> 'int' lvalue Var 0x5ff7690 'x' 'int'  
          | '-BinaryOperator 0x6023e30 <col:7, col:9> 'int' '-'  
          | | '-DeclRefExpr 0x6023db0 <col:7> 'int' lvalue ParmVar 0x5ff7490 'a' 'int'  
          | | '-DeclRefExpr 0x6023dd8 <col:9> 'int' lvalue ParmVar 0x5ff7500 'b' 'int'  
          '-ReturnStmt 0x6023ef0 <line:7:2, col:9>  
            '-DeclRefExpr 0x6023eb0 <col:9> 'int' lvalue Var 0x5ff7690 'x' 'int'
```



### ► (Gekürzte) LLVM-IR-Generierung für ein If-Statement

```
void CodeGenFunction::EmitIfStmt(const IfStmt &S) {
    llvm::BasicBlock *ThenBlock = createBasicBlock("if.then");
    llvm::BasicBlock *ContBlock = createBasicBlock("if.end");
    llvm::BasicBlock *ElseBlock = ContBlock;
    if (S.getElse())
        ElseBlock = createBasicBlock("if.else");
    EmitBranchOnBoolExpr(S.getCond(), ThenBlock, ElseBlock);

    EmitBlock(ThenBlock);
    EmitStmt(S.getThen());
    EmitBranch(ContBlock);

    if (const Stmt *Else = S.getElse()) {
        EmitBlock(ElseBlock);
        EmitStmt(Else);
        EmitBranch(ContBlock);
    }

    EmitBlock(ContBlock, true);
}
```



# LLVM-IR

## Ein Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
int c = 42;  
int f1(int a, int b) { return (a + b) * c; }
```

# LLVM-IR

## Ein Beispiel



```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42
```

```
define i32 @f1(i32 %a, i32 %b) {
entry:
    %add = add i32 %a, %b
    %0   = load i32* @c
    %mul = mul i32 %add, %0

    ret i32 %mul
}
```

# LLVM-IR

## Ein Beispiel

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42
```

Globale Variable

```
define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b
  %0   = load i32* @c
  %mul = mul i32 %add, %0

  ret i32 %mul
}
```

Funktionsdefinition

# LLVM-IR

## Ein Beispiel



```
int c = 42;  
int f1(int a, int b) { return (a + b) * c; }
```

```
@c = global i32 42  
↑           ↑   ↑  
Name       Typ Initialer Wert
```

# LLVM-IR

## Ein Beispiel

```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

Rückgabebetyp      Funktionsname      Argumente

↓                    ↓                    ↓

```
define i32 @f1(i32 %a, i32 %b) {
```



# LLVM-IR

## Ein Beispiel



```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

```
define i32 @f1(i32 %a, i32 %b) {
entry:
  %add = add i32 %a, %b ← Operanden
}
```

↑ Zielregister

↑ Opcode

↑ Ergebnistyp

# LLVM-IR

## Ein Beispiel



```
int c = 42;
int f1(int a, int b) { return (a + b) * c; }
```

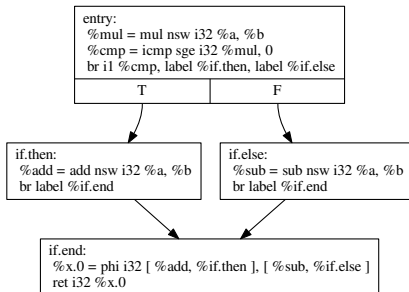
```
@c = global i32 42
```

```
define i32 @f1(i32 %a, i32 %b) {
entry:
    %add = add i32 %a, %b
    %0   = load i32* @c
    %mul = mul i32 %add, %0

    ret i32 %mul
}
```



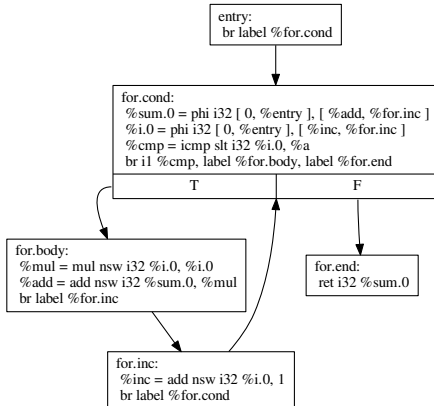
```
int addabs(int a, int b) {
    int x;
    if (a*b >= 0)
        x = a+b;
    else
        x = a-b;
    return x;
}
```



CFG for 'addabs' function



```
int sumsquares(int a) {
    int sum = 0, i;
    for (i=0; i<a; i++) {
        sum += i*i;
    }
    return sum;
}
```



CFG for 'sumsquares' function



- ▶ Erkenntnis: Sieht aus wie eine Assembler-Darstellung für einen RISC-Prozessor.
  - ▶ unendlich viele Register
  - ▶ Jedes Register kann nur einmal von einer eindeutig bestimmten Instruktion beschrieben werden (→ **SSA-Form**).
  - ▶ typisiert
- ▶ “low level”: im Kontrast zu Java / .NET VMs
  - ▶ keine Klassen/Objekte
  - ▶ keine Vererbung
  - ▶ keine Polymorphie
  - ▶ kein Exception Handling
  - ▶ ...
  - ▶ **Aber:** alle diese Konstrukte lassen sich auf LLVM-IR abbilden!



**phi** Die  $\Phi$ -Funktion der SSA-Form

- ▶ explizite Instruktion, Tupel von Wert und Label als Argumente
- ▶ Beispiel:

```
%x = phi i32 [ %add, %then ], [ %sub, %else ]
```

**call** Funktionsaufruf

- ▶ abstrahiert Aufrufkonventionen, erhält Funktionsname und -argumente als Parameter
- ▶ Beispiel: **%y = call** i32 @Get\_Bits(i32 1)

**switch** Switch-Instruktion

- ▶ Beispiel:

```
switch i32 %val, label %def [  
    i32 0, label %onzero,  
    i32 1, label %onone,  
    i32 2, label %ontwo ]
```

Jeder Wert hat einen Typ (unabhängig von der Quellsprache)!

- ▶ Integerwerte: `i1`, `i8`, `i16`, `i32`, ...  
alle Bitbreiten möglich, keine signed/unsigned-Unterscheidung
- ▶ Fließkommazahlen: `half`, `float`, `double`, ...
- ▶ Zeiger: `i64*`
- ▶ Arrays: `[10 x i32]`, `[2 x [2 x float]]`
- ▶ Strukturen: `{i32, float, i32}`
- ▶ Vektoren (SIMD): `<i8, i8, i8, i8>`
- ▶ ...



- ▶ *Typsichere* Adressrechnung für Array- oder Struktur-Elemente (wichtig für Optimierungen!)
  - ▶ Erhält einen Basiszeiger und eine Folge von Indizes; liefert einen Zeiger. Macht keine Speicherzugriffe!
  - ▶ Beispiel: `int arr[2]; arr[1] = ...` →  
*; Alloziert Speicher auf dem Stack*  
`%arr = alloca [2 x i32]`  
  
*; Liefert i32\**  
`%z = getelementptr [2 x i32]* %arr, i32 0, i32 1`
  - ▶ Warum ist der erste Index 0?
- Auch der Base-Pointer muss indiziert werden!





```
struct munger_struct {
    int f1;
    int f2;
};

void munge(struct munger_struct *P) {
    P[0].f1 = P[1].f1 + P[2].f2;
}

...
munger_struct Array[3];
...
munge(Array);
```

Beispiel von <http://llvm.org/docs/GetElementPtr.html>



```
void munge(struct munger_struct *P) {  
    P[0].f1 = P[1].f1 + P[2].f2;  
}
```

```
void %munge(%struct.munger_struct* %P) {  
entry:  
    %tmp = getelementptr %struct.munger_struct* %P, i32 1, i32 0  
    %tmp = load i32* %tmp  
    %tmp6 = getelementptr %struct.munger_struct* %P, i32 2, i32 1  
    %tmp7 = load i32* %tmp6  
    %tmp8 = add i32 %tmp7, %tmp  
    %tmp9 = getelementptr %struct.munger_struct* %P, i32 0, i32 0  
    store i32 %tmp8, i32* %tmp9  
    ret void  
}
```



```
%x = getelementptr  
    {i16, [4 x [4 x float]]}* %ptr,  
    i32 2, i32 1, i32 3, i32 0
```

- ▶ Welchen Typ hat `x`?
- ▶ Welches Offset (in Bytes) hat `x` zu `ptr`?

# LLVM-IR

## C++ Beispiel



```
#include <iostream>
using namespace std;

class A {
int x;
public:
    virtual void foo() { cout << "A::foo" << endl; }
};

class B : public A {
public:
    virtual void foo() { cout << "B::foo" << endl; }
};

int main() {
    A *ab = new B();
    ab->foo();
    return 0;
}
```



# LLVM-IR

## C++ Beispiel



```
%class.B = type { %class.A }  
%class.A = type { i32 (...)**, i32 }
```

```
@_ZTV1B = linkonce_odr unnamed_addr constant [3 x i8*] [i8* null,  
i8* bitcast ({ i8*, i8*, i8* }* @_ZTI1B to i8*),  
i8* bitcast (void (%class.B)* @_ZN1B3fooEv to i8*)]
```

```
@_ZTV1A = linkonce_odr unnamed_addr constant [3 x i8*] [i8* null,  
i8* bitcast ({ i8*, i8* }* @_ZTI1A to i8*),  
i8* bitcast (void (%class.A)* @_ZN1A3fooEv to i8*)]
```

# LLVM-IR

## C++ Beispiel



```
define i32 @main() #2 {  
entry:  
  %call = call noalias i8* @_Znwm(i64 16)  
  %0 = bitcast i8* %call to %class.B*  
  %1 = bitcast %class.B* %0 to i8*  
  call void @llvm.memset.p0i8.i64(i8* %1, i8 0, i64 16, i32 8, i1 false)  
  call void @_ZN1BC1Ev(%class.B* %0) #1  
  %2 = bitcast %class.B* %0 to %class.A*  
  %3 = bitcast %class.A* %2 to void (%class.A*)***  
  %vtable = load void (%class.A*)*** %3  
  %vfn = getelementptr inbounds void (%class.A*)** %vtable, i64 0  
  %4 = load void (%class.A*)** %vfn  
  call void %4(%class.A* %2)  
  ret i32 0  
}
```

# Unions

```
#include <stdio.h>

typedef union {
    long long ll;
    long      l;
    int       i;
    short     s;
    char      c;
} ints_u;

int main() {
    ints_u U;
    U.ll = 0xDEADBEEFCAFEBABELL;

    printf ("U.s = %d\n", U.s);
    return 0;
}
```

- Typ des größten Elements
- Frontend erzeugt bitcasts für Zugriff auf andere Elemente

```
; ModuleID = 'union.ll'
target datalayout = "e-p:64:64:64-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-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.9.0"

%union.ints_u = type { i64 }

@.str = private unnamed_addr constant [10 x i8] c"U.s = %d\0A
\00", align 1

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
    %U = alloca %union.ints_u, align 8
    %1 = bitcast %union.ints_u* %U to i64*
    store i64 -2401053089206453570, i64* %1, align 8
    %2 = bitcast %union.ints_u* %U to i16*
    %3 = load i16* %2, align 2
    %4 = sext i16 %3 to i32
    %5 = call i32 (i8*, ...)* @printf(i8*
getelementptr inbounds ([10 x i8]* @.str, i32 0, i32 0), i32 %4)
    ret i32 0
}
```



# λ-Funktionen (aka Closures)

## Neues Sprachfeature in C++11

```
$ clang++ --std=c++11 lambda.cpp
```

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;

    auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };
    func(42);

    auto func2 = [] (const string &s) -> void { cout << "Have a " << s << " day!" << endl; };
    func2("nice");

    return 0;
}
```

# λ-Funktionen (aka Closures)

## Neues Sprachfeature in C++11

```
$ clang++ --std=c++11 lambda.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    int x;  
    cin >> x;
```

```
    auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };  
    func(42);
```

```
    auto func2 = [] (const string &s) -> void { cout << "Have a " << s << " day!" << endl; };  
    func2("nice");
```

Capture specification

[]	„Nichts“
[&]	„By-Reference“
[=]	„By-Value“

```
    return 0;
```

```
}
```

# λ-Funktionen (aka Closures)

## Neues Sprachfeature in C++11

```
$ clang++ --std=c++11 lambda.cpp
```

```
#include <iostream>
```

```
using namespace std; Parameterliste
```

```
int main() {
    int x;
    cin >> x;
```

```
    auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };
    func(42);
```

```
    auto func2 = [] (const string &s) -> void { cout << "Have a " << s << " day!" << endl; };
    func2("nice");
```

```
    return 0;
```

```
}
```

**Capture specification**

[]	„Nichts“
[&]	„By-Reference“
[=]	„By-Value“

# λ-Funktionen (aka Closures)

## Neues Sprachfeature in C++11

```
$ clang++ --std=c++11 lambda.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int x;
    cin >> x;
```

Parameterliste

Rückgabewert (optional)

```
auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };
func(42);
```

```
auto func2 = [] (const string &s) -> void { cout << "Have a " << s << " day!" << endl; };
func2("nice");
```

Capture specification

[]	„Nichts“
[&]	„By-Reference“
[=]	„By-Value“

```
return 0;
```

```
}
```

# $\lambda$ -Funktionen (aka Closures)

- Lowering passiert im Frontend
- Für (fast) jede  $\lambda$ -Funktion wird eine neue Klasse erzeugt:
  - Der ()-Operator wird entsprechend (Parameter, Rückgabewert, Rumpf) überschrieben.
  - Ein Konstruktor zur Übergabe der gecapture'ten Variablen wird erzeugt.
  - Ausnahme:  $\lambda$ s, die keine Variablen capturen, werden zu Funktionen.

Quelle (und gute Erklärung): <http://www.cprogramming.com/c++11/c++11-lambda-closures.html>

# λ-Funktionen (aka Closures)

Darstellung in LLVM (Ausschnitt aus ~ 1000 Zeilen LLVM Assembler... )

```
auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };  
func(42);
```

```
define i32 @main() #0 {  
  %x = alloca i32, align 4  
  %func = alloca %class.anon, align 8           ; %class.anon = type { i32* }  
  %func2 = alloca %class.anon.0, align 1       ; %class.anon.0 = type { i8 }  
  ...  
  %3 = getelementptr inbounds %class.anon* %func, i32 0, i32 0  
  store i32* %x, i32** %3, align 8  
  call void @"_ZZ4mainENK3$_0cLEi"(%class.anon* %func, i32 42)
```

# λ-Funktionen (aka Closures)

Darstellung in LLVM (Ausschnitt aus ~ 1000 Zeilen LLVM Assembler... )

```
auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };  
func(42);
```

Closure-Objekt

```
define i32 @main() #0 {  
  %x = alloca i32, align 4  
  %func = alloca %class.anon, align 8  
  %func2 = alloca %class.anon.0, align 1  
  ...  
  %3 = getelementptr inbounds %class.anon* %func, i32 0, i32 0  
  store i32* %x, i32** %3, align 8  
  call void @"_ZZ4mainENK3$_0cLEi"(%class.anon* %func, i32 42)
```

*(Note: In the original image, an arrow points from the text 'Closure-Objekt' to the line '%func = alloca %class.anon, align 8' in the LLVM assembly code.)*

# λ-Funktionen (aka Closures)

Darstellung in LLVM (Ausschnitt aus ~ 1000 Zeilen LLVM Assembler... )

```
auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };  
func(42);
```

```
define i32 @main() #0 {  
  %x = alloca i32, align 4  
  %func = alloca %class.anon, align 8  
  %func2 = alloca %class.anon.0, align 1  
  ...  
  %3 = getelementptr inbounds %class.anon* %func, i32 0, i32 0  
  store i32* %x, i32** %3, align 8  
  call void @"_ZZ4mainENK3$_0cLEi"(%class.anon* %func, i32 42)
```

Closure-Objekt

Adresse von x captuern  
(statt Konstruktor-Aufruf)



# λ-Funktionen (aka Closures)

Darstellung in LLVM (Ausschnitt aus ~ 1000 Zeilen LLVM Assembler... )

```
auto func = [&] (int i) -> void { cout << "Hello, No. " << i << ", you typed " << x << endl; };  
func(42);
```

```
define i32 @main() #0 {  
  %x = alloca i32, align 4  
  %func = alloca %class.anon, align 8  
  %func2 = alloca %class.anon.0, align 1  
  ...  
  %3 = getelementptr inbounds %class.anon* %func, i32 0, i32 0  
  store i32* %x, i32** %3, align 8  
  call void @"_ZZ4mainENK3$_0cLEi"(%class.anon* %func, i32 42)
```

Closure-Objekt

Adresse von x captuern  
(statt Konstruktor-Aufruf)

Die Methode aufrufen

# λ-Funktionen (aka Closures)

```
auto func2 = [] (const string &s) -> void { cout << "Have a " << s << " day!" << endl; };  
func2("nice");
```

invoke void

```
@,,_ZZ4mainENK3$_1cLERKNSt3__112basic_stringIcNS0_11char_traitsIcEENS0_9allocatorIcEEEE"(  
%class.anon.0* %func2, %"class.std::__1::basic_string"* %1)  
to label %10 unwind label %11
```

Davor: String-Initialisierung  
Danach: Exception Handling

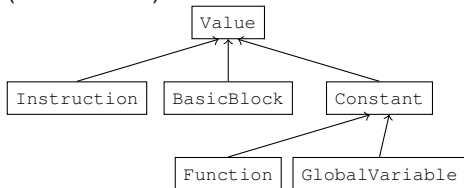
- ▶ Steuerfluss: `ret` `br` `switch` `indirectbr` `invoke` `resume` `unreachable`
- ▶ Arithmetisch: `add` `fadd` `sub` `fsub` `mul` `fmul` `udiv` `sdiv` `fdiv` `urem`  
`srem` `frem` `shl` `lshr` `ashr` `and` `or` `xor`
- ▶ Elementzugriff: `extractelement` `insertelement` `shufflevector` `extractvalue`  
`insertvalue`
- ▶ Speicher und Adressierung: `alloca` `load` `store` `fence` `cmpxchg`  
`atomicrmw` `getelementptr`
- ▶ Konversionen: `trunc` `zext` `sext` `fptrunc` `fpext` `fptoui` `fptosi` `uitofp`  
`sitofp` `ptrtoint` `inttoptr` `bitcast`
- ▶ Andere: `icmp` `fcmp` `phi` `select` `call` `va_arg` `landingpad`



## Implementieren Sie die Fibonacci-Funktion in LLVM!

```
<result> = add <ty> <op1>, <op2>
<result> = sub <ty> <op1>, <op2>
<result> = call <ty> <fnptrval>(<function args>)
<result> = icmp <cond> <ty> <op1>, <op2> ;
                (cond = eq, ne,
                  ugt, uge, ult, ule,
                  sgt, sge, slt, sle)
```

- ▶ Oberklasse für fast alle IR-Elemente: `Value` modelliert (SSA-)Werte.
  - ▶ Jeder Wert hat einen `Type`.
- ▶ (Vereinfachte) Klassenhierarchie:



- ▶ `Instructions` speichern ihre Operanden als Zeiger zu anderen `Value`-Objekten.
- ▶ **Beispiel: Konstruktor von `BranchInst`:**  
`BranchInst(BasicBlock *IfTrue, BasicBlock *IfFalse, Value *Cond)`

# „Navigation“ in der IR

- Argumente
  - `Instruction *I = ... ; I->getOperand(1);`
  - `BranchInst *Br = ...; Br->getCondition();`
  - `CallInst *Ci = ...; Ci->getArgOperand(4); Ci->getCalledFunction();`
- CFG
  - `Function *F = ...; F->getEntryBlock();`
  - `BasicBlock *BB = ...;`
    - `for (pred_iterator PI = pred_begin(BB), PE = pred_end(BB); PI != PE; PI++) { ...`
    - `for (succ_iterator SI = succ_begin(BB), SE = succ_end(BB); SI != SE; SI++) { ...`
    - `TerminatorInst *TI = BB->getTerminator();`
    - `if (BranchInst *BI = dyn_cast<BranchInst>(TI))`  
`BI->getSuccessor(1) // „false“`

# „Navigation“ in der IR

```
Module *M = ...;
for (Module::iterator MI = M->begin(), ME = M->end();
    MI != ME; MI++) {
    Function *F = MI;
    for (Function::iterator FI = F->begin(), FE = F->end();
        FI != FE; FI++) {
        BasicBlock *BB = FI;
        for (BasicBlock::iterator BBI = BB->begin(), BBE = BB->end();
            BBI != BBE; BBI++) {
            Instruction *I = BBI;

            assert (I->getParent() == BB);
            assert (I->getParent()->getParent() == F);
            assert (I->getParent()->getParent()->getParent() == M);
        }
    }
}
```

# „Navigation“ in der IR (neu)

```
Module &M = ...;
```

```
for (auto &F : M) {  
    for (auto &BB : F) {  
        for (auto &I : BB) {  
            assert (I.getParent() == &BB);  
            assert (I.getParent()->getParent() == &F);  
            assert (I.getParent()->getParent()->getParent() == &M);  
        }  
    }  
}
```



# „Navigation“ in der IR

- Jeder Value hat eine Liste von Usern

```
Value *V = ...;  
for (use_iterator UI* = V->use_begin(), UE = V->use_end();  
     UI != UE; UI++) {  
    User *U = UI;
```

- User sind alle IR-Elemente, die andere Werte referenzieren können
  - Hauptsächlich Instructions und Konstanten
- Beispiel: Alle (direkten) Aufrufstellen einer Funktion F

```
for (use_iterator UI* = F->use_begin(), UE = F->use_end();  
     UI != UE; UI++) {  
    if (CallInst *CI = dyn_cast<Function>(UI)) {  
        ...
```

# InstVisitor

```
class MyVisitor : public InstVisitor<MyVisitor> {  
public:  
    void visitAdd(BinaryOperator &I) { ... }  
  
    void visitStoreInst(StoreInst &I) { ... }  
    void visitTerminatorInst(TerminatorInst &I) { ... }  
};
```

```
// somewhere  
Function &F = ...;  
MyVisitor V; V.visit(F);
```

- Überschreibbare Methoden für Opcodes, spezifische Instruktionen, und Klassen von Instruktionen.
- Makromagie, keine `accept(...)`-Methode in Instruction-Hierarchie.

# (Post)DominatorTree

- Im Pass die Analyse anfordern

```
void getAnalysisUsage(AnalysisUsage &AU) const {  
    AU.addRequired<DominatorTree>();  
}
```

- Verwendung

```
DominatorTree &DT = getAnalysis<DominatorTree>();  
if (DT.dominates(BB1, BB2)) ...
```

```
DomTreeNode *DTN1 = DT[BB3], *DTN2;  
DTN2 = DTN1->getIDom();
```

```
DT.findNearestCommonDominator(DTN1, DTN2);
```

# LoopInfo

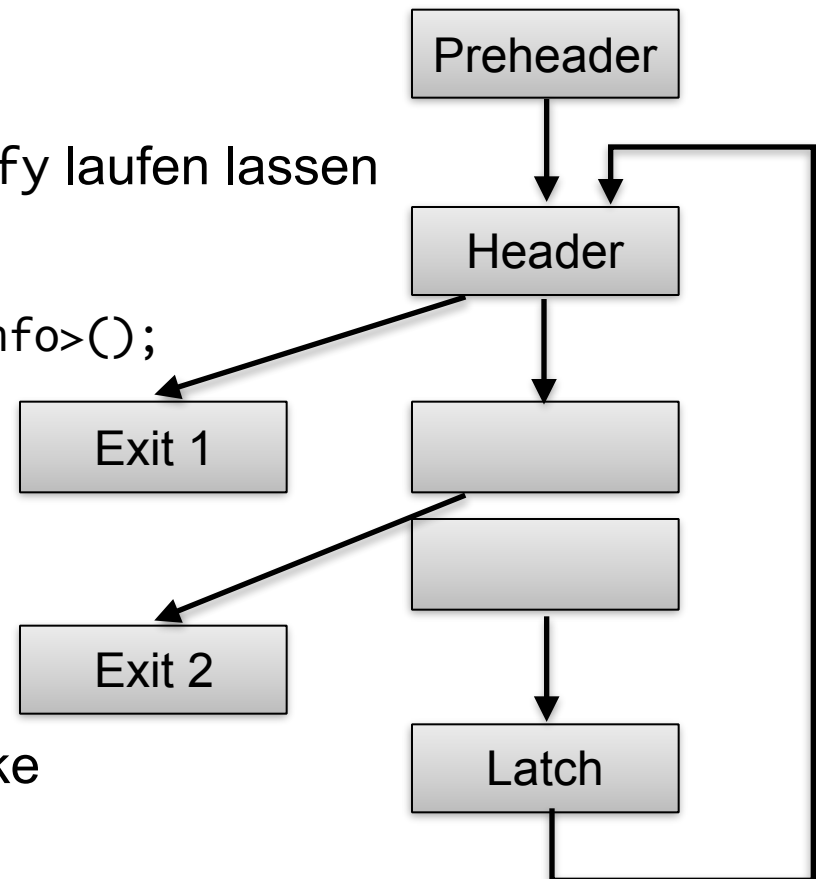
- Liefert natürliche Schleifen (= ein Eintrittspunkt)
- Im Pass die Analyse anfordern  
`AU.addRequired<LoopInfo>();`
- Sehr sinnvoll: Vorher `-loop-simplify` laufen lassen

- Verwendung

```
LoopInfo &LI = getAnalysis<LoopInfo>();
for (LoopInfo::iterator ...
    // Toplevel Loops
```

```
Loop *L = LI.getLoopFor(BB);
unsigned d = LI.getLoopDepth(BB);
```

- Verwendung der Loop-Objekte
  - Zugriff auf ausgezeichnete Blöcke  
(falls eindeutig)



- ▶ LLVM-IR enthält genug Informationen, um auch “high-level” Analysen und Transformationen durchzuführen.
- ▶ Gekapselt als Pässe.
- ▶ Auszug aus der Liste der mitgelieferten Pässe:
  - ▶ Analysen: (Post-)Dominatorbaum, natürliche Schleifen, Aliasanalyse(n), ...
  - ▶ Transformationen: Dead Code Elimination, Reassociation, Loop Invariant Code Motion, Global Value Numbering, ...
- ▶ Abhängigkeiten zwischen Pässen werden automatisch aufgelöst.
- ▶ Man kann sogar Transformationen einzeln auf ein Programm anwenden:

```
$ opt -S -reassociate -o prog_opt.ll prog.ll  
$ opt -S -licm -o prog_opt2.ll prog_opt.ll
```

# Optimierungen

## Konstantenpropagation

```
while (!WorkList.empty()) {
    Instruction *I = *WorkList.begin(); WorkList.erase(WorkList.begin());

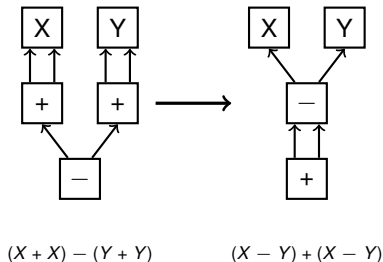
    if (Constant *C = ConstantFoldInstruction(I, TD, TLI)) {
        // Add all of the users of this instruction to the worklist, they might
        // be constant propagatable now...
        for (Value::use_iterator UI = I->use_begin(), UE = I->use_end(); UI != UE; ++UI)
            WorkList.insert(cast<Instruction>(*UI));

        // Replace all of the uses of a variable with uses of the constant.
        I->replaceAllUsesWith(C);

        // Remove the dead instruction.
        WorkList.erase(I);
        I->eraseFromParent();
    }
}
```

- ▶ Inhalt von `ConstantFoldInstruction`: “Erzeuge neue Konstante, wenn alle Operanden von `I` konstant sind”

## Implementierung einer Peephole-Optimierung



(aus: Bersch, Thomas: Generierung lokaler Optimierungen. Diplomarbeit, 2012)



```
class SpecialSub : public FunctionPass {
    static char ID;
    SpecialSub() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) {
        for (inst_iterator II = inst_begin(F), IE = inst_end(F);
            II != IE; ++II)
            performSpecialSubOptimization(&*II);
        return true;
    }
};

static RegisterPass<SpecialSub> X("specialsub",
    "Special_subtraction_transformation_pattern");
```





```
class SpecialSub : public FunctionPass { ←———— erben
    static char ID;
    SpecialSub() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) {
        for (inst_iterator II = inst_begin(F), IE = inst_end(F);
            II != IE; ++II)
            performSpecialSubOptimization(&*II);
        return true;
    }
};

static RegisterPass<SpecialSub> X("specialsub",
    "Special_subtraction_transformation_pattern");
```

# Optimierungen

## Pass-Implementierung



```
class SpecialSub : public FunctionPass { ←———— erben
```

```
    static char ID;  
    SpecialSub() : FunctionPass(ID) {}
```

```
    bool runOnFunction(Function &F) { ←———— implementieren
```

```
        for (inst_iterator II = inst_begin(F), IE = inst_end(F);  
            II != IE; ++II)  
            performSpecialSubOptimization(&*II);  
        return true;
```

```
    }  
};
```

```
static RegisterPass<SpecialSub> X("specialsub",  
    "Special_subtraction_transformation_pattern");
```

# Optimierungen

## Pass-Implementierung

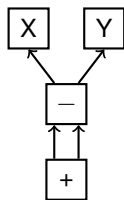
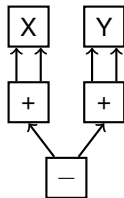


```
class SpecialSub : public FunctionPass { ← erben
    static char ID;
    SpecialSub() : FunctionPass(ID) {}
```

```
    bool runOnFunction(Function &F) { ← implementieren
        for (inst_iterator II = inst_begin(F), IE = inst_end(F);
            II != IE; ++II)
            performSpecialSubOptimization(&*II);
        return true;
    }
};
```

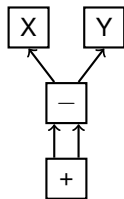
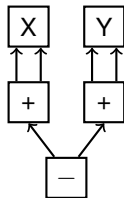
```
static RegisterPass<SpecialSub> X("specialsub",
    "Special_subtraction_transformation_pattern"); ← registrieren
```

```
void performSpecialSubOptimization(Instruction *I) {  
    if (match(I, m_Sub(  
        m_Add(m_Value(X1), m_Value(X2)),  
        m_Add(m_Value(Y1), m_Value(Y2))  
    )) && X1 == X2 && Y1 == Y2) {  
  
        newSub = CreateSub(X1, Y1, "newsub");  
        newAdd = CreateAdd(newSub, newSub, "newadd");  
  
        newSub->insertBefore(I);  
        newAdd->insertBefore(I);  
  
        I->replaceAllUsesWith(newAdd);  
    }  
}
```



```
void performSpecialSubOptimization(Instruction *I) {  
    if (match(I, m_Sub(  
        m_Add(m_Value(X1), m_Value(X2)),  
        m_Add(m_Value(Y1), m_Value(Y2))  
    )) && X1 == X2 && Y1 == Y2) {  
        newSub = CreateSub(X1, Y1, "newsub");  
        newAdd = CreateAdd(newSub, newSub, "newadd");  
  
        newSub->insertBefore(I);  
        newAdd->insertBefore(I);  
  
        I->replaceAllUsesWith(newAdd);  
    }  
}
```

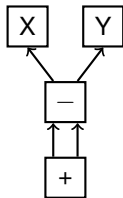
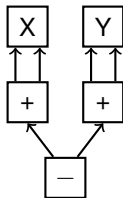
← Muster finden



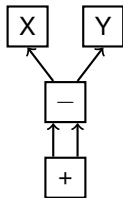
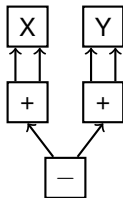
```
void performSpecialSubOptimization(Instruction *I) {  
    if (match(I, m_Sub(  
        m_Add(m_Value(X1), m_Value(X2)),  
        m_Add(m_Value(Y1), m_Value(Y2))  
    )) && X1 == X2 && Y1 == Y2) {  
        newSub = CreateSub(X1, Y1, "newsub");  
        newAdd = CreateAdd(newSub, newSub, "newadd");  
        newSub->insertBefore(I);  
        newAdd->insertBefore(I);  
        I->replaceAllUsesWith(newAdd);  
    }  
}
```

← Muster finden

↑ erzeugen



```
void performSpecialSubOptimization(Instruction *I) {  
    if (match(I, m_Sub(  
        m_Add(m_Value(X1), m_Value(X2)),  
        m_Add(m_Value(Y1), m_Value(Y2))  
    )) && X1 == X2 && Y1 == Y2) { ← Muster finden  
  
        newSub = CreateSub(X1, Y1, "newsub");  
        newAdd = CreateAdd(newSub, newSub, "newadd");  
  
        newSub->insertBefore(I);  
        newAdd->insertBefore(I); ← erzeugen  
        ← einfügen  
  
        I->replaceAllUsesWith(newAdd);  
    }  
}
```



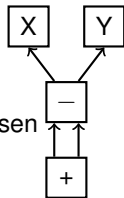
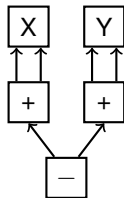
```
void performSpecialSubOptimization(Instruction *I) {  
    if (match(I, m_Sub(  
        m_Add(m_Value(X1), m_Value(X2)),  
        m_Add(m_Value(Y1), m_Value(Y2))  
    )) && X1 == X2 && Y1 == Y2) {  
        newSub = CreateSub(X1, Y1, "newsub");  
        newAdd = CreateAdd(newSub, newSub, "newadd");  
        newSub->insertBefore(I);  
        newAdd->insertBefore(I);  
        I->replaceAllUsesWith(newAdd);  
    }  
}
```

← Muster finden

↑ erzeugen

← einfügen

← Verwender anpassen





# Optimierungen

## Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

# Optimierungen

## Beispiel



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

```
%add = add i32 %a, %a  
%add1 = add i32 %b, %b  
%sub = sub i32 %add, %add1  
ret i32 %sub
```

# Optimierungen

## Beispiel



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

```
%add = add i32 %a, %a  
%add1 = add i32 %b, %b  
%sub = sub i32 %add, %add1  
ret i32 %sub
```

```
↓ opt -load SpecialSub.so -specialsub -dce prog.ll ↓
```

# Optimierungen

## Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

```
%add = add i32 %a, %a  
%add1 = add i32 %b, %b  
%sub = sub i32 %add, %add1  
ret i32 %sub
```

```
↓ opt -load SpecialSub.so -specialsub -dce prog.ll ↓
```

```
%newsub = sub i32 %a, %b  
%newadd = add i32 %newsub, %newsub  
ret i32 %newadd
```

Optimierungen können Analyseinformationen anfordern:

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const {  
    AU.addRequired<AliasAnalysis>();  
}
```

Verwendung:

```
...  
AliasAnalysis &AA = getAnalysis<AliasAnalysis>();  
if (AA.alias(V1, V2)) {  
    ...  
}
```

- ▶ Einheitliche Schnittstelle für ...
  - ▶ Alias-Anfragen:  $(Addr, Size) \times (Addr, Size) \rightarrow \{No, May, Must, Partial\}Alias$
  - ▶ ModRef-Anfragen:  $(Instr, Addr, Size) \rightarrow \{No, Mod, Ref, ModRef\}$
  - ▶ `doesNotAccessMemory`, `onlyReadsMemory` für Funktionen
- ▶ Konvention: Analysen werden verkettet, deswegen auch sehr spezielle Verfahren möglich.
- ▶ Bequemere Verwendung der Information:
  - ▶ Memory Dependence-Analyse: findet Abhängigkeiten zwischen zustandsverändernden Instruktionen.
  - ▶ AliasSets: Einteilung von Zeigern in disjunkte Mengen.

**basic** Analyse basierend auf Kenntnis von Fakten,  
z.B. *“globale Variablen und allozierter Speicher auf Stack und Heap  
aliasen nicht und sind nie `NULL`”*.

**steens** Steensgard Points-to-Analyse.

**ds** Data Structure Analysis (von den LLVM-Autoren).

**scev** Einbeziehung der Scalar-Evolution-Information, d.h. Entwicklung  
von Werten in Schleifen.

**globalsmodref** Spezielle Analyse für globale Variablen, deren Adressen nicht  
ausgewertet werden.

## In LLVM gibt es u.a.:

- ▶ Inlining
- ▶ Interprozedurale Konstantenpropagation (einfach und Sparse Conditional Constant Propagation)

## Als Beispiel: IPConstantPropagation

- ▶ Wenn ein Funktionsargument bei allen Aufrufen im Programm konstant ist, ersetze es in der Funktion.
- ▶ Wenn der Rückgabewert konstant ist, verwende ihn bei allen Aufrufen direkt.





### ► Wie findet man alle Aufrufe einer Funktion?

```
for (Value::use_iterator UI = F.use_begin(), E = F.use_end(); UI != E; ++UI) {  
    User *U = *UI;
```

### ► Wie iteriert man über die Argumente?

```
CallSite CS(cast<Instruction>(U));  
// Check out all of the potentially constant arguments.  
CallSite::arg_iterator AI = CS.arg_begin();  
Function::arg_iterator Arg = F.arg_begin();  
for (unsigned i = 0, e = ArgumentConstants.size(); i! e; ++i, ++AI, ++Arg) {  
  
    // If this argument is known non-constant, ignore it.  
    if (ArgumentConstants[i].second) continue;
```



▶ Ist das Argument eine Konstante?

```
Constant *C = dyn_cast<Constant>(*AI);  
if (C && ArgumentConstants[i].first == 0) {  
    ArgumentConstants[i].first = C;    // First constant seen.
```

▶ Sind zwei Konstanten gleich?

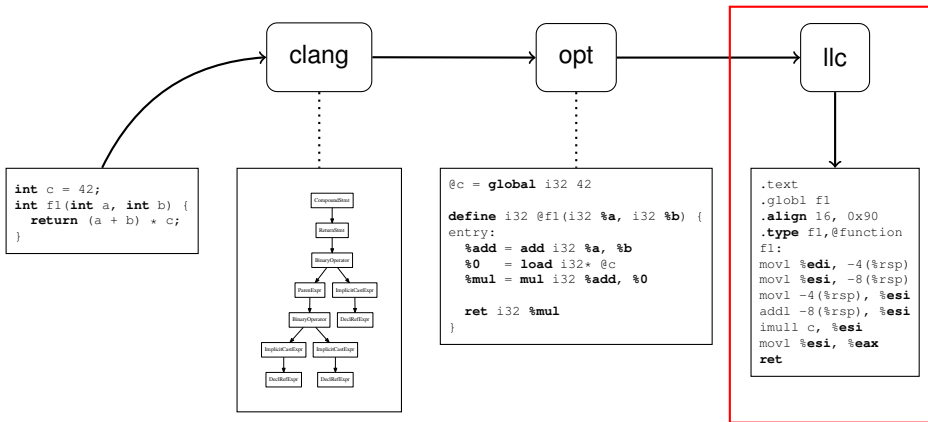
```
    } else if (C && ArgumentConstants[i].first == C) {  
        // Still the constant value we think it is.  
    } else {  
        // Argument became non-constant. If all arguments are non-constant now,  
        // give up on this function.  
        if (++NumNonconstant == ArgumentConstants.size()) return false;  
        ArgumentConstants[i].second = true;  
    }  
}  
}
```



- ▶ Wie ersetzt man das Argument durch die gefundene Konstante?

```
Function::arg_iterator AI = F.arg_begin();
for (unsigned i = 0, e = ArgumentConstants.size(); i != e; ++i, ++AI) {
    // Do we have a constant argument?
    if (ArgumentConstants[i].second || AI->use_empty() ||
        (AI->hasByValAttr() && !F.onlyReadsMemory()))
        continue;

    Value *V = ArgumentConstants[i].first;
    if (V == 0) V = UndefValue::get(AI->getType());
    AI->replaceAllUsesWith(V);
}
```



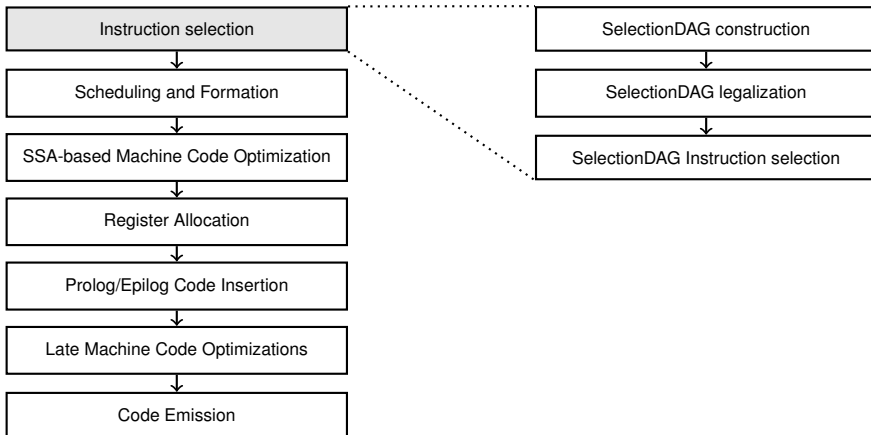
- ▶ Verfügbare Targets im Backend: x86, ARM, PowerPC, SPARC, MIPS, ...
- ▶ Im Großen und Ganzen sind für diese Architekturen die selben Aufgaben zu bewältigen.
- ▶ In LLVM:



- ▶ Targets außerhalb dieses Schemas sind natürlich auch möglich (z.B. C++-Backend, Verilog)

# Codeerzeugung

## Instruction Selection





# Instruction Selection

## SelectionDAG Construction

- ▶ SelectionDAG ist ein gerichteter, azyklischer Graph pro BasicBlock.
  - ▶ Knoten: Operationen (hauptsächlich Opcode und Operanden).
  - ▶ Kanten: Daten (Integer, Float, ...) und “Chain”-Kanten (Ordnung zwischen Operationen)
  - ▶ Konzept des “legalen/illegalen Graphen”: Graph darf am Anfang Knoten enthalten, die das Target nicht direkt auf Instruktionen abbilden kann.
- ▶ Aufbau: **SelectionDAGBuilder**
  - ▶ Prinzip: Visitor über LLVM-Instruktionen, erzeugt **SDNodes**.
  - ▶ Übersetzung einfacher Instruktionen festverdrahtet, das Lowering für andere Instruktionen ist Target-spezifisch (später mehr dazu).

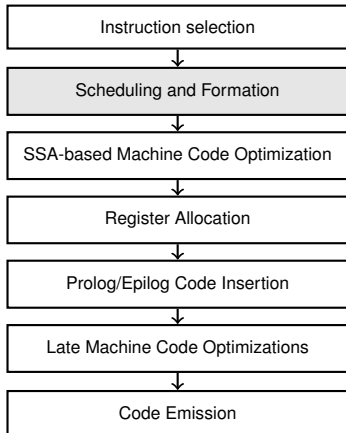






# Codeerzeugung

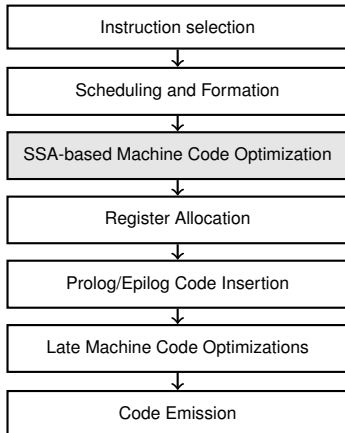
## Scheduling and Formation



- ▶ Totalordnung der Instruktionen wird berechnet (unter Berücksichtigung der Abhängigkeitskanten).
- ▶ Die Knoten werden durch **MachineInstr** ersetzt.
  - ▶ Klasse ist abstrakt: nur Opcode, Operanden. (Semantik Target-spezifisch!)
  - ▶ **MachineInstr** bilden **MachineBasicBlocks**, aus denen eine **MachineFunction** besteht.
- ▶ Darstellung erstmal weiterhin in SSA-Form!

# Codeerzeugung

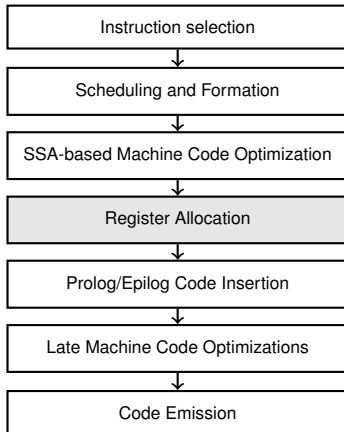
## SSA-based Machine Code Optimization



- ▶ OptimizePHIs
  - ▶ Eliminiert tote Phi-Zyklen.
- ▶ Dead Machine Instruction Elimination
- ▶ Machine LICM
  - ▶ Verschieben von schleifeninvariantem Code aus Schleifen heraus.
- ▶ Machine CSE
- ▶ Machine Sinking
  - ▶ verschiebt Instruktion soweit wie möglich in Nachfolgerblöcke.
- ▶ Peephole-Optimierungen

# Codeerzeugung

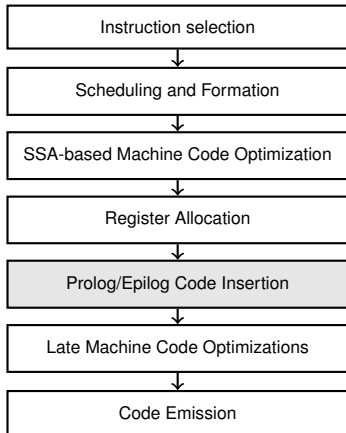
## Register Allocation



- ▶ Virtuelle Register werden auf physikalische Register abgebildet.
- ▶ Mehrere Heuristiken auswählbar.
- ▶ SSA deconstruction: Phi-Instruktionen müssen ersetzt werden.  
Hier: Einfügen von Kopieranweisungen.

# Codeerzeugung

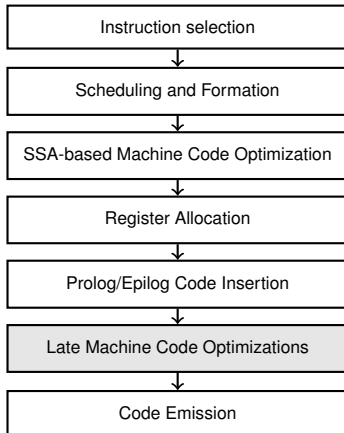
## Prolog/Epilog Code Insertion



- ▶ Nach der Registerallokation ist die Größe aller Stacks bekannt  
→ Prolog beim Funktionseintritt und Epilog beim Verlassen können eingefügt werden.
- ▶ Außerdem: Informationen für das Stack-Unwindung (Exception Handling) werden erzeugt.

# Codeerzeugung

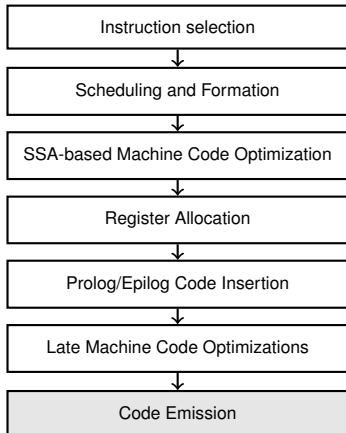
## Late Machine Code Optimizations



- ▶ Zusammenfassen von BasicBlocks, die mit unbedingten Sprüngen enden.
- ▶ Copy Propagation

# Codeerzeugung

## Code Emission

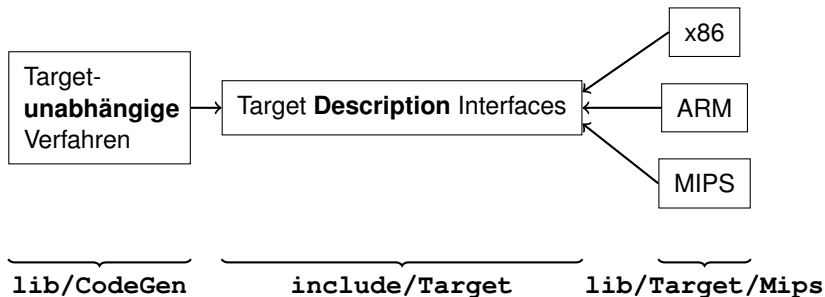


- ▶ Transformation der Darstellung auf das Abstraktionsniveau des System-Assemblers
  - ▶ Labels, Direktiven, Instruktionen
- ▶ Grundgerüst für die Umwandlung: **AsmPrinter**.
- ▶ Wichtiges Hilfsmittel: **MCStreamer** hat Methoden wie **EmitLabel(..)**, **EmitValue(..)**, **SwitchSection(..)**



# Target-Definition

## Übersicht



## Ab nun: Target-Definition, am Beispiel der Mips-Architektur

- ▶ Zentraler Einstiegspunkt ist das Interface **TargetMachine**.  
Von dort aus Zugriff auf die folgenden Interfaces (u.a.):
  - ▶ **DataLayout**
  - ▶ **TargetRegisterInfo**
  - ▶ **TargetInstrInfo**
  - ▶ **TargetLowering**
- ▶ Um die (target-unabhängige) Infrastruktur im Codegenerator zu nutzen, muss ein Target diese Interfaces implementieren.

- ▶ Werkzeug und Sprache, um Menge von abstrakt Datensätzen (records) zu beschreiben.
  - ▶ Bedeutung erhalten sie erst durch verschiedene (TableGen-) Backends.
  - ▶ Oberstes Ziel: Möglichst viele Gemeinsamkeiten der Datensätze ausfaktorisieren.
- ▶ Ein konkreter Datensatz wird mit `def` eingeleitet.
- ▶ Klassen `class` dienen als Schablonen für `defs`.

```
class Foo<int i, string s > {
    bit b = 1;
    string msg = s#" "#i;
}

def aFoo : Foo<1, "Hello World">;
def bFoo : Foo<42, "Don't panic">;
let b = 0 in
    def notFoo : Foo<99, "Override">;
----- liefert -----
def aFoo { // Foo
    bit b = 1;
    string msg = "Hello World 1";
}
def bFoo { // Foo
    bit b = 1;
    string msg = "Don't panic 42";
}
def notFoo { // Foo
    bit b = 0;
    string msg = "Override 99";
}
```



- ▶ LLVM ist eine Sammlung von Bibliotheken und Werkzeugen zur Compilerentwicklung.
- ▶ LLVM-IR  $\approx$  Assembler für virtuelle Maschine mit typischen RISC-Befehlssatz plus ein paar Extras.
- ▶ Beim Design der Infrastruktur wurde größter Wert auf Modularität und Wiederverwendbarkeit der Komponenten gelegt.

- ▶ <http://llvm.org>
- ▶ <http://llvm.org/docs/LangRef.html>
- ▶ <http://llvm.org/docs/ProgrammersManual.html>
- ▶ <http://llvm.org/docs/WritingAnLLVMPass.html>
- ▶ <http://llvm.org/docs/AliasAnalysis.html>
- ▶ <http://llvm.org/docs/WritingAnLLVMBackend.html>
- ▶ <http://llvm.org/docs/CodeGenerator.html>
- ▶ <http://www.aosabook.org/en/llvm.html>
- ▶ Chris Lattner und Vikram Adve: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation".
- ▶ Christoph Erhardt: "Design and Implementation of a TriCore Backend for the LLVM Compiler Framework".



# Backup-Folien



- ▶ C-Programm nach LLVM übersetzen:  
\$ **clang** -S -emit-llvm -o prog.ll prog.c
- ▶ IR in “richtige” SSA-Form bringen:  
\$ **opt** -S -mem2reg -o prog-ssa.ll prog.ll
- ▶ Zwischen Assembler- und Bitcode-Format konvertieren:  
\$ **llvm-as** prog.ll # erzeugt prog.bc  
\$ **llvm-dis** prog.bc # erzeugt prog.ll
- ▶ Codeerzeugung (= System-Assemblercode generieren)  
\$ **llc** prog.ll # erzeugt prog.s

- ▶ Codequalität: kein klarer Sieger, leichter Vorteil für gcc 4.8 ggü. clang 3.2
- ▶ gcc unterstützt mehr Sprachen und Architekturen
- ▶ clang ist modularer, schneller und braucht weniger Speicher
- ▶ Linux Kernel: muss noch gepatcht werden, ARM besser unterstützt als x86.

[http://www.phoronix.com/scan.php?page=article&item=llvm\\_clang32\\_final](http://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final)

<http://clang.llvm.org/comparison.html>

[http://llvm.linuxfoundation.org/index.php/Main\\_Page](http://llvm.linuxfoundation.org/index.php/Main_Page)



# Target-Definition

## Endianness, Datentypen und Alignment

- ▶ Interface: **DataLayout**
- ▶ Wird nicht abgeleitet, sondern instantiiert.
- ▶ Beispiel (Mips, 32 bit, little endian):

```
...  
DL("e-p:32:32:32-i8:8:32-i16:16:32-i64:64:64-n32-S64");  
...
```

- ▶ Bedeutet:
  - ▶ e: little endian
  - ▶ p:32:32:32: Zeiger sind 32 bit breit. Das in der ABI definierte Alignment ist 32 bits, das bevorzugte Alignment ebenfalls.
  - ▶ i8:8:32: 8 bit-Werte müssen 8 bit-aligned sein (ABI), bevorzugt ist aber die Ausrichtung auf 32 bit.
  - ▶ n32: Nativ werden nur 32 bit-Werte unterstützt.
  - ▶ S64: Der Stack ist 64 bit-aligned.



- ▶ Interface ist **TargetRegisterInfo**.
  - ▶ Welche Register und Registerklassen gibt es?  
Registerklasse: Menge von funktional gleichwertigen Registern.
  - ▶ Welche Beschränkungen gelten jeweils?
- ▶ Das RegisterInfo-Backend findet u.a. alle Records der vordefinierten (TableGen-)Klasse **Register** in **MipsRegisterInfo.td**.
- ▶ Daraus generiert es (hauptsächlich) enums und Datenarrays in die Klasse **MipsGenRegisterInfo : TargetRegisterInfo**.
- ▶ Was sich dort nicht ausdrücken lässt, wird in einer Unterklasse in C++ formuliert: **MipsRegisterInfo : MipsGenRegisterInfo**

# Target-Definition

## Registersatz (Beispiel)



## MipsRegisterInfo.td

```
class MipsReg<bits<16> Enc, string n> : Register<n> {
  let HWEncoding = Enc;
  let Namespace = "Mips";
}

class MipsGPRReg<bits<16> Enc, string n> : MipsReg<Enc, n>;

...
// General Purpose Registers
def ZERO : MipsGPRReg< 0, "zero">, DwarfRegNum<[0]>;
def AT   : MipsGPRReg< 1, "1">, DwarfRegNum<[1]>;
def V0   : MipsGPRReg< 2, "2">, DwarfRegNum<[2]>;
def V1   : MipsGPRReg< 3, "3">, DwarfRegNum<[3]>;
def A0   : MipsGPRReg< 4, "4">, DwarfRegNum<[4]>;
def A1   : MipsGPRReg< 5, "5">, DwarfRegNum<[5]>;
def A2   : MipsGPRReg< 6, "6">, DwarfRegNum<[6]>;
...
```

# Target-Definition

## Registersatz (Beispiel)



## MipsRegisterInfo.td

```
class CPURegsClass<list<ValueType> regTypes> :
  RegisterClass<"Mips", regTypes, 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1, A2, A3,
  // Not preserved across procedure calls
  T0, T1, T2, T3, T4, T5, T6, T7,
  // Callee save
  S0, S1, S2, S3, S4, S5, S6, S7,
  // Not preserved across procedure calls
  T8, T9,
  // Reserved
  K0, K1, GP, SP, FP, RA)>;

def CPURegs : CPURegsClass<[i32]>;
def DSPRegs : CPURegsClass<[v4i8, v2i16]>;
...
```

# Target-Definition

## Registersatz (Beispiel)

### MipsRegisterInfo.cpp

```
MipsRegisterInfo::getCalleeSavedRegs(const MachineFunction *MF);  
MipsRegisterInfo::getCallPreservedMask(CallingConv::ID);  
// liefern generiertes Array zurueck
```

```
BitVector MipsRegisterInfo::  
getReservedRegs(const MachineFunction &MF) const {  
    static const uint16_t ReservedCPURegs[] = {  
        Mips::ZERO, Mips::K0, Mips::K1, Mips::SP  
    };  
    ...  
    return ReservedCPURegs + ... // pseudo-code  
}
```

- ▶ Interface: **TargetInstrInfo**
- ▶ Befehlsformate und verfügbare Instruktionen liegen als TableGen-Beschreibung vor.
- ▶ Daraus wird generiert (für Mips):
  - ▶ **MipsGenInstrInfo** : **TargetInstrInfo**: eine Klasse voller Enums und statischer Arrays.  
Diese wird noch durch C++-Code in **MipsInstrInfo** erweitert.
  - ▶ Target-spezifische Teile des Pattern Matchings für die Befehlsauswahl:  
**MipsGenDAGISel.inc** wird generiert und in **MipsDAGToDAGISel** inkludiert.  
In **MipsDAGToDAGISel::Select(SDNode \*Node)** werden zuerst die Sonderfälle behandelt. Ansonsten wird der generische Matcher mit einer generierten Tabelle aufgerufen.

# Target-Definition

## Befehlssatz (Beispiel)



### MipsInstrInfo.td/MipsInstrFormats.td

```
// Arithmetic and logical instructions with 3 register operands.
class ArithLogicR<string opstr, RegisterOperand RO, bit isComm = 0,
                InstrItinClass Itin = NoItinerary,
                SDPatternOperator OpNode = null_frag>:
  InstSE<(outs RO:$rd), (ins RO:$rs, RO:$rt),
        !strconcat(opstr, "\t$rd, $rs, $rt"),
        [(set RO:$rd, (OpNode RO:$rs, RO:$rt))], Itin, FrmR> {
  let isCommutable = isComm;
  ...
}

class ADD_FM<bits<6> op, bits<6> funct> { // aus MipsInstrFormats
  bits<5> rd; bits<5> rs; bits<5> rt;
  bits<32> Inst;
  let Inst{31-26} = op; let Inst{25-21} = rs; let Inst{20-16} = rt;
  let Inst{15-11} = rd; let Inst{10-6} = 0; let Inst{5-0} = funct;
}

def ADDu : ArithLogicR<"addu", CPURegsOpnd, 1, IIALu, add>, ADD_FM<0, 0x21>;
def SUBu : ArithLogicR<"subu", CPURegsOpnd, 0, IIALu, sub>, ADD_FM<0, 0x23>;
```

# Target-Definition

## Befehlssatz (Beispiel)



## MipsInstrInfo.td

```
// Conditional Branch
class CBranch<string opstr, PatFrag cond_op, RegisterClass RC> :
  InstSE<(outs), (ins RC:$rs, RC:$rt, brtarget:$offset),
    !strconcat(opstr, "\t$rs, $rt, $offset"),
    [(brcond (i32 (cond_op RC:$rs, RC:$rt)), bb:$offset)], IIBranch,
    FrmI> {
  let isBranch = 1;
  let isTerminator = 1;
  let hasDelaySlot = 1;
  let Defs = [AT];
}

def BEQ      : CBranch<"beq", seteq, CPURegs>, BEQ_FM<4>;
def BNE      : CBranch<"bne", setne, CPURegs>, BEQ_FM<5>;
```



- ▶ Steuerung der SelectionDAG-Konstruktion aus LLVM-IR.
  - ▶ Insbesondere: Was unterstützt das Target nativ?
- ▶ Interface: **TargetLowering**
  - ▶ `isIntDivCheap()`, `isJumpExpensive()`, ...
  - ▶ Intern: "action table":  
Unterklassen können dort eintragen, wie nicht unterstützte Operation behandelt werden sollen.
- promote** Kleine Typen in größere Typen konvertieren  
Bsp.: `i16` → `i32`.
- expand** Größere Typen in kleine Typen aufsplitten oder Bibliotheksaufruf nutzen  
Bsp.: `i64` → `i32, i32`.
- custom** Target-spezifische Implementierung von  
**virtual** `SDValue LowerOperation(SDValue Op, ...)`  
wird aufgerufen.

# Target-Definition

## Lowering (Beispiel)



## MipsTargetLowering in MipsSelLowering.cpp

```
MipsTargetLowering::
MipsTargetLowering(MipsTargetMachine &TM) {
    // Mips does not have il type, so use i32 for
    // setcc operations results (slt, sgt, ...).
    setBooleanContents(ZeroOrOneBooleanContent);

    // Set up the register classes
    addRegisterClass(MVT::i32, &Mips::CPURegsRegClass);
    ...

    // Load extended operations for il types must be promoted
    setLoadExtAction(ISD::EXTLOAD, MVT::il, Promote);
    setLoadExtAction(ISD::ZEXTLOAD, MVT::il, Promote);
    setLoadExtAction(ISD::SEXTLOAD, MVT::il, Promote);
    ...

    // Mips Custom Operations
    setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
    setOperationAction(ISD::BlockAddress, MVT::i32, Custom);

    ...
    // Operations not directly supported by Mips.
    setOperationAction(ISD::ROTL, MVT::i32, Expand);
    setOperationAction(ISD::ROTL, MVT::i64, Expand);
    ...
}
```

# Target-Definition

## Was noch zu tun ist..



- ▶ Aufrufkonventionen definieren.
- ▶ Unterklasse von **AsmPrinter** implementieren, um Ausgabe kompatibel mit dem System-Assembler zu machen.
- ▶ Subtargets definieren.
- ▶ Ggf. eigenen Pässe implementieren, die das Programm für besondere Target-Eigenschaften transformieren.
- ▶ ...