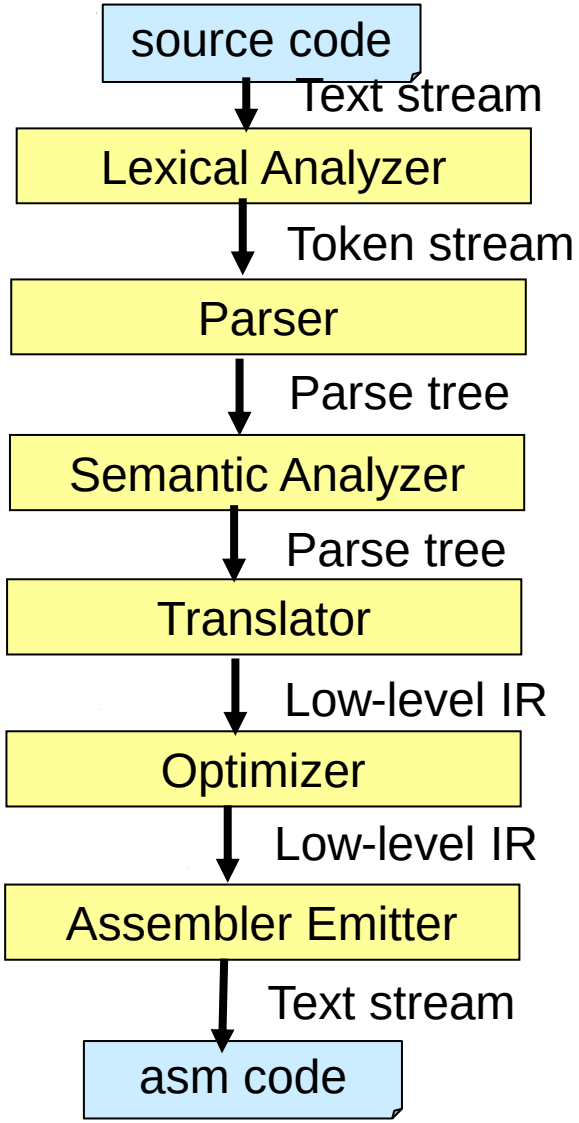
Multi-Level Intermediate Representations

Local CSE, DAGs, Lowering Call Sequences

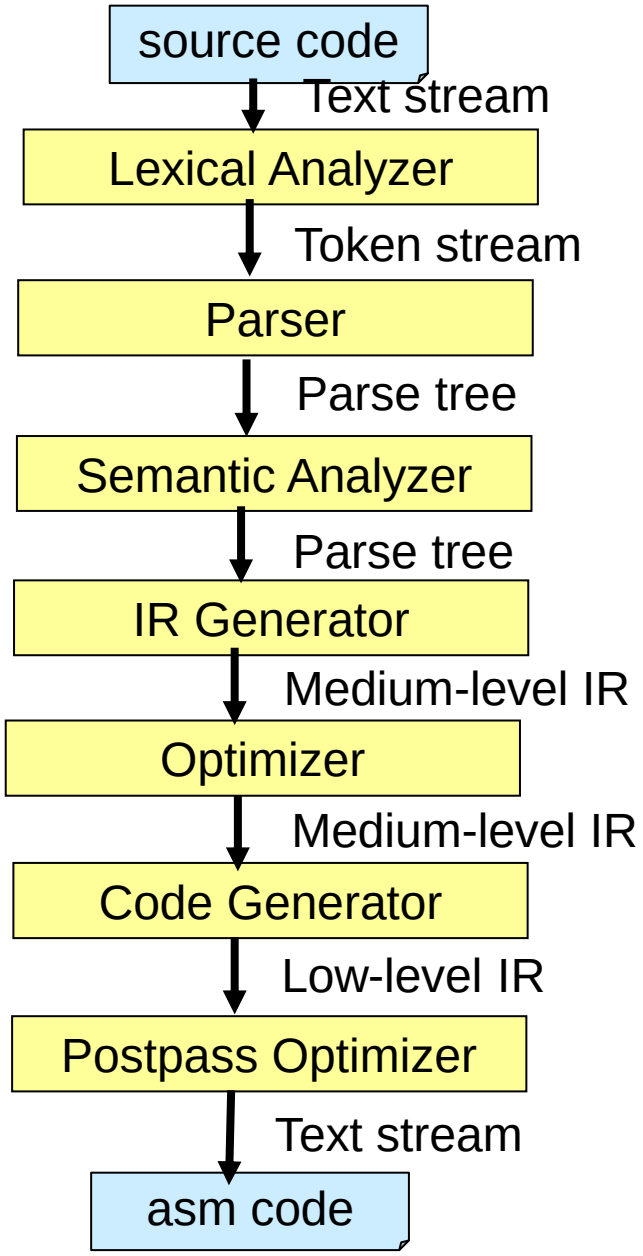
Survey of some Compiler Frameworks

Christoph Kessler, IDA,
Linköping University

Compiler Flow

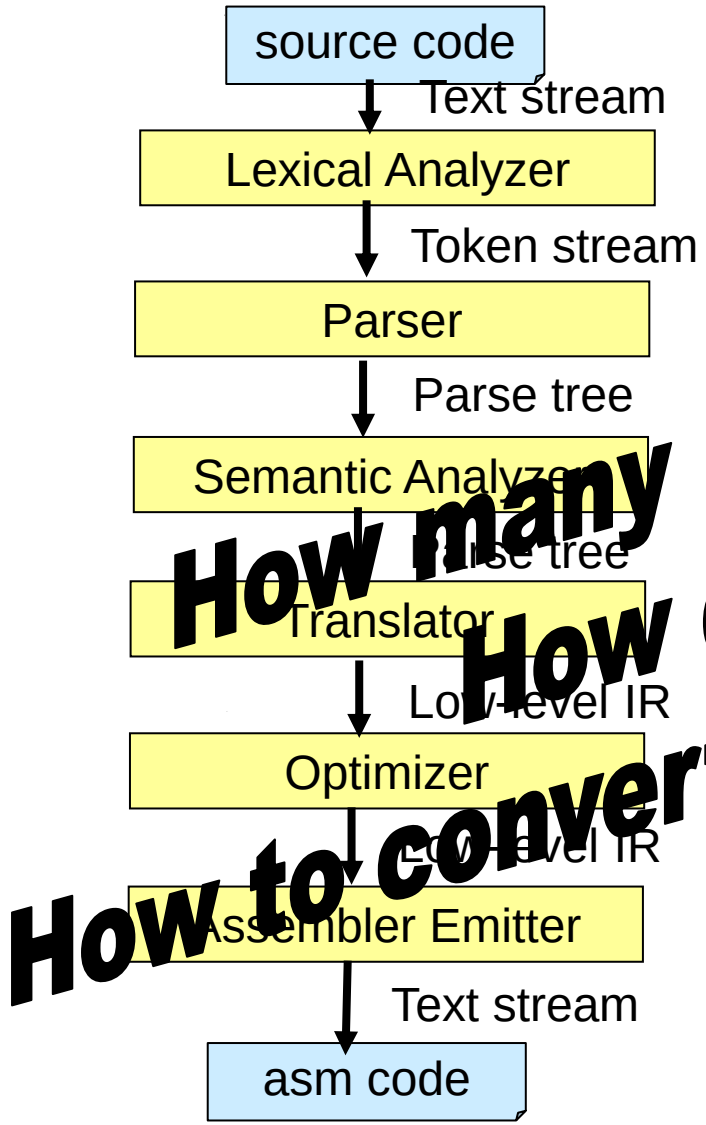


(a) Optimizations on low-level IR only

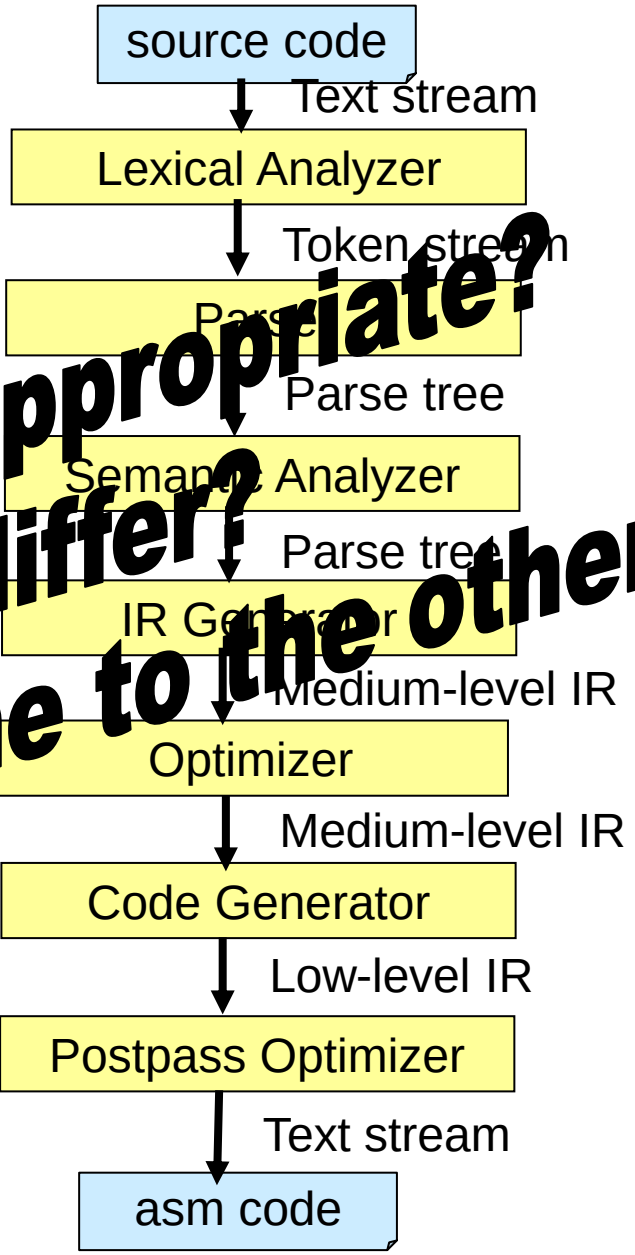


(b) Mixed model

Compiler Flow



(a) Optimizations on low-level IR only



(b) Mixed model

How many IRs are appropriate?

How do they differ?

How to convert from one to the other?

Multi-Level IR

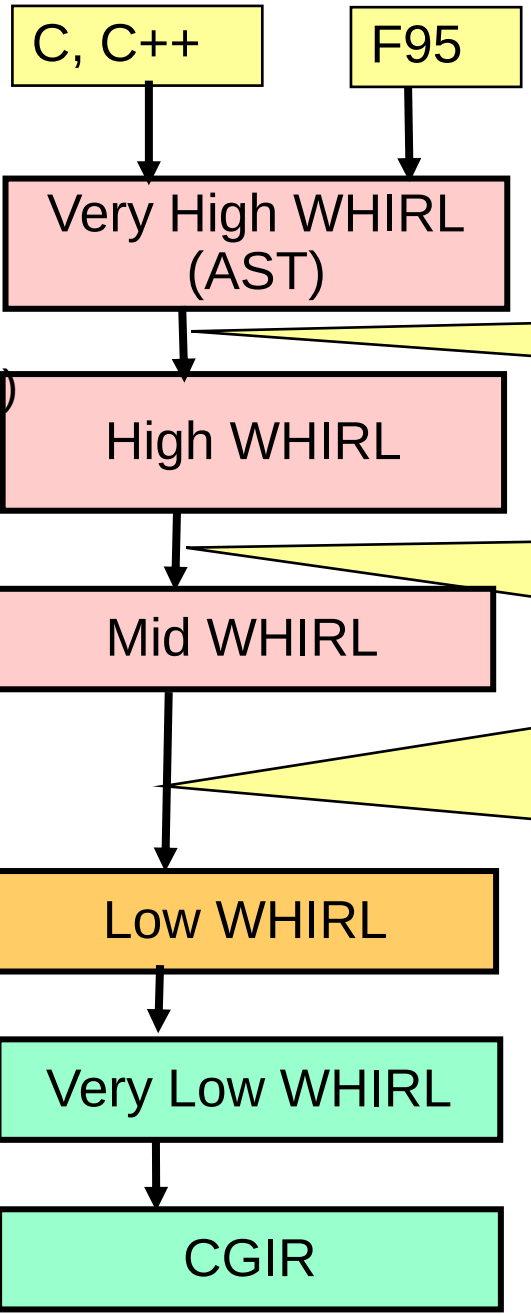
- **Multi-level IR,** e.g.
  - AST abstract syntax tree – implicit control and data flow
  - HIR high-level IR
  - MIR medium-level IR
  - LIR low-level IR, symbolic registers
  - VLIR very low-level IR, target specific, target registers
- Standard form and possibly also SSA (static single assignment) form
- Open form (tree, graph) and/or closed (linearized, flattened) form
  - For expressions: Trees vs DAGs (directed acyclic graphs)
- Translation by **lowering**

☺ Analysis / Optimization engines can work on the most appropriate level of abstraction

☺ Clean separation of compiler phases, somewhat easier to extend and debug

☹ Framework gets larger and slower

Example: WHIRL (Open64 Compiler)



VHO
standalone inliner

IPA (interprocedural analysis)
PREOPT
LNO (Loop nest optimizer)

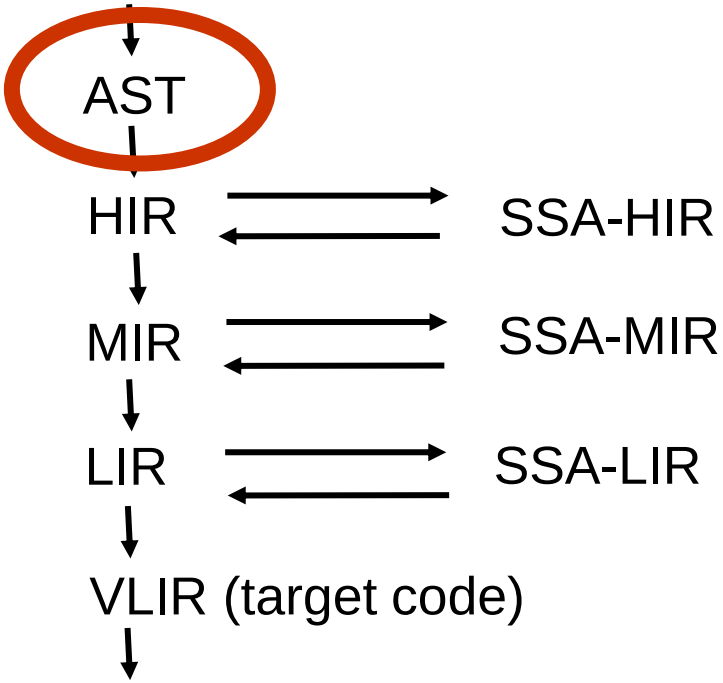
WOPT (global optimizer, uses internally an SSA IR)
RVI1 (register variable identification)

RVI2

CG

CG

Multi-Level IR Overview

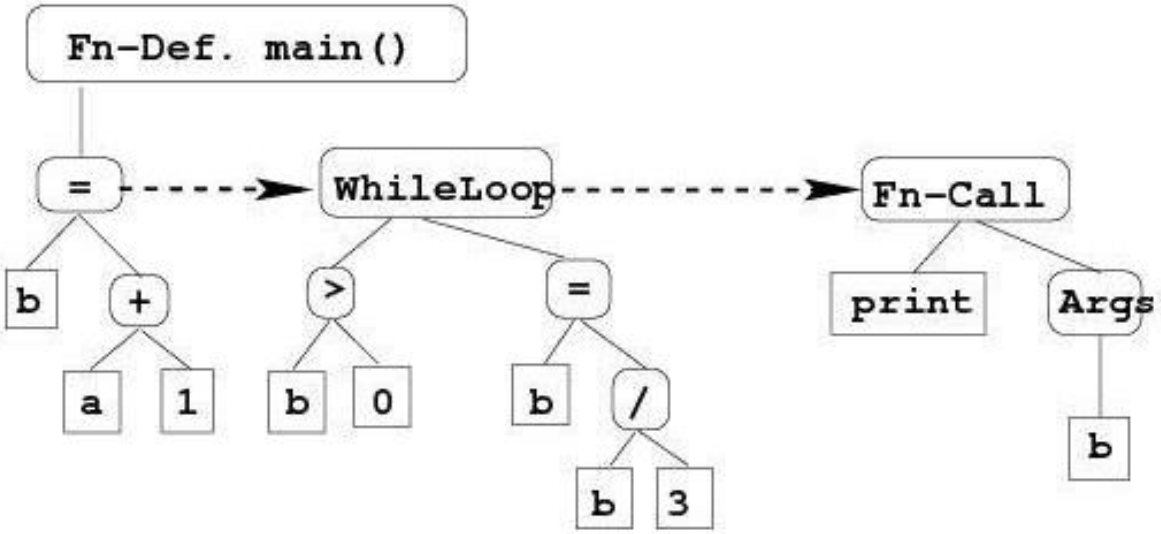


AST, Symbol table

Example program:

```
int a = 4;
extern void print( int );
main()
{ int b = a + 1;
  while ( b > 0 )
       b = b / 3;
  print( b );
}
```

Abstract Syntax Tree:



Symbol table:

globals (Level 0)

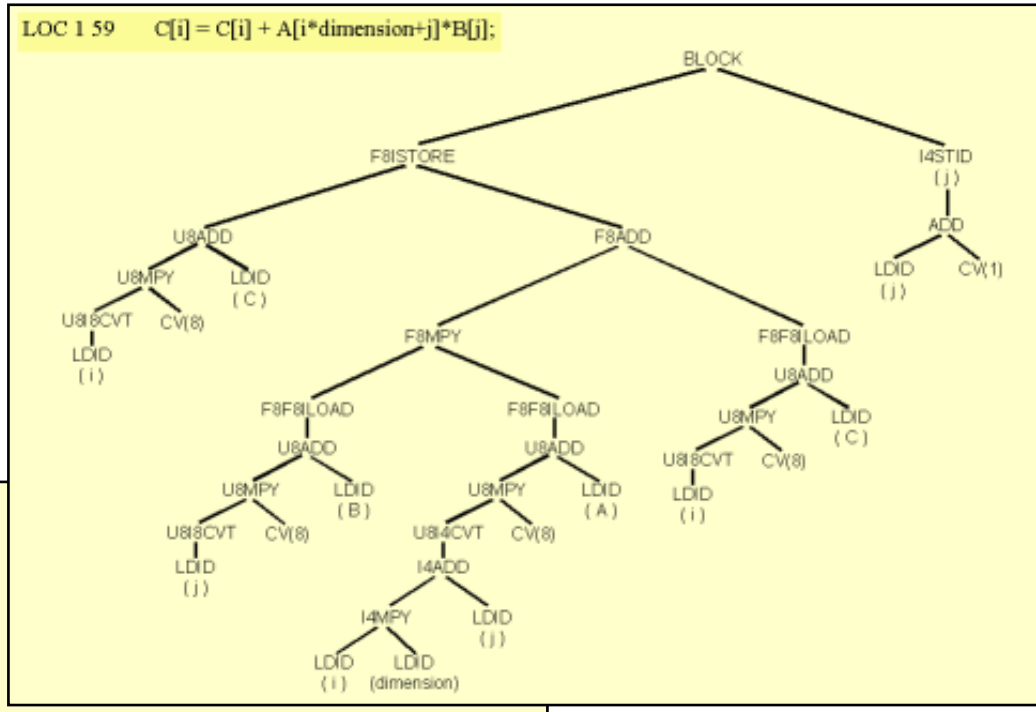
	Name	Type	Value
1	a	int	4
2	print	void	-
3	main	int	●

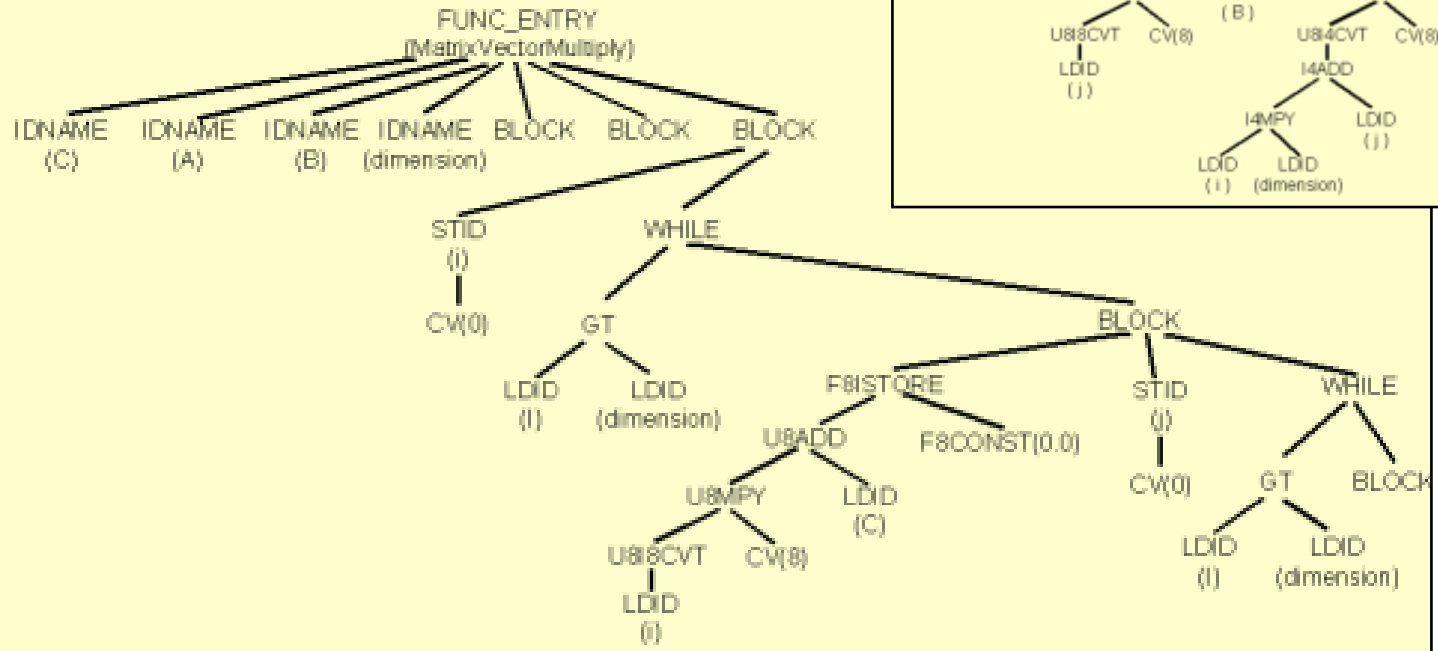
locals, level 1

	Name	Type	Value
1	b	int	-

Hierarchical symbol table follows nesting of scopes

AST Example: Open64 VH-WHIRL





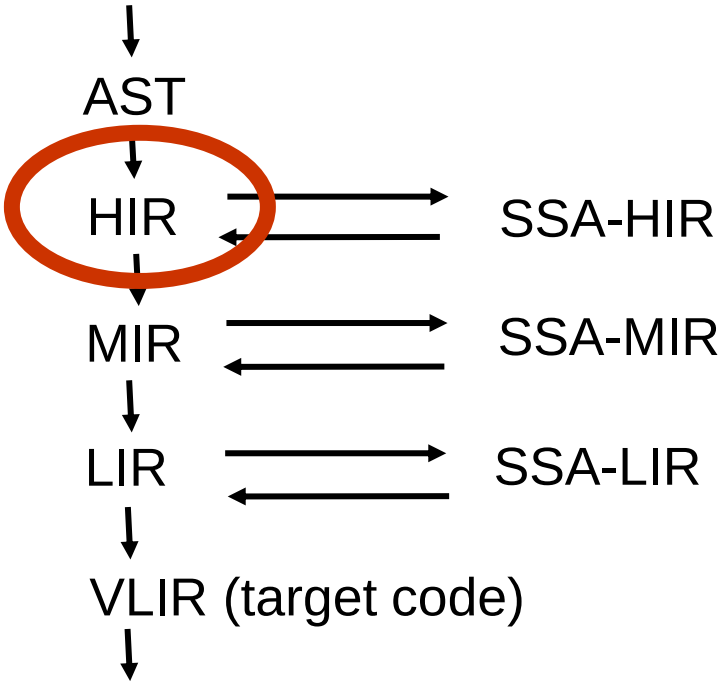
Highest WHIRL Representation.

Symbol table

- Some typical fields in a symbol table entry

Field Name	Field Type	Meaning
name	char *	the symbol's identifier
sclass	enum { STATIC, ...}	storage class
size	int	size in bytes
type	struct type *	source language data type
basetype	struct type *	source-lang. type of elements of a constructed type
machtype	enum { ... }	machine type corresponding to source type (or element type if constructed type)
basereg	char *	base register to compute address
disp	int	displacement to address on stack
reg	char *	name of register containing the symbol's value

Multi-Level IR Overview



HIR - high-level intermediate representation

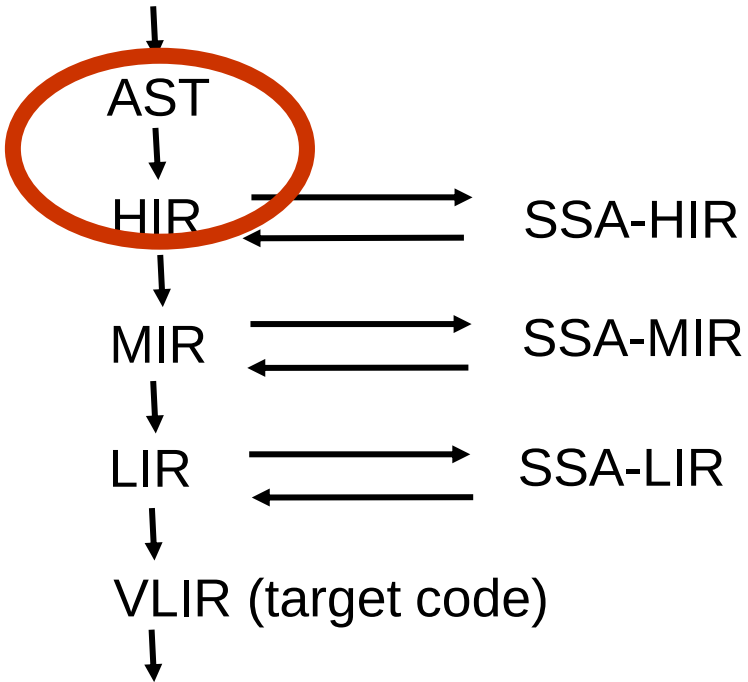
- A (linearized) control flow graph,
  but level of abstraction close to AST
  - loop structures and bounds explicit
  - array subscripts explicit

```
for  v = v1 by v2 to v3 do
     a[i] = 2
endfor
```

→ suitable for data dependence analysis
and loop transformation / parallelization

  - artificial entry node for the procedure
  - assignments var = expr
  - unassigned expressions, e.g. conditionals
  - function calls

Flattening 0:
From AST to HIR (or other CFG repr.)

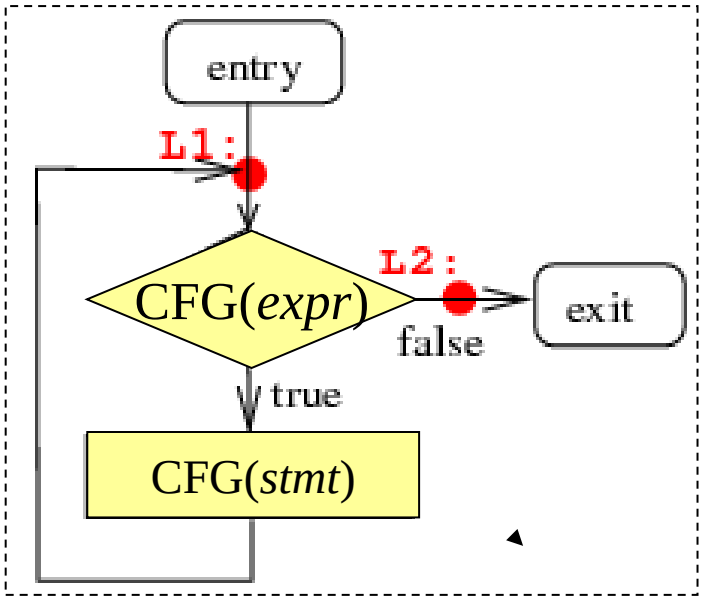


Christoph Kessler, IDA,
Linköping University

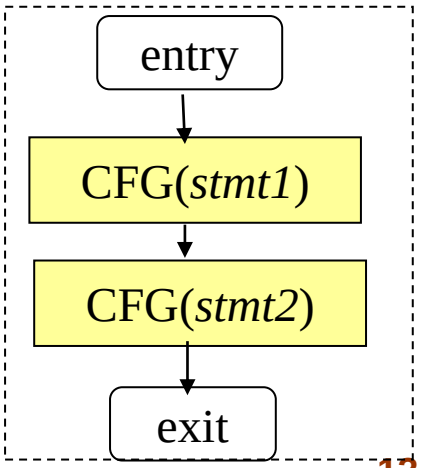
Generating a CFG from AST

- Straightforward for structured programming languages
  - Traverse AST and compose control flow graph recursively
  - As in syntax-directed translation, but separate pass
  - Stitching points: single entry, single exit point of control; symbolic labels for linearization

CFG (**`while`** (*expr*) *stmt*) =



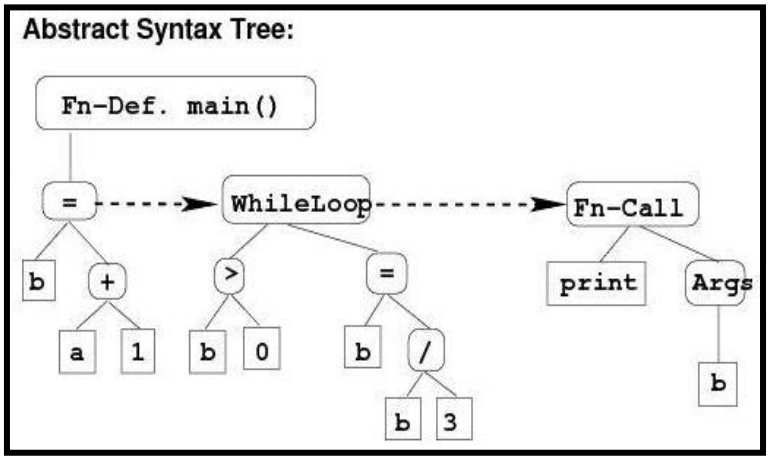
CFG (*stmt1*; *stmt2*) =

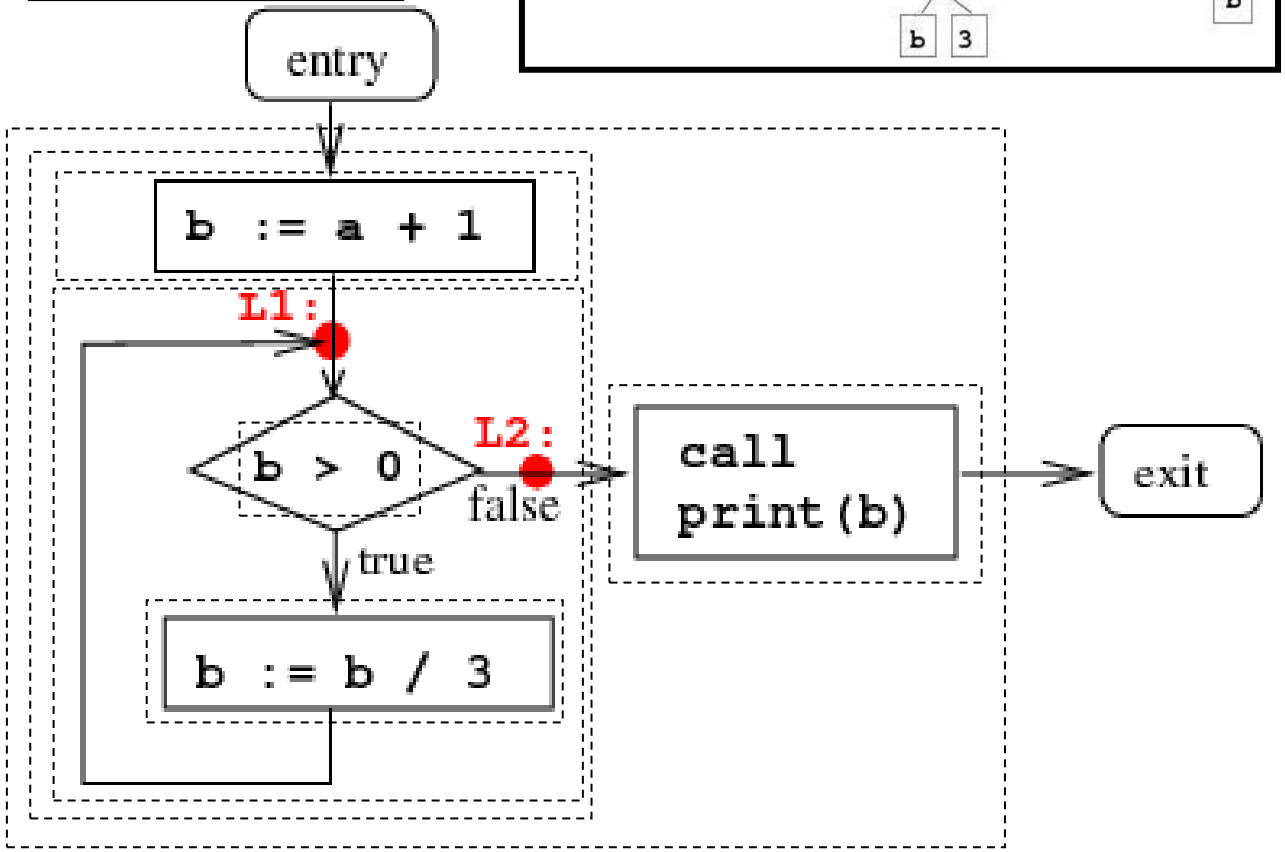


Creating a CFG from AST (2)

- Traverse AST recursively, compose CFG
- Example:

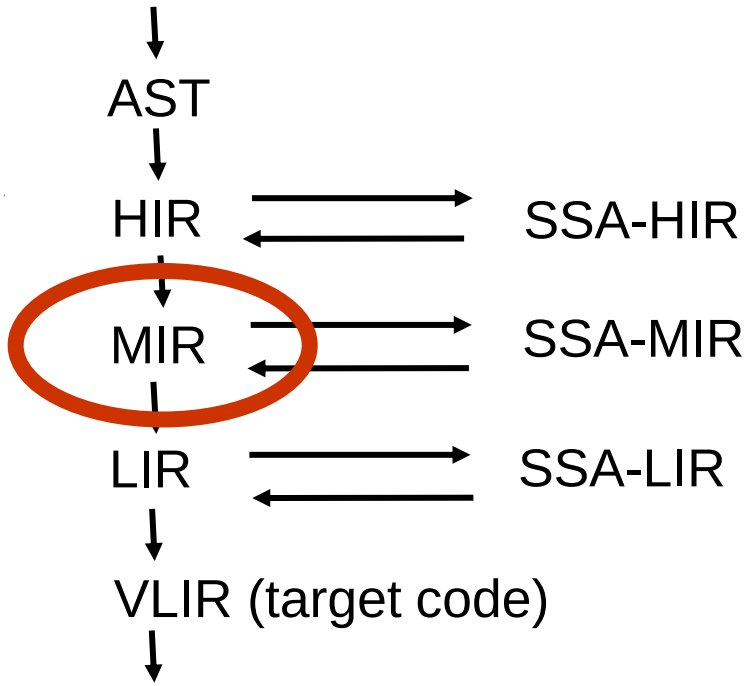
```
{
 b = a + 1;
 while (b>0)
    b = b / 3;
 print(b);
}
```



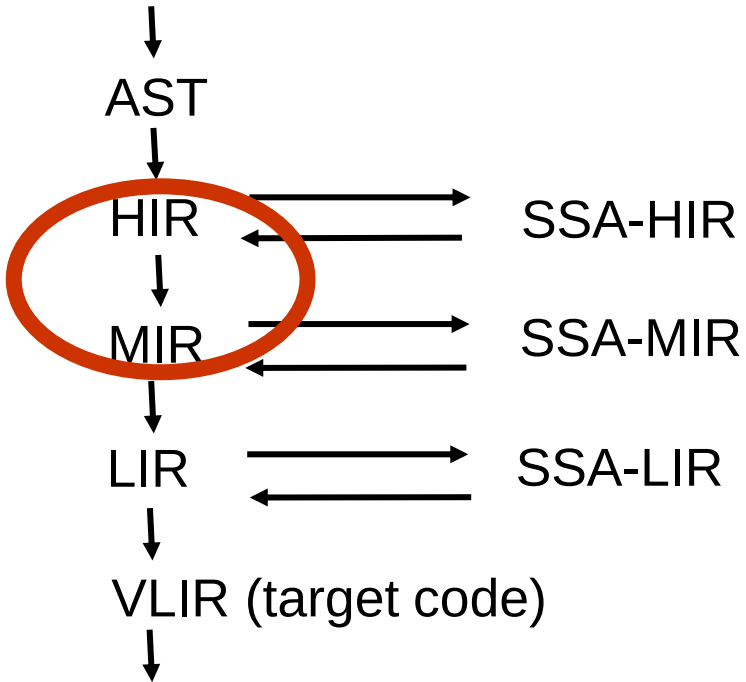


MIR – medium-level intermediate representation

- “language independent”
- control flow reduced to simple branches, call, return
- variable accesses still in terms of symbol table names
- explicit code for procedure / block entry / exit
- suitable for most optimizations
- basis for code generation



Flattening 1: From HIR to MIR



Christoph Kessler, IDA,
Linköping University

HIR→MIR (1): Flattening the expressions

By a **postorder traversal of each expression tree** in the CFG:

- Decompose the nodes of the expression trees (operators, ...) into simple operations (ADD, SUB, MUL, ...)
- Infer the types of operands and results (language semantics)
  - annotate each operation by its (result) type
  - insert explicit conversion operations where necessary
- Flatten each expression tree (= partial order of evaluation) to a sequence of operations (= total order of evaluation) using temporary variables t1, t2, ... to keep track of data flow
  - This is static scheduling!
    May have an impact on space / time requirements

HIR→MIR (2): Lowering Array References (1)

- HIR:
  t1 = a [i, j+2]

- the Lvalue of a [i, j+2] is
  (on a 32-bit architecture)

  (addr a) + 4 * (i * 20 + j + 2)

- MIR:
  t1 = j + 2
  t2 = i * 20
  t3 = t1 + t2
  t4 = 4 * t3
  t5 = addr a
  t6 = t5 + t4
  t7 = *t6

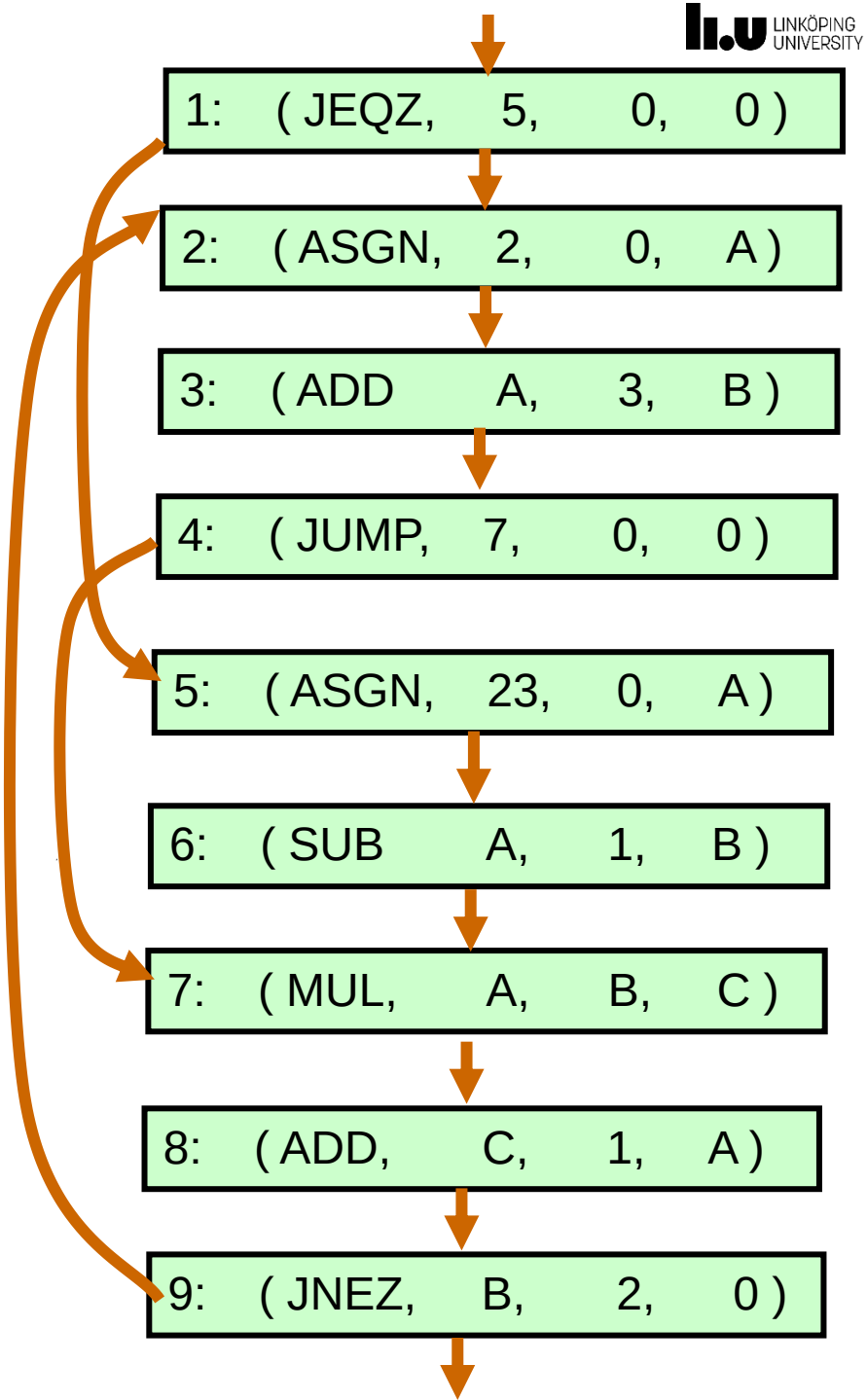
HIR→MIR (2): Flattening the control flow graph

- Depth-first search of the control flow graph
- Topological ordering of the operations, starting with *entry* node
  - at conditional branches:
    one exit fall-through, other exit branch to a label
- **Basic blocks** = maximum-length subsequences of statements containing no branch nor join of control flow
- **Basic block graph** obtained from CFG by merging statements in a basic block to a single node

Control flow graph

- Nodes: primitive operations (e.g., quadruples)
- Edges: control flow transitions
- Example:

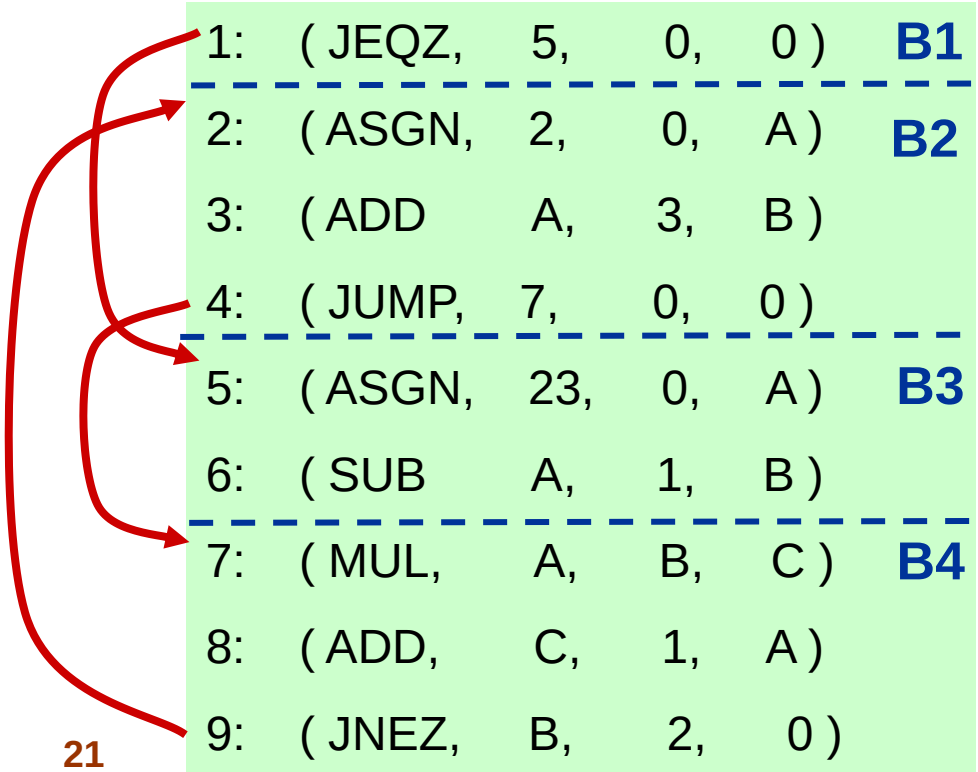
```
1:  ( JEQZ,   5,   0,   0 )
2:  ( ASGN,   2,   0,   A )
3:  ( ADD     A,   3,   B )
4:  ( JUMP,   7,   0,   0 )
5:  ( ASGN,   23,  0,   A )
6:  ( SUB     A,   1,   B )
7:  ( MUL,    A,   B,   C )
8:  ( ADD,    C,   1,   A )
9:  ( JNEZ,   B,   2,   0 )
```



Basic block

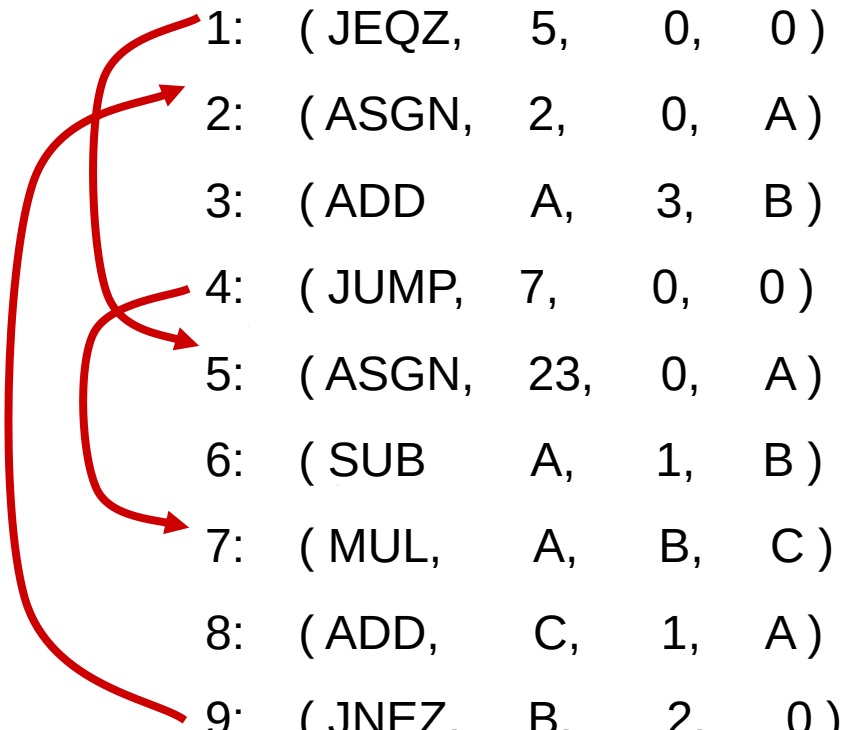
A **basic block** is a sequence of textually consecutive operations (e.g. MIR operations, LIR operations, quadruples) that contains no branches (except perhaps its last operation) and no branch targets (except perhaps its first operation).

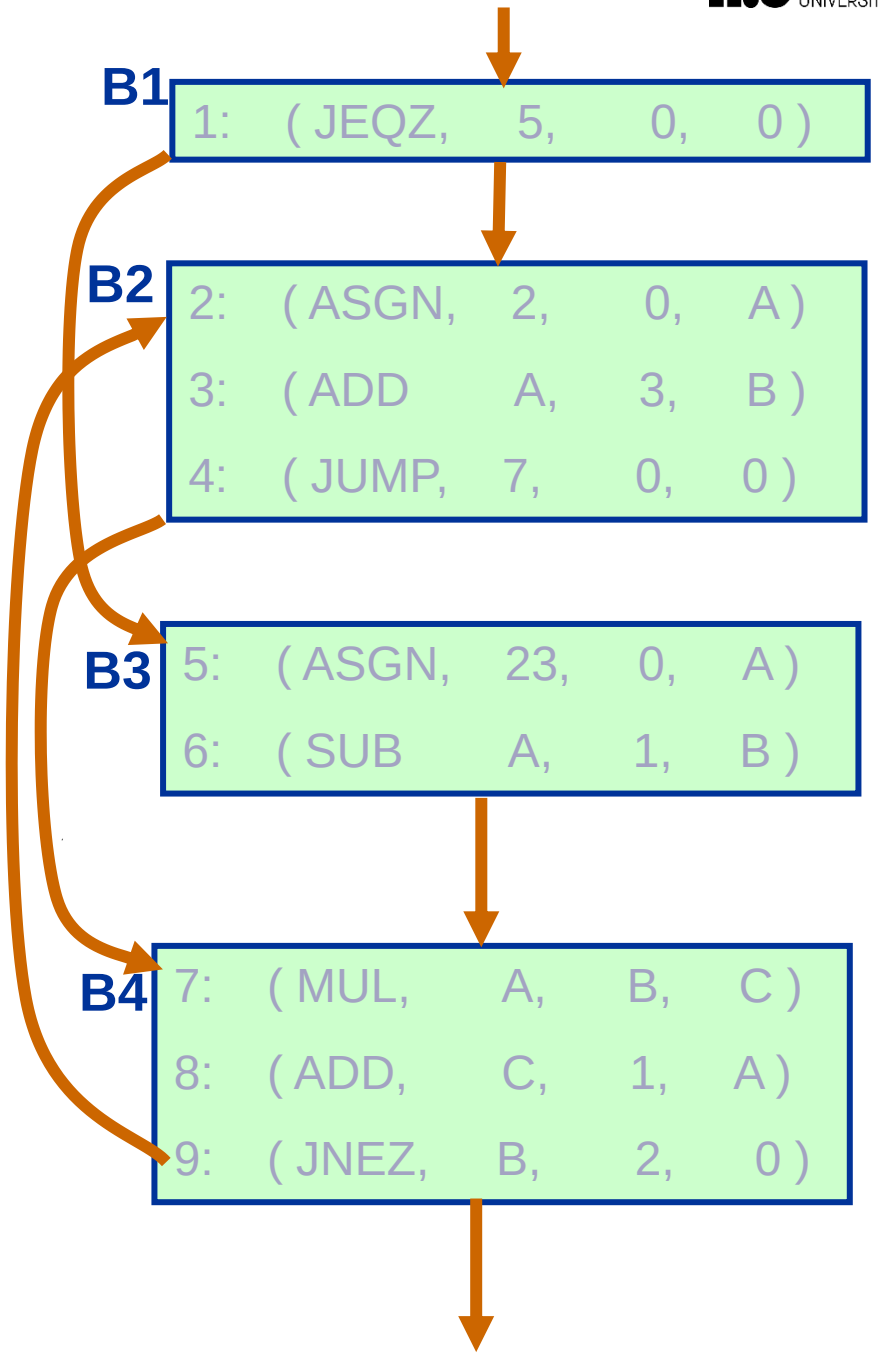
- Always executed in same order from entry to exit
- A.k.a. *straight-line code*



Basic block graph

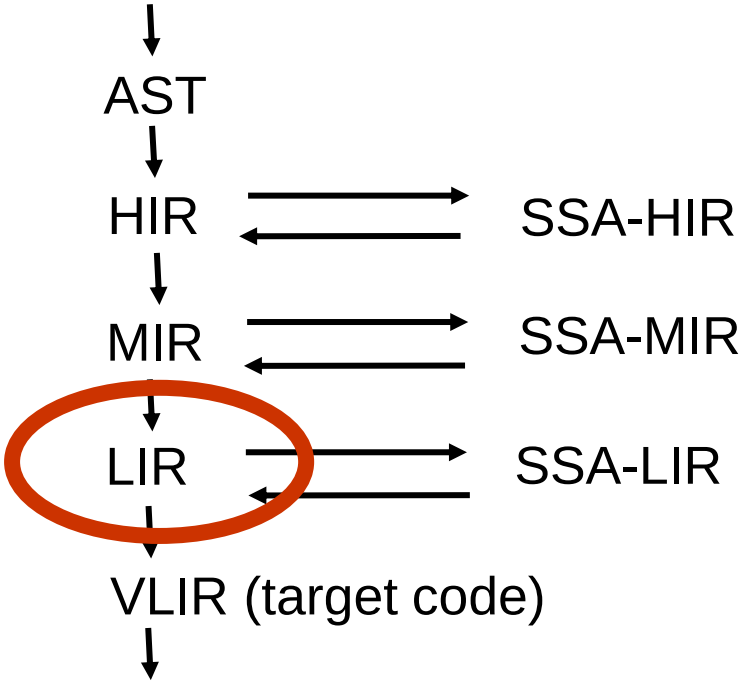
- Nodes: basic blocks
- Edges: control flow transitions





LIR – low-level intermediate representation

- in GCC: Register-transfer language (RTL)
- usually architecture dependent
  - e.g. equivalents of target instructions + addressing modes for IR operations
  - variable accesses in terms of target memory addresses



MIR→LIR: Lowering Variable Accesses

Seen earlier:

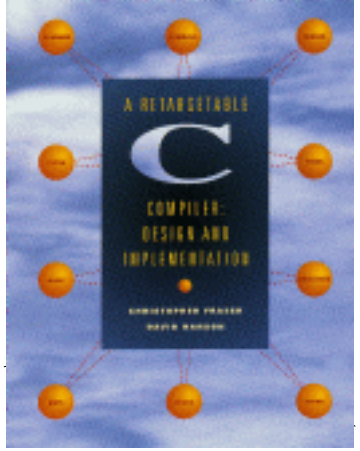
- HIR:
  t1 = a [i, j+2]
- the Lvalue of a [i, j+2] is
  (on a 32-bit architecture)

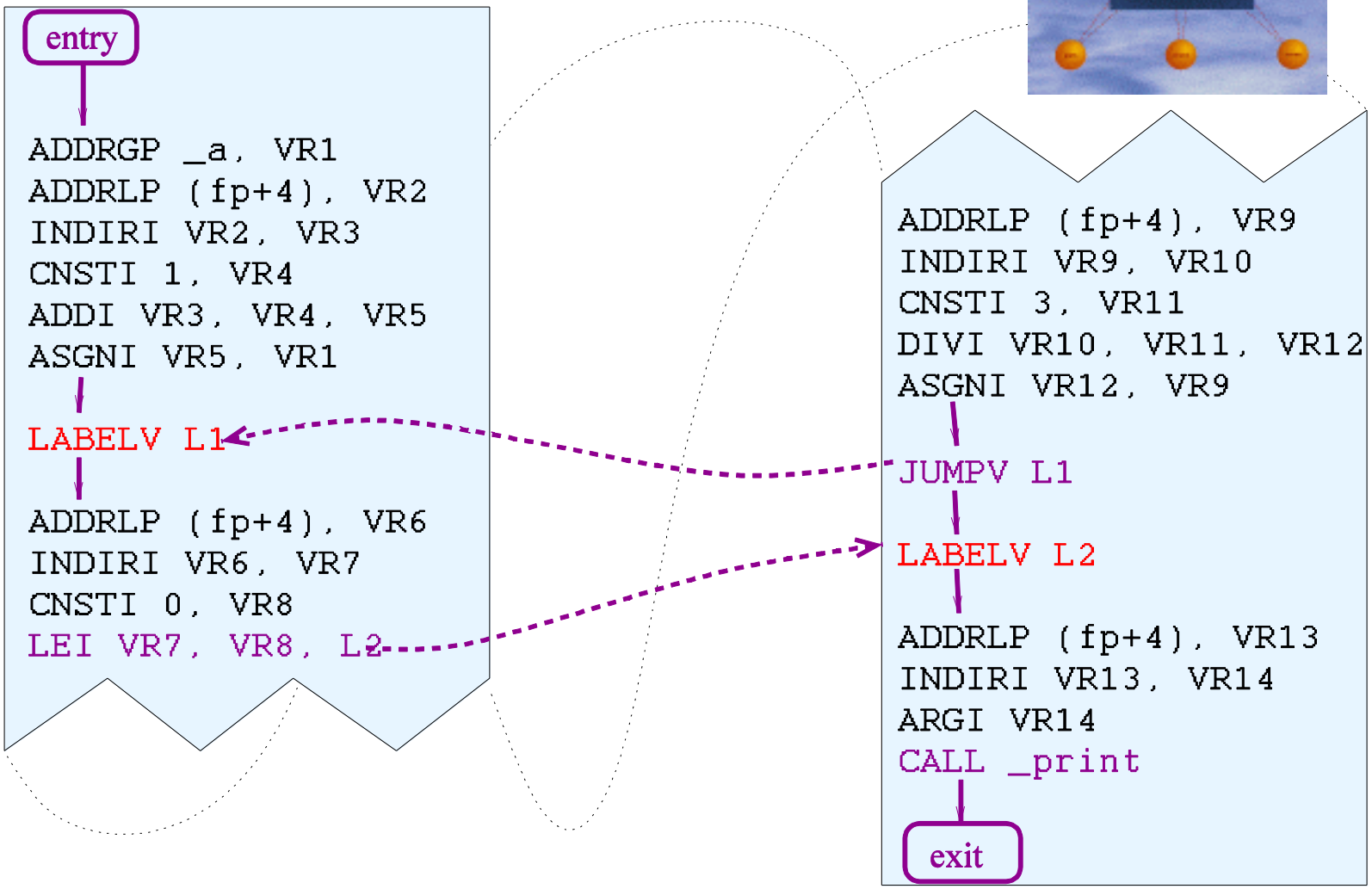
  (addr a) + 4 * (i * 20 + j + 2)
- MIR:
  t1 = j + 2
  t2 = i * 20
  t3 = t1 + t2
  t4 = 4 * t3
  t5 = addr a
  t6 = t5 + t4
  t7 = *t6

- Memory layout:
  - Local variables relative to procedure frame pointer fp
  - j at fp – 4
  - i at fp – 8
  - a at fp – 216
- LIR:
  r1 = [fp – 4]
  r2 = r1 + 2
  r3 = [fp – 8]
  r4 = r3 * 20
  r5 = r4 + r2
  r6 = 4 * r5
  r7 = fp – 216
  f1 = [r7 + r6]

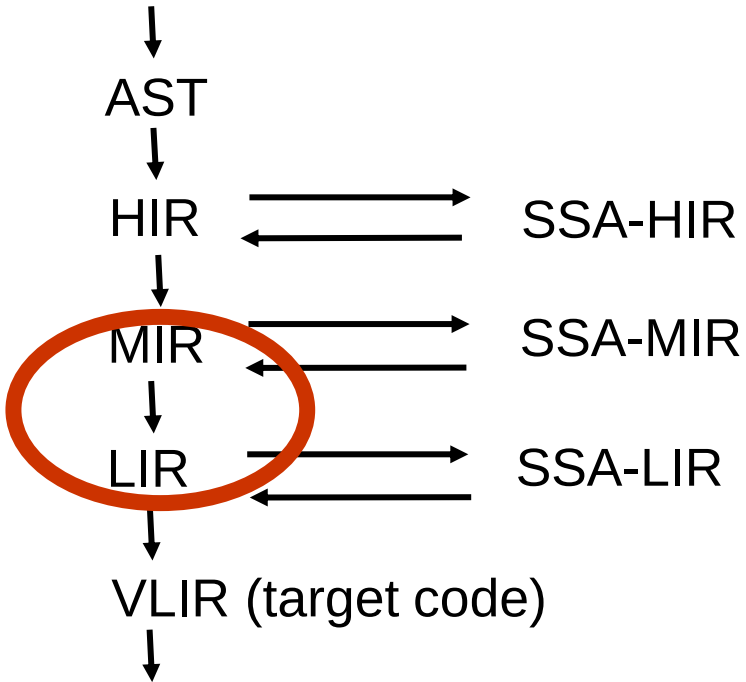
Example: The LCC-IR

- LIR – DAGs (Fraser, Hanson '95)





Flattening 2: From MIR to LIR



Christoph Kessler, IDA,
Linköping University

MIR→LIR: Storage Binding

- mapping variables (symbol table items) to addresses
- (virtual) register allocation
- procedure frame layout implies addressing of formal parameters and local variables relative to frame pointer fp, and parameter passing (call sequences)
- for accesses, generate Load and Store operations
- further lowering of the program representation

MIR→LIR translation example

MIR:

a = a * 2

b = a + c [1]

LIR, bound to storage locations:

r1 = [gp+8] // Load
r2 = r1 * 2
[gp+8] = r2 // Store
r3 = [gp+8]
r4 = [fp – 56]
r5 = r3 + r4
[fp – 20] = r5

LIR, bound to symbolic registers:

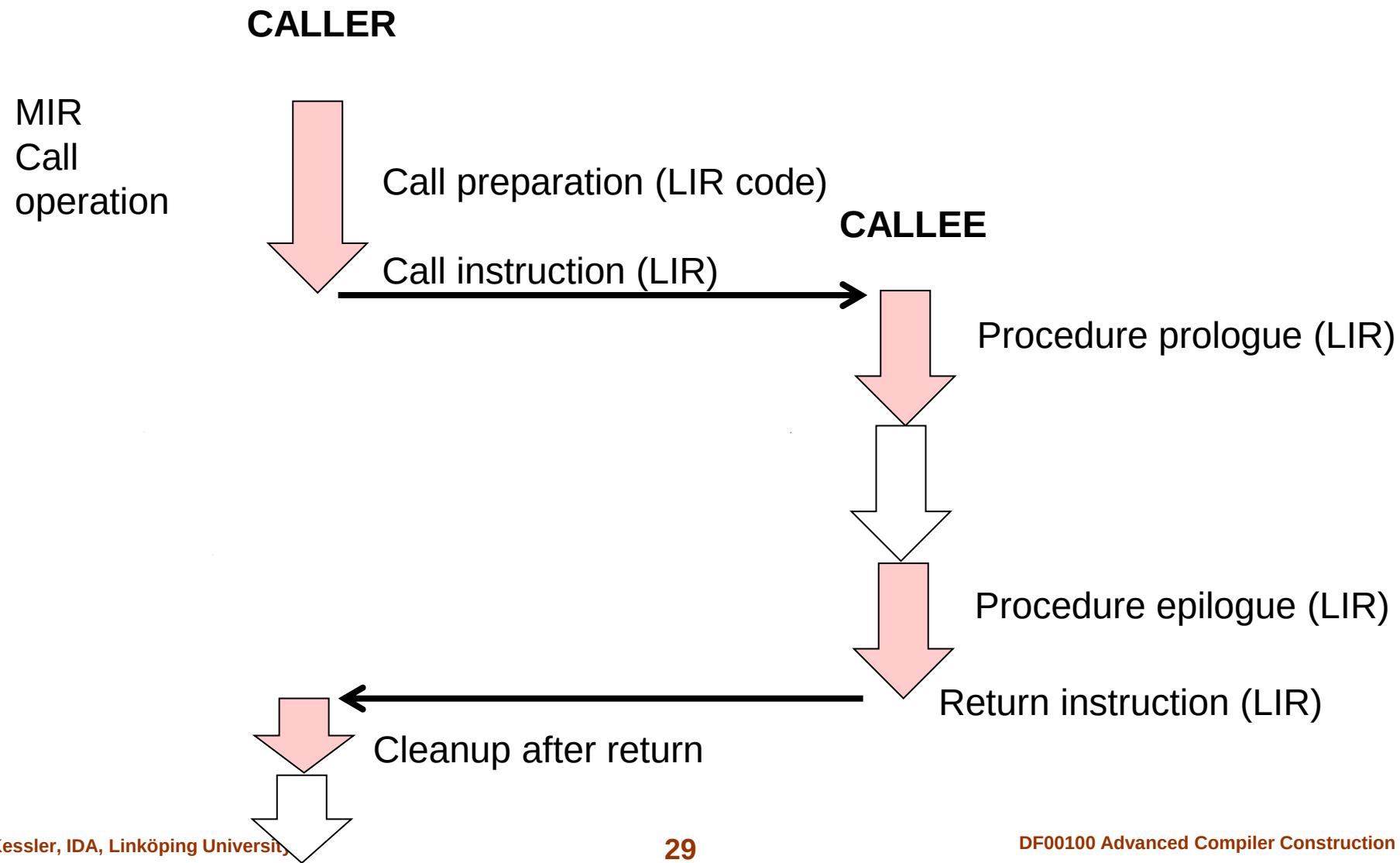
s1 = s1 * 2

s2 = [fp – 56]
s3 = s1 + s2

Storage layout:
Global variable a addressed relative to global pointer gp
local variables b, c relative to fp

MIR→LIR: Procedure call sequence (0)

[Muchnick 5.6]

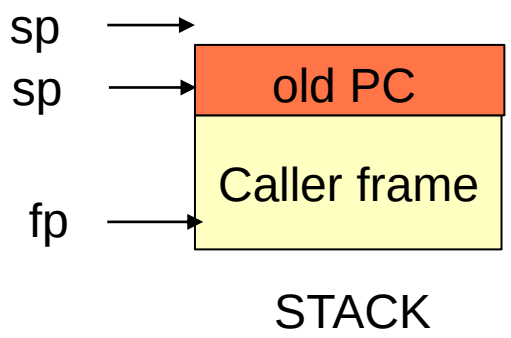


MIR→LIR: Procedure call sequence (1)

[Muchnick 5.6]

MIR call instruction assembles arguments and transfers control to callee:

- evaluate each argument (reference vs. value param.) and
  - push it on the stack, or write it to a parameter register
- determine code address of the callee (mostly, compile-time or link-time constant)
- store caller-save registers (usually, push on the stack)
- save return address (usually in a register) and branch to code *entry* of callee.

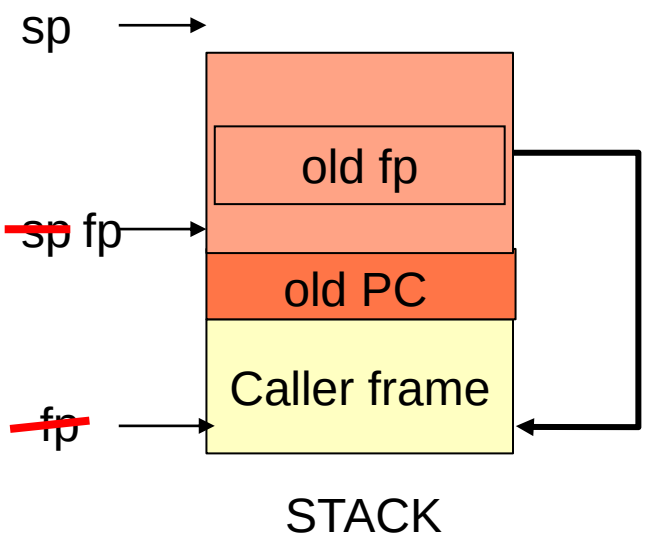


MIR→LIR: Procedure call sequence (2)

Procedure prologue

executed on entry to the procedure

- save old frame pointer fp
- old stack pointer sp becomes new frame pointer fp
- determine new sp (creating space for local variables)
- save callee-save registers



MIR→LIR: Procedure call sequence (3)

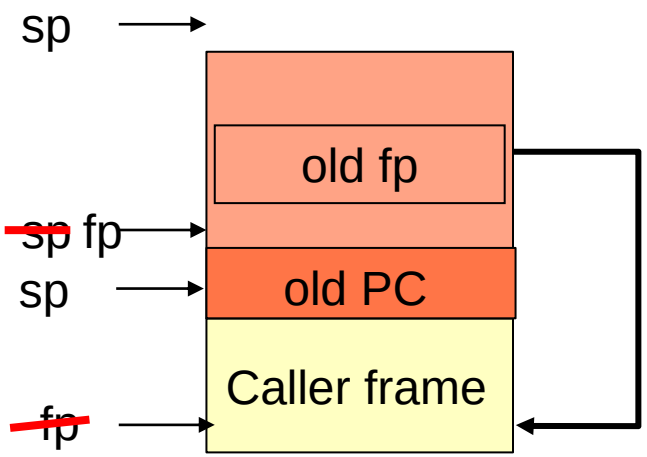
Procedure epilogue

executed at return from procedure

- restore callee-save registers
- put return value (if existing) in appropriate place (reg/stack)
- restore old values for sp and fp
- branch to return address (pop old PC)

Caller cleans up upon return:

- restore caller-save registers
- use the return value (if applicable)



From Trees to DAGs:

Common Subexpression Elimination (CSE)

E.g., at MIR→LIR Lowering

Christoph Kessler, IDA,
Linköping University

From Trees to DAGs: Local CSE (Common Subexpression Elimination)

start with an empty DAG.

for each (MIR) operation $v = (\ t,\ op,\ u_1,\ u_2\ )$ $\qquad t \leftarrow u_1\ op\ u_2$

  for each operand u_i of v

    if u_i not yet represented by a DAG node d_i

      create DAG leaf node $d_i \leftarrow$ **new dagnode**$(\ u_i,\ LEAF,\ NULL,\ NULL\ )$

      // LEAF denotes a value live on entry to basic block

  if there is no parent node p of all d_i with operator op in the DAG

    create $p \leftarrow$ **new dagnode**$(\ v,\ op,\ d_1,\ d_2\ )$

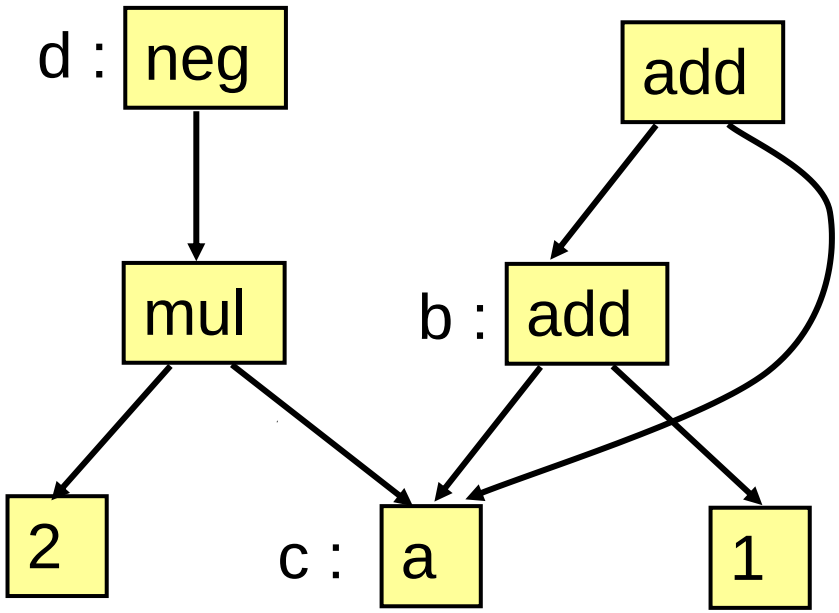
  label p with t *// result (temp.) variable*

  remove t as label of any other node in the DAG

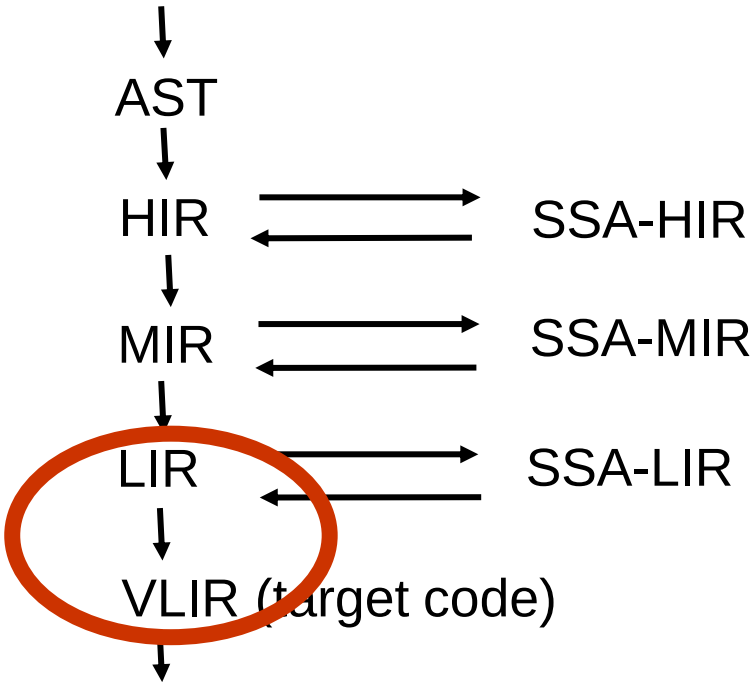
for all non-removed labels create assignments.

Local CSE on MIR produces a MIR DAG

1. c = a
2. b = a + 1
3. c = 2 * a
4. d = – c
5. c = a + 1
6. c = b + a
7. d = 2 * a
8. b = c



Flattening 3: From LIR to VLIR



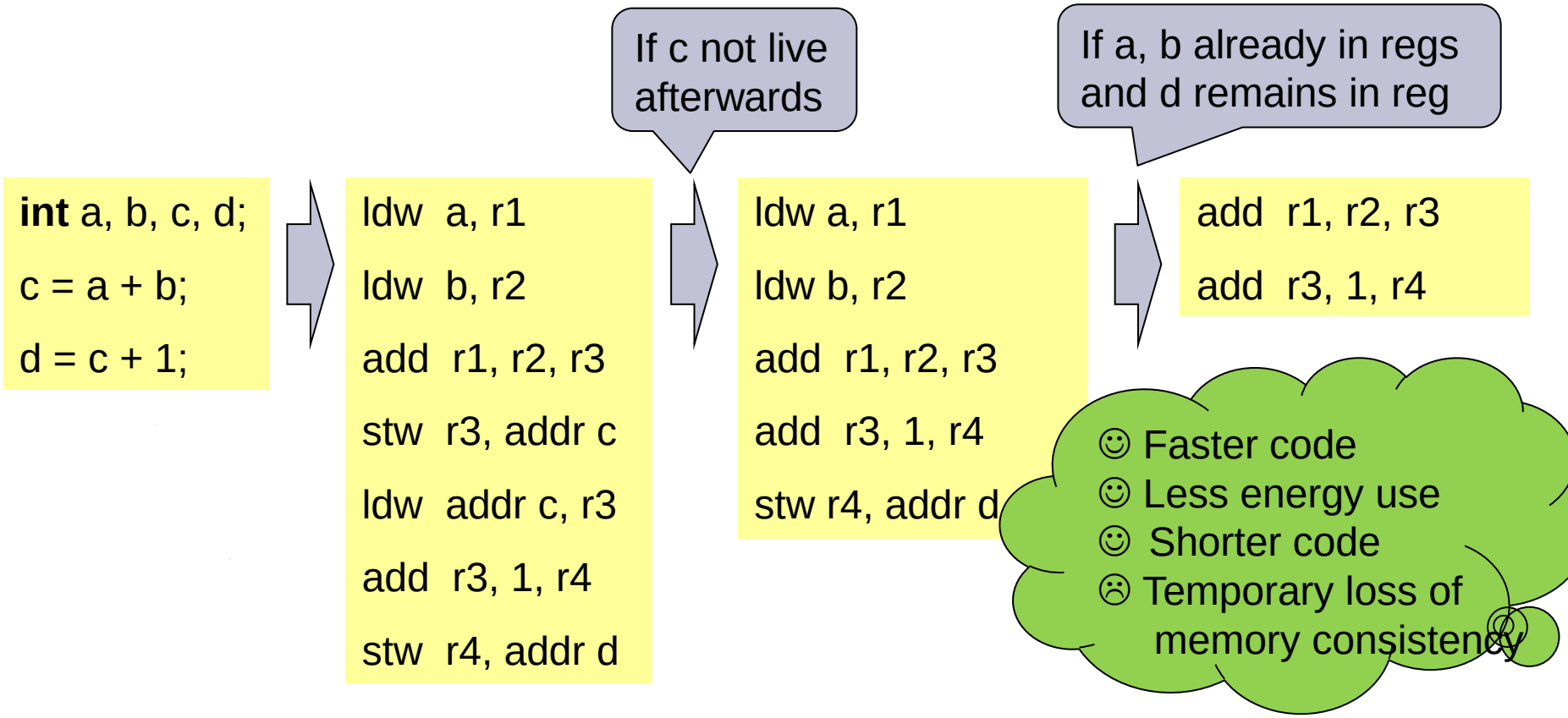
Christoph Kessler, IDA,
Linköping University

LIR→VLIR: Instruction selection

- LIR has often a lower level of abstraction than most target machine instructions (esp., CISC, or DSP-MAC).
- One-to-one translation LIR-operation to equivalent target instruction(s) (“macro expansion”) cannot make use of more sophisticated instructions
- Pattern matching necessary!

LIR / VLIR: Register Allocation

- Example for a SPARC-specific VLIR



- Loads and stores can be very expensive in both time and energy on modern CPUs, especially if not hitting in L1 cache → a lot can be gained by good register allocation.
- Register allocation needs to know about life spans of variables → program analysis

On LIR/VLIR: Global register allocation

- **Register allocation**
  - determine what values to keep in a register
  - “symbolic registers”, “virtual registers”
- **Register assignment**
  - assign virtual to physical registers
  - Two values cannot be mapped to the same register if they are *alive* simultaneously, i.e. their *live ranges* overlap (depends on schedule).

On LIR/VLIR: Instruction scheduling

- Instruction scheduling reorders the instructions (LIR/VLIR) (subject to precedence constraints given by dependences) to minimize
  - space requirements (# registers)
  - time requirements (# CPU cycles)
  - power consumption
  - ...

Remarks on IR design (1) [Cooper'02]

AST? DAGs? Call graph? Control flow graph? Program dep. graph? SSA? ...

- Level of abstraction is critical for implementation cost and opportunities:
  - representation chosen affects the entire compiler

Example 1: Addressing for arrays and aggregates (structs)

- source level AST: hides entire address computation A[i+1][j]
- pointer formulation: may hide critical knowledge (bounds)
- low-level code: may make it hard to see the reference

→ "best" representation depends on how it is used

- for dependence-based transformations: source-level IR (AST, HIR)
- for fast execution: pointer formulation (MIR, LIR)
- for optimizing address computation: low-level repr. (LIR, VLIR, target)

Remarks on IR Design (2)

Example 2: Representation for comparison&branch

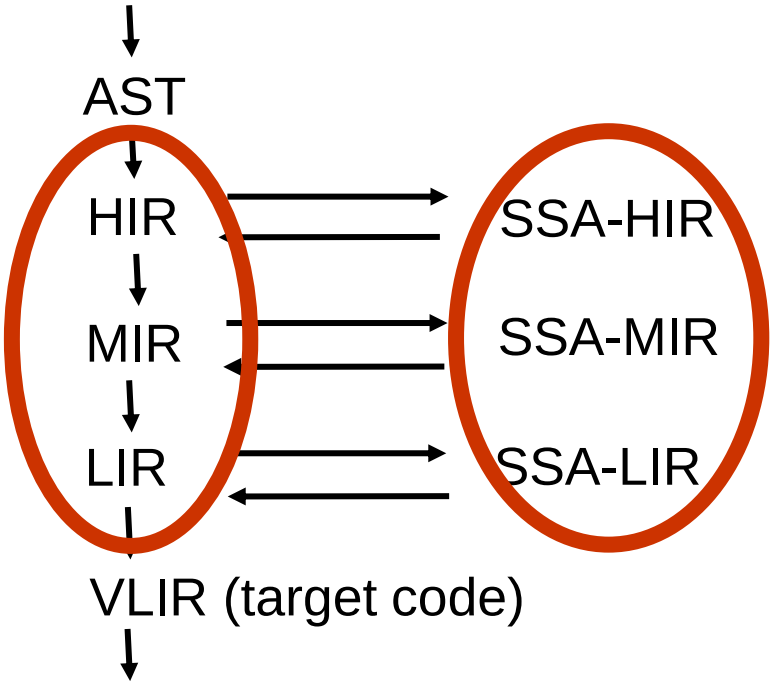
- fundamentally, 3 different operations:
  - compare → convert result to boolean → branch

  combined in different ways by processor architects
- "best" representation may depend on target machine

- r7 = (x < y)
br r7, L12 | cmp x y (sets CC)
brLT L12 | r7 = (x < y)
[r7] br L12

→ design problem for a retargetable compiler

Multi-Level IR, Standard vs. SSA Form



Christoph Kessler, IDA,
Linköping University

What is Static Single Assignment (SSA) Form?

A Short Introduction

(Details will follow later)

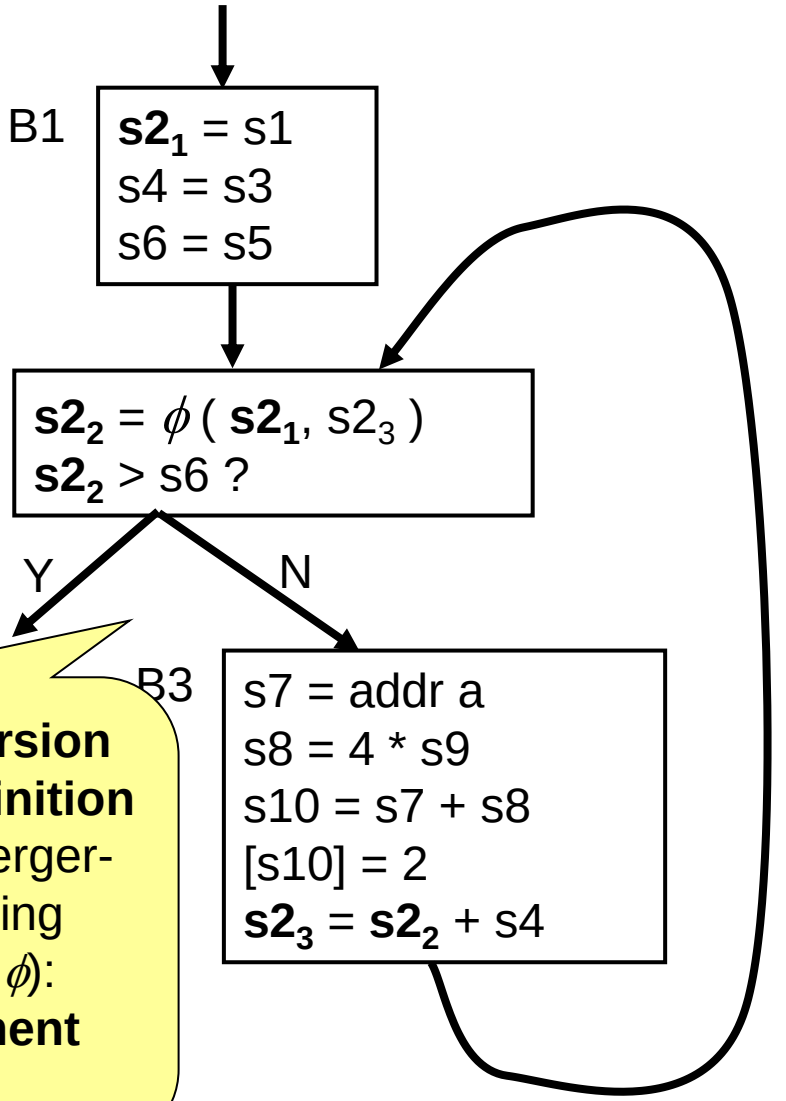
Christoph Kessler, IDA,
Linköping University

Example with SSA-LIR (adapted from Muchnick'97)

s2 is assigned (written, defined) multiple times in the program text (i.e., *multiple static assignments*)

- LIR:

```
      s2 = s1
      s4 = s3
      s6 = s5
L1:   if s2 > s6 goto L2
      s7 = addr a
      s8 = 4 * s9
      s10 = s7 + s8
      [s10] = 2
      s2 = s2 + s4
      goto L1
L2:
```



After introducing one **version** of s2 **for each static definition** and explicit data-flow merger-ops for different reaching versions (***phi nodes***, ϕ): **Static single assignment (SSA) form**

Static Single Assignment (SSA) Form

Goal:

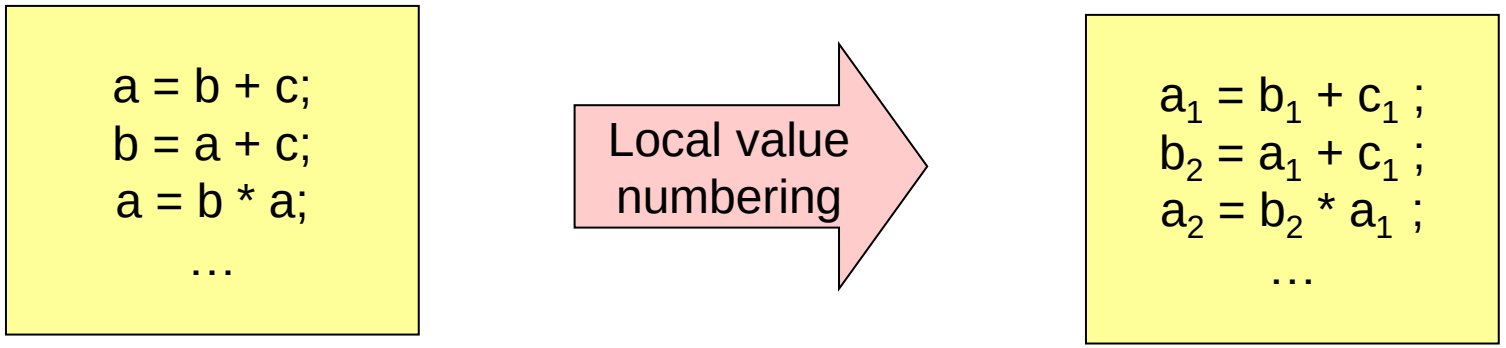
- increase efficiency of inter/intra-procedural analyses and optimizations
- speed up dataflow analysis
- represent def-use relations explicitly

Idea:

- Represent program as a directed graph of operations op
  - Represent statements / quadruples / instructions as assignments v = v' op v'' with *v, v', v''* a variable / label / symbolic register / temporary (edge) connecting operations
- **SSA-Property:**
  There is only *one position* (statement, quadruple, instruction) in a program/procedure defining a variable version *v* → *static value*
  - Does not mean that *v* is computed only once at runtime:
    Due to iteration / recursion, the program point may be executed more than once with different *dynamic* values.

SSA Construction (1): Value Numbering in a Single Basic Block

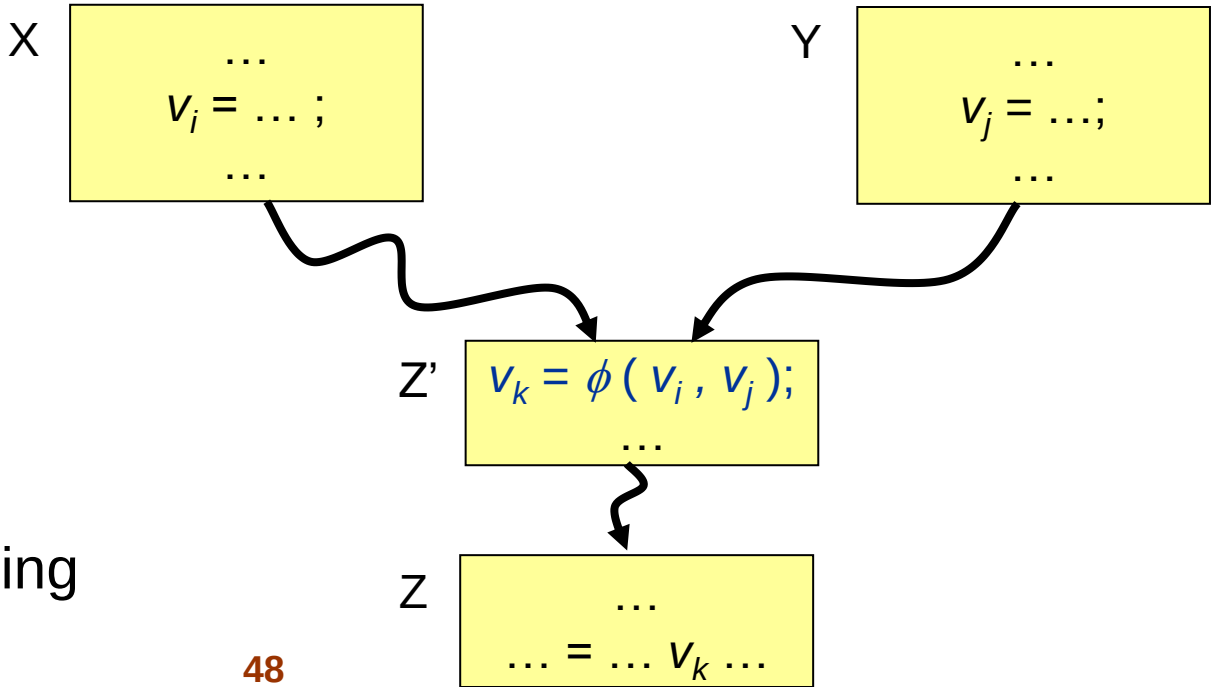
- Assign a distinct name (e.g. variable name + index) to each static value computed in the block
- Can be done on-the-fly when constructing DAGs (CSE, see Lecture 1)
- Makes local Def-Use chains explicit
  - (See lecture on data-flow analysis)
- For several basic blocks: use (procedure-wide) unique indices



SSA Construction (2) – Insert Phi nodes to stitch DU-chains between blocks together

For different basic blocks X and Y both defining a **variable** *v*, say v_i in X and v_j in Y, if non-empty **paths** X→$^+$ Z and Y→$^+$ Z exist in the control flow graph to a common successor block Z containing a **use** v_k of *v* that is **reached by both definitions**, with Z' being the **first common node** on the two paths, then a **Phi node** $v_k = \phi(v_i, v_j)$ must be inserted in Z'.

- In general, a Phi node in a block *B* has | Pred(*B*) | operands



→ Global value numbering

Algorithms for SSA Construction

- Standard algorithm by Cytron et al. 1989 (iterated dominance frontiers)
  - See Muchnick, Section 8.11
- Other algorithms for SSA construction exist
  - Optimize number of Phi nodes
- Standard transformations like constant folding, arithmetic simplification, common subexpression elimination can also reduce the number of Phi nodes.

- See the guest lectures by W. Löwe

R. Cytron et al.: Efficiently computing static single assignment form. Proc. POPL'89, pp. 25-35, ACM, 1989.

Muchnick: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997, Section 8.11

SSA: Some ramifications…

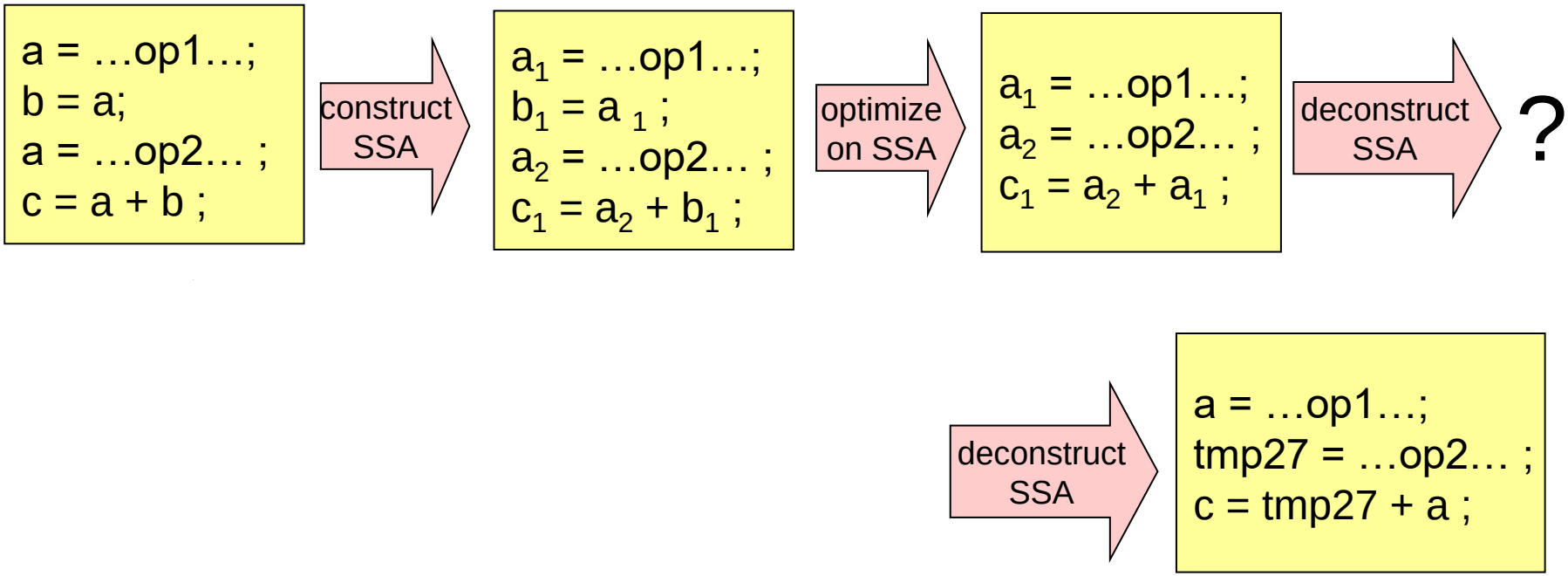
- Array variables??
  - Phi-node to copy entire array if only one element is written
  - Use special array Phi operators
- Dynamically allocated objects??
  - Example:

```
while (…) {
    ptr = new Listitem();
    ptr->next = list;
    list = ptr;
}
```

  - Different created Listitem fields can no longer be identified and named statically
- **Memory-SSA**
  - For target-level SSA form: Dependences between Load and Store instructions through memory should be made explicit with special memory-Phi nodes

Converting from SSA back to standard IR

- Simply throwing away the indices and Phi nodes????
  - **No!**
    Optimizations on SSA representation may have created overlaps of different static values of the same variable…
  - Example:



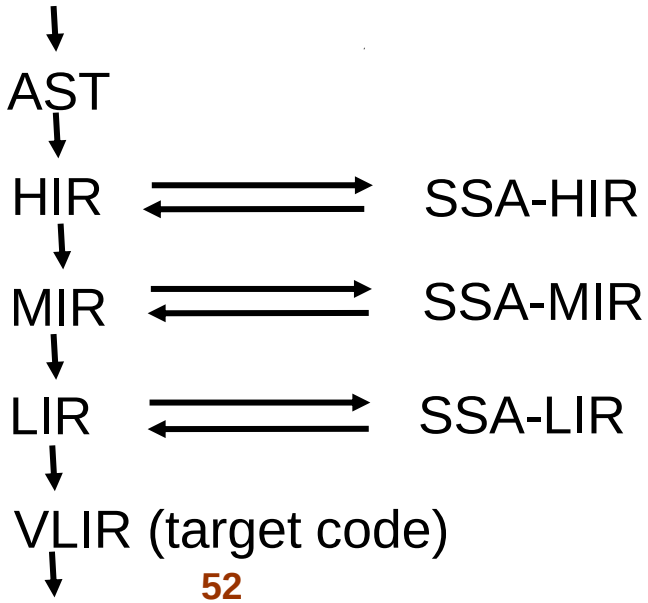
Summary (so far)

- Modern advanced compiler frameworks provide a **Multi-level IR**
  - Compilation flow = progressive lowering
  - ☺ Program analyses and transformations can work on the most appropriate level of abstraction
  - ☺ Clean separation of compiler phases
  - ☹ Compiler framework gets larger and slower

Lowering:

Gradual loss of source-level information

Increasingly target dependent



Survey of Some Compiler Frameworks

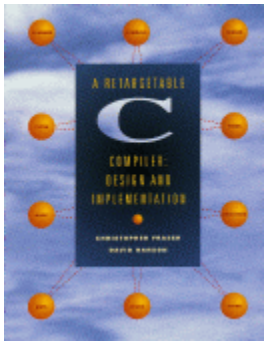
A (non-exhaustive) survey with a focus on open-source C/C++ compiler frameworks

(as far as time permits...)

Christoph Kessler, IDA,
Linköping University

LCC (Little C Compiler)

- Dragon-book style C compiler implementation in C
- Very small (20K Loc), well documented, well tested, widely used
- Open source: https://drh.github.io/lcc/
- Textbook *A retargetable C compiler* [Fraser, Hanson 1995] contains complete source code
- One-pass compiler, fast
- C frontend (hand-crafted scanner and recursive descent parser) with own C preprocessor
- Low-level IR
  - Basic-block graph containing DAGs of quadruples
  - No AST
- Interface to IBURG code generator generator
  - Example code generators for MIPS, SPARC, Alpha, x86 processors
  - Tree pattern matching + dynamic programming
- Few optimizations only
  - local common subexpression elimination, constant folding
- Good choice for source-to-target compiling if a prototype is needed soon



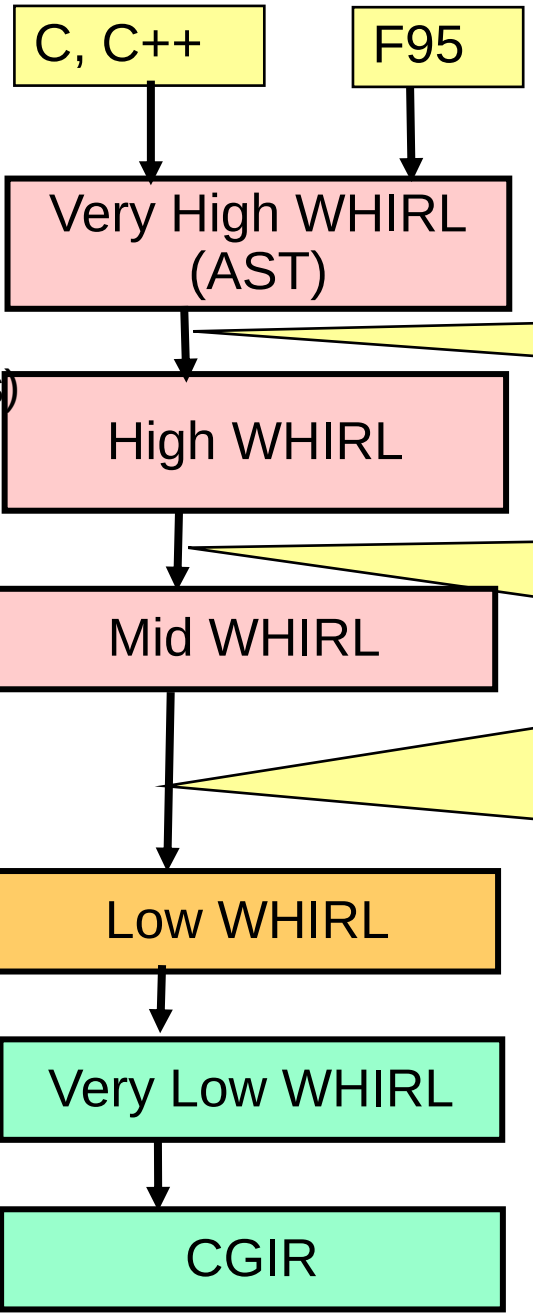
GCC

- Gnu Compiler Collection (earlier: Gnu C Compiler)
- Compilers for C, C++, Fortran, Java, Objective-C, Ada …
  - sometimes with own extensions, e.g. Gnu-C
- Open-source, developed since 1985
- Very large
- 3 IR formats (all language independent)
  - GENERIC: tree representation for whole function (also statements)
  - GIMPLE (simple version of GENERIC for optimizations)
    based on trees but expressions in quadruple form.
    High-level, low-level and SSA-low-level form.
  - RTL (Register Transfer Language, low-level, Lisp-like – the traditional GCC-IR)
    only word-sized data types; stack explicit; statement scope
- Many optimizations
- Many target architectures
- Since version 4.x (~2004) GCC has strong support for retargetable code generation
  - Machine description in .md file
  - Reservation tables for instruction scheduler generation
- Good choice if one has the time to get into the framework
- The compiler research community increasingly switches to LLVM instead

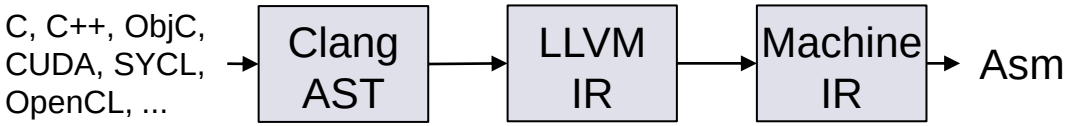
Open64 / ORC Open Research Compiler

- Based on SGI Pro-64 Compiler for MIPS processor, written in C++, went open-source in 2000
- Several tracks of development (Open64, ORC (ipf-orc.sourceforge.net), …)
- For Intel Itanium (IA-64) and x86 (IA-32) processors.
  Also retargeted to x86-64, Ceva DSP, Tensilica, XScale, ARM …
  ”simple to retarget” (?)
- Languages: C, C++, Fortran95 (uses GCC as frontend),
  OpenMP and UPC (for parallel programming)
- Industrial strength, with contributions from Intel, Pathscale, …
- 6-layer IR:
  - WHIRL (VH, H, M, L, VL) – 5 levels of abstraction
    - All levels semantically equivalent
    - Each level is a lower level subset of the higher form
  - and target-specific very low-level CGIR
- Many optimizations, many third-party contributed components
- Has been used in a number of research projects and also by industry, e.g. in the Nvidia CUDA toolchain for part of the optimizations.
- Nowadays mostly replaced by LLVM
  as the main research framework for 64-bit target compilation

Open64 WHIRL



LLVM https://llvm.org



- ”Low-level virtual machine”
- Started ~2002 by C. Lattner and V. Adve from Univ. of Illinois at Urbana-Champaign
- Front-ends (Clang, GCC) for C, C++, Objective-C, Fortran, …
  - Can be used for source-to-source compilation, but not designed for it
- One IR level: a LIR + SSA-LIR,
  - linearized form, printable, shippable, but target-dependent
  - ”LLVM instruction set”
- compiles to many target platforms
  - x86, Itanium, ARM, Alpha, SPARC, PowerPC, Cell SPE, RISC-V, …
  - And to low-level C
- Link-time interprocedural analysis and optimization framework for whole-program analysis
- JIT support available for x86, PowerPC
- Open source
- Many subprojects
  - e.g. clang, polly, MLIR, runtime libraries (C, C++, OpenMP, OpenCL), ...
- Today the most common research compiler platform
- 2012 ACM Software System Award

More in the lab intro ...

Domain-Specific Compiler Frameworks

For example, the compilers for

- **Halide** (DSL for image processing)
- **Tensor Comprehensions** (DSL for linear algebra)
- **TensorFlow** (DSL for Machine learning)
- **OpenModelica** (DSL for Cyberphysical system modeling)
- **SkePU** (DSL for data-parallel computing using skeletons)
- **Polyhedral compilation** (for DSLs for loop nests accessing multidimensional dense arrays with affine subscript expressions)
- ...

DSLs may exist stand-alone (e.g. Modelica) or be **embedded** in general-purpose languages (e.g. SkePU).

DSL compilers often implement **domain-specific optimizations** leveraging **domain-specific restrictions of program semantics** that may not be applicable in general-purpose programming languages.

MLIR https://mlir.llvm.org

- Open-source compiler infrastructure project under LLVM
- Meta-IR (IR construction kit) to support DSL compilation atop LLVM
- **Customizable** IR with **only few concepts built-in:**
  - Namespace
  - Operations (no fixed set) and basic blocks
  - Values
  - Types known at compile-time
  - Attributes
- IR customization → MLIR dialects
  - Different abstraction levels (dialects) can co-exist
  - Reuse of MLIR tool infrastructure, e.g. parser (likewise customized)
- **Dialect** = grouped abstractions (rules and semantics) for some domain (DSL), IR level, or target architecture (e.g. GPU, FPGA)
  - declarative specification of custom operations with types, parsing, semantics... specified in MLIR's Operation Description Specification Language (ODS)
- Predefined dialects (~20): **Standard**, **SCF** (structured control flow), **LinAlg** (linear algebra), **Affine** (polyhedral representation), **Stencil** (iterative application of a stencil kernel), **F18** (FORTRAN), ...; the **LLVM** dialect models LLVM-IR (i.e., LIR)
  - Higher-level dialects are lowered to lower-level ones, eventually to LLVM
  - Lowering strategies and code generation included for predefined dialects

MLIR Example: Lowering

MLIR allows mixing levels of abstraction

- Easy dialect-to-dialect lowering
- Operations from different dialects can mix in same IR
- Lowering from “A” to “D” may skip “B” and “C”
- Avoid lowering too early and losing information
- Do compiler analyses at most suitable level of abstraction

MLIR tutorial, mlir.llvm.org

Example source:
L. Chelini: *Abstraction Raising in General-Purpose Compilers*. PhD thesis, TU Eindhoven, 2021

```
%0 = linalg.matmul(%A, %B, %C)        // linalg dialect
```

lowered to

```
affine.for %i = 0 to %N                          // affine dialect
  affine.for %j = 0 to %N
    affine.for %k = 0 to %N {
      %0 = affine.load %C[%i,%j] : memref<?x?xf32>
      %1 = affine.load %C[%i,%k] : memref<?x?xf32>
      %2 = affine.load %C[%k,%j] : memref<?x?xf32>
      %3 = std.mulf %1,%2
      %4 = std.addf %3,%0
      affine.store %4,%C[%i,%j] : memref<?x?xf32>
  }
```

lowered to

```
%s = constant 1 : index                          // scf dialect
for %i = 0 to %N step %S
  for %j = 0 to %N step %S
    for %k = 0 to %N step %S {
      %0 = load %C[%i,%j] : memref<?x?xf32>
      %1 = load %C[%i,%k] : memref<?x?xf32>
      %2 = load %C[%k,%j] : memref<?x?xf32>
      %3 = std.mulf %1,%2
      %4 = std.addf %3,%0
      store %4,%C[%i,%j] : memref<?x?xf32>
  }
```

Source-to-Source compiler frameworks

- **Cetus** (by Purdue University) https://engineering.purdue.edu/Cetus/
  - C / OpenMP source-to-source compiler written in Java.
  - Open source
- **ROSE** (by Lawrence Livermore National Labs) http://rosecompiler.org
  - C++, C, Fortran, UPC, OpenMP source-to-source compiler
  - AST representation
  - Open source (but commercial frontend in binary form)
  - Very complex, nontrivial to install
- **Mercurium** (by Barcelona Supercomputing Centre) https://pm.bsc.es/mcxx
  - C++, C, Fortran, OmpSs source-to-source compiler
  - AST representation
  - Extension by writing compiler phases
  - Open source
- **Clang**
  - C++ LLVM frontend, can be (mis-)used for source-to-source translation
  - Not designed for advanced source-to-source translation
- **Polyhedral compilation: Polly** (for LLVM), **PLUTO, CLoog, MIT Tiramisu, ...**
- **Tools and generators**
  - TXL source-to-source transformation system
  - ANTLR frontend generator ...

More frameworks (mostly historical) …

- **Some influential compiler frameworks of the 1990s/2000s**
  - **SUIF** Stanford university intermediate format, https://suif.stanford.edu
  - **Trimaran** (for instruction-level parallel processors) https://trimaran.org
  - **VEX** compiler (VLIW code generation)
  - **Polaris** (Fortran) UIUC
  - **Jikes** RVM (Java) IBM
  - **Soot** (Java)
  - GMD Toolbox / Cocolab **Cocktail**™ compiler generation tool suite
  - **CoSy**
  - and many others …
- And many more e.g. for the embedded domain …

Homework

- In preparation for tomorrow's lecture on control-flow analysis, recapitulate **Depth-First Search** from the DFS slide set available on the course web page.
  - If necessary, read up in
    - textbook "*Introduction to Algorithms*" by Cormen et al., or any other good algorithms textbook covering graph algorithms;
    - or in Section 7.2 of Muchnick's book "*Advanced Compiler Design and Implementation*", Morgan Kaufmann 1997

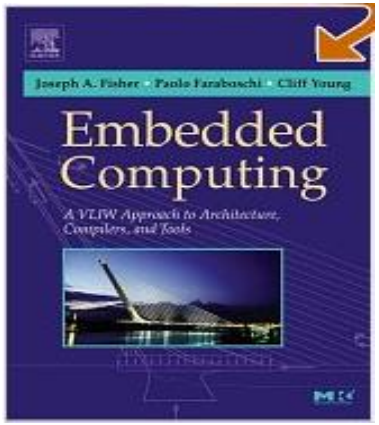
APPENDIX

More on compiler frameworks

Christoph Kessler, IDA,
Linköping University

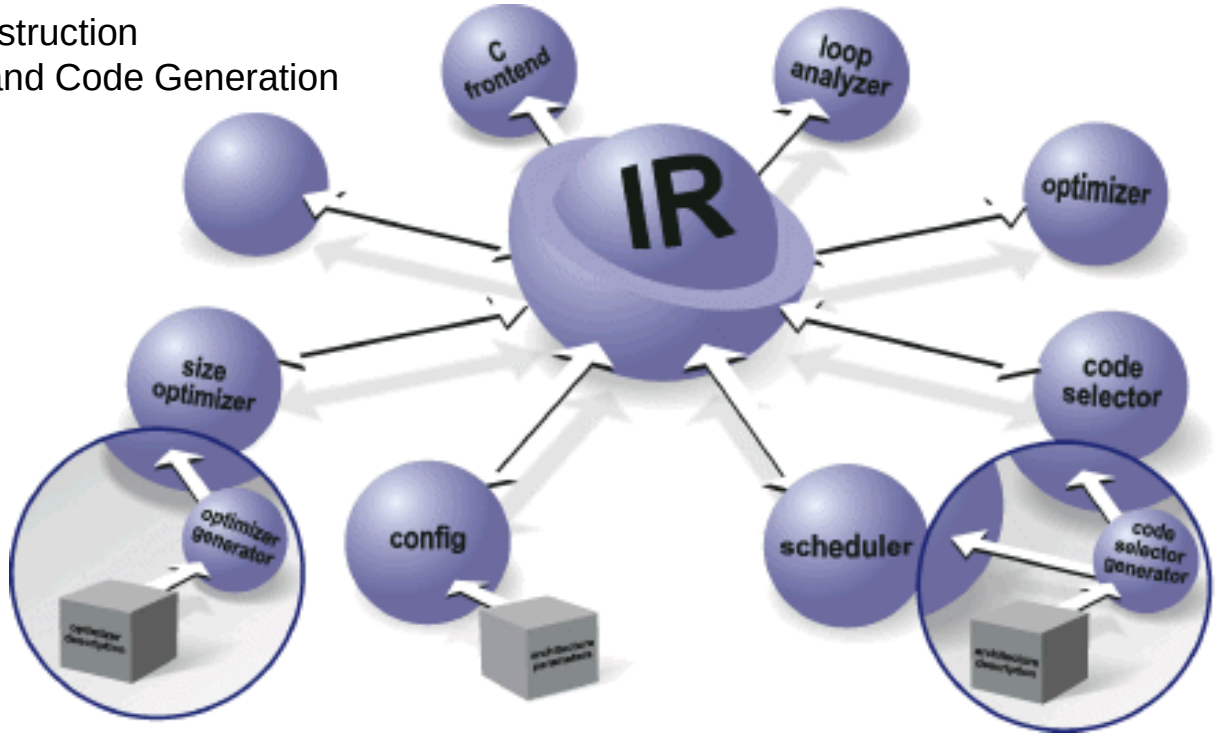
VEX Compiler

- VEX: ”**V**LIW **EX**ample”
  - Generic clustered VLIW Architecture and Instruction Set
- From the book by Fisher, Faraboschi, Young:
  Embedded Computing, Morgan Kaufmann 2005
  - www.vliw.org/book
- Developed at HP Research
  - Based on the compiler for HP/ST Lx (ST200 DSP)



- Compiler, Libraries, Simulator and Tools
  available in binary form from HP for non-commercial use
  - IR not accessible, but CFGs and DAGs can be dumped or visualized
- Transformations controllable by options and/or #pragmas
  - Scalar optimizations, loop unrolling, prefetching, function inlining, …
  - Global scheduling (esp., trace scheduling),
    but no software pipelining

CoSy



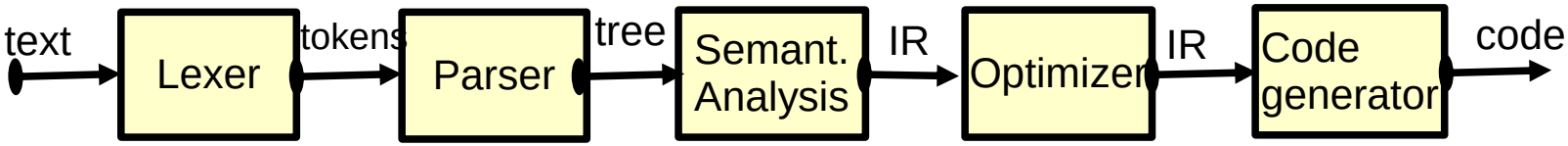
A commercial compiler framework

(formerly, www.ace.nl)

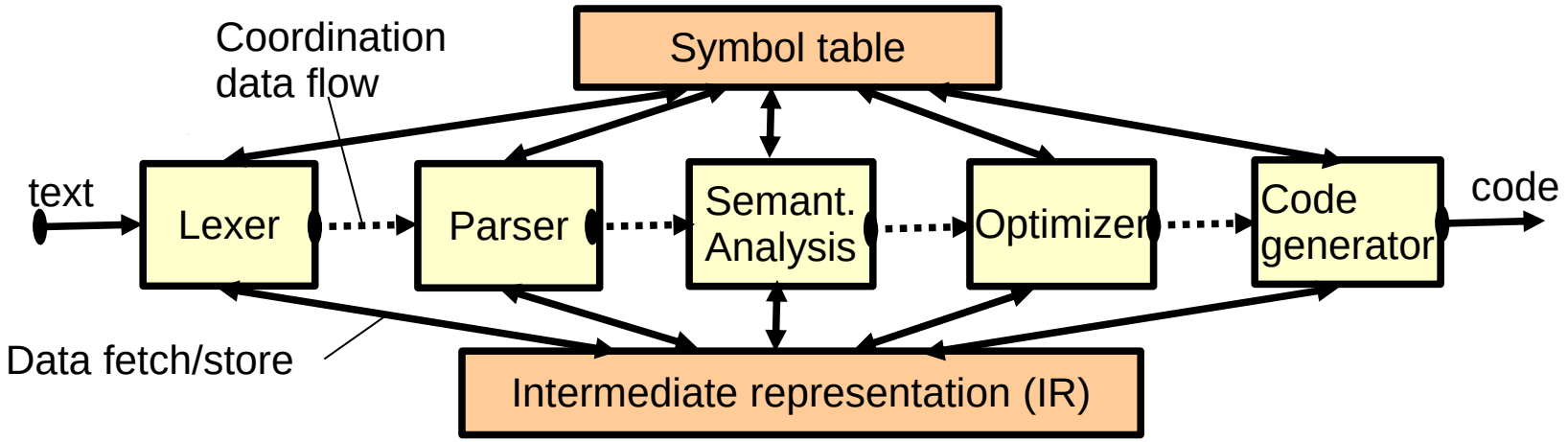
Christoph Kessler, IDA,
Linköping University

Traditional Compiler Structure

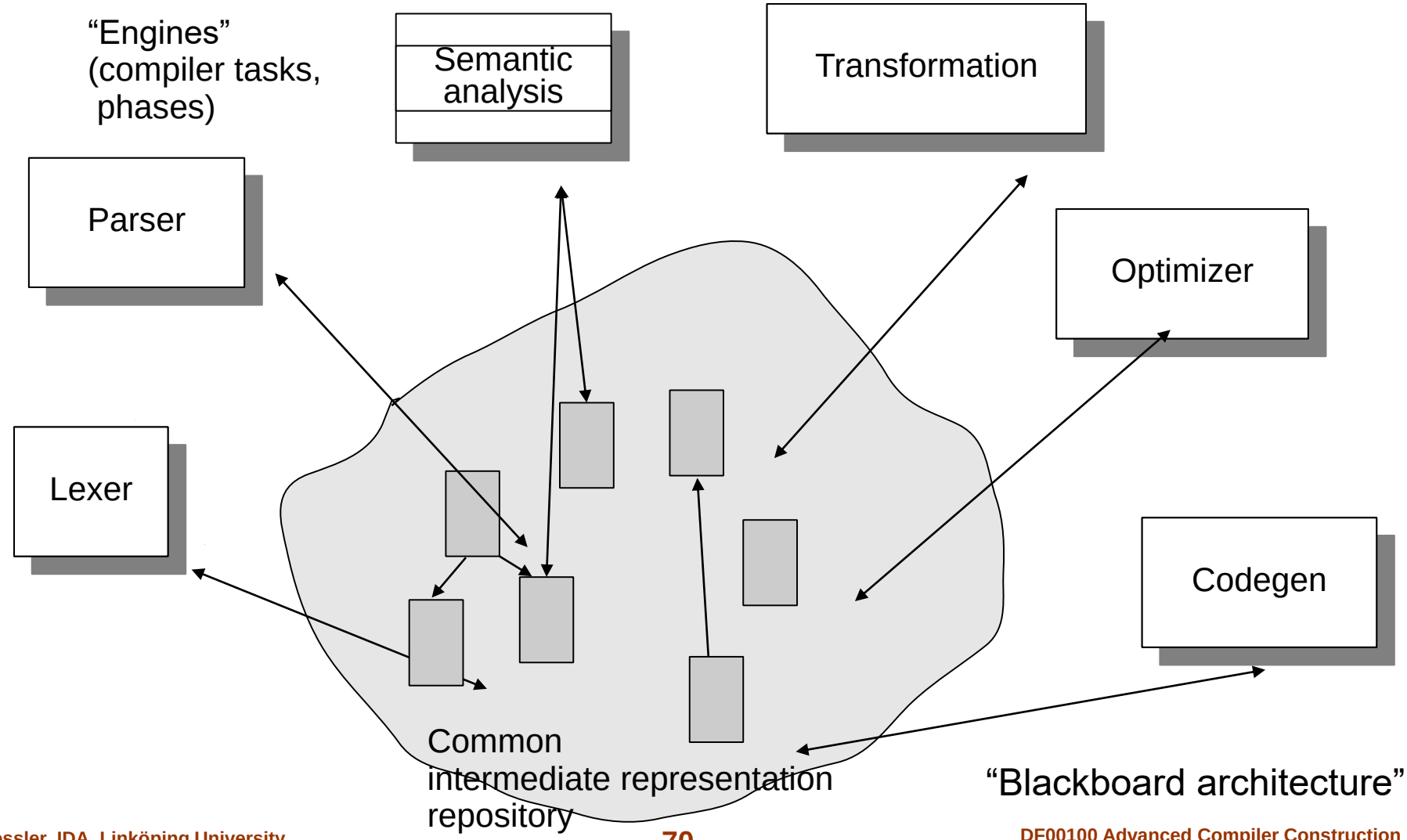
- Traditional compiler model: sequential process



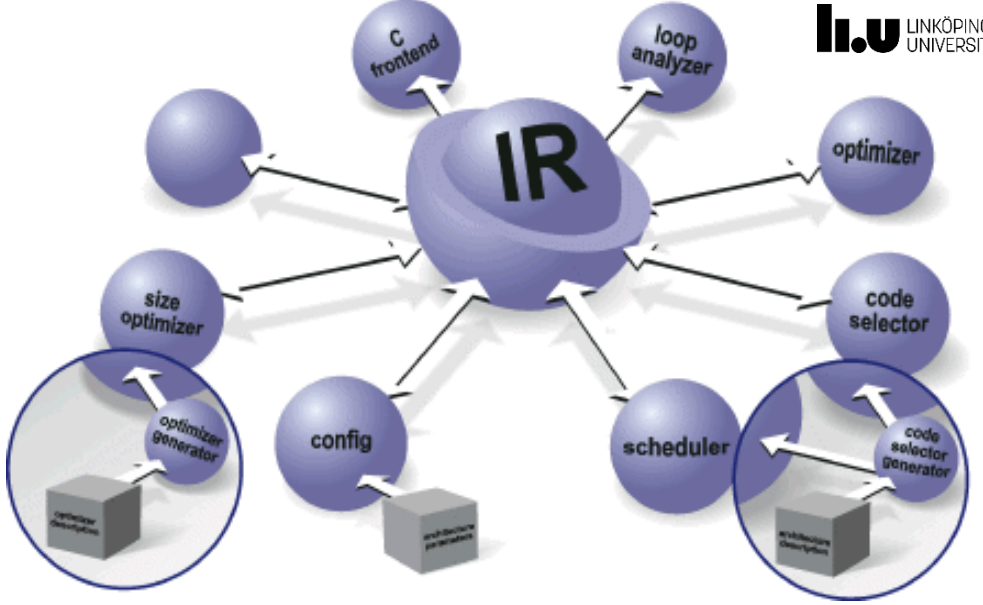
- Improvement: Pipelining
  (by files/modules, classes, functions)
- More modern compiler model with shared symbol table and IR:



A CoSy Compiler with Repository-Architecture



Engine



- Modular compiler building block
- Performs a well-defined task
- Focus on algorithms, not compiler configuration
- Parameters are handles on the underlying common IR repository
- Execution may be in a separate process or as subroutine call -
  the engine writer does not know!
- View of an engine class:
  the part of the common IR repository that it can access
  (scope set by access rights: read, write, create)

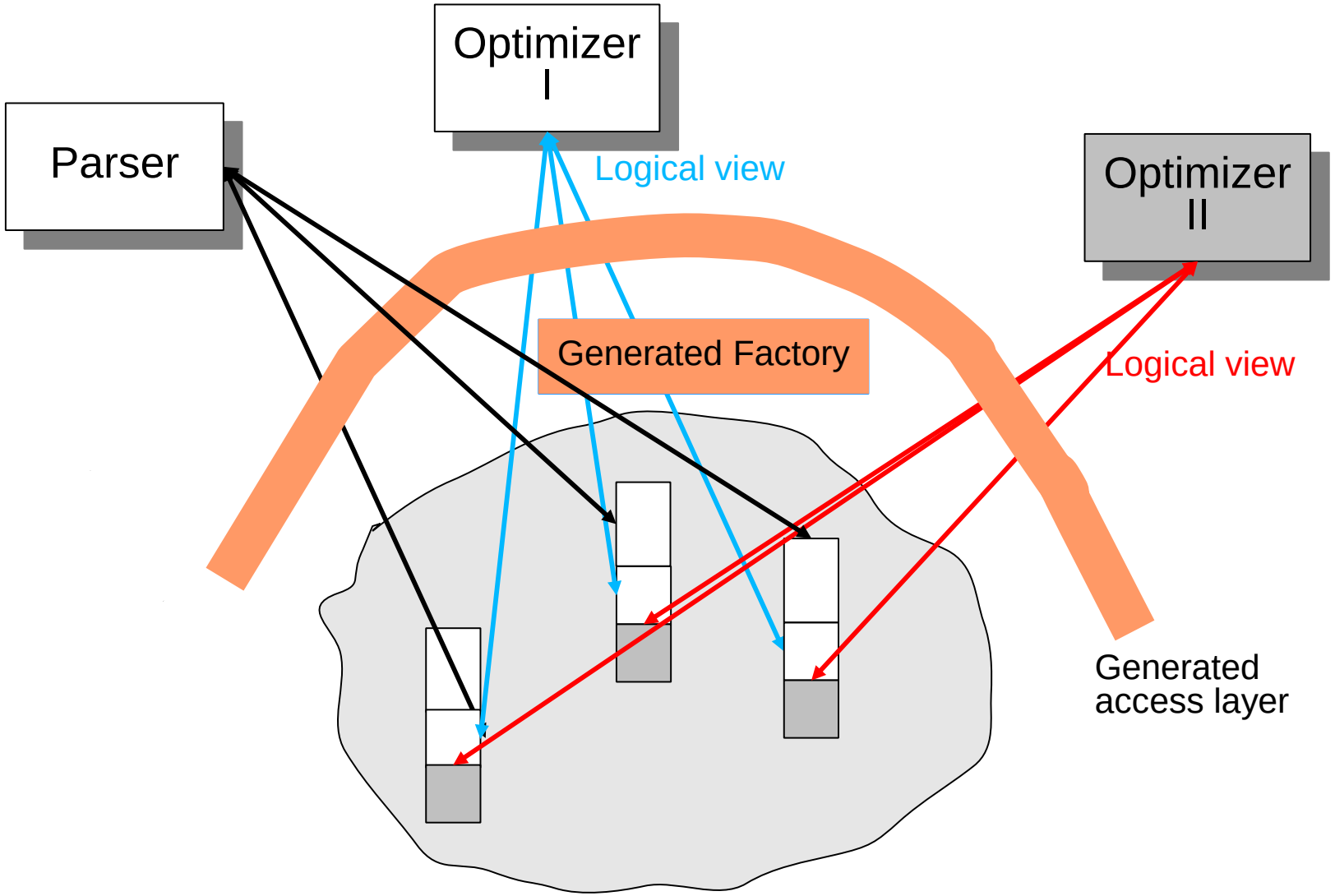
- Examples: Analyzers, Lowerers, Optimizers, Translators, Support

Composite Engines in CoSy

- Built from simple engines or from other composite engines **by combining engines in interaction schemes** (Loop, Pipeline, Fork, Parallel, Speculative, ...)
- Described in EDL (Engine Description Language)
- View defined by the joint effect of constituent engines
- A compiler is nothing more than a large composite engine

```
ENGINE CLASS compiler (IN u: mirUNIT) {
    PIPELINE
        frontend (u)
        optimizer (u)
        backend (u)
}
```

A CoSy Compiler



Example for CoSy EDL (Engine Description Language)

- Component classes (engine class)
- Component instances (engines)
- Basic components are implemented in C
- Interaction schemes (cf. skeletons) form complex connectors
  - SEQUENTIAL
  - PIPELINE
  - DATAPARALLEL
  - SPECULATIVE
- EDL can embed automatically
  - Single-call-components into pipes
  - p<> means a stream of p-items
  - EDL can map their protocols to each other (p vs p<>)

```
ENGINE CLASS optimizer ( procedure p )
{
    ControlFlowAnalyser cfa;
    CommonSubExprEliminator cse;
    LoopVariableSimplifier lvs;
    PIPELINE cfa(p); cse(p); lvs(p);
}

ENGINE CLASS compiler ( file f )
{  ....
   Token token;
   Module m;
   PIPELINE  // lexer takes file, delivers token stream:
            lexer( IN f, OUT token<> );
            // Parser delivers a module
            parser( IN token<>, OUT m );
            sema( m );
            decompose( m, p<> );
            // here comes a stream of procedures
            // from the module
            optimizer( p<> );
            backend( p<> );
}
```

Evaluation of CoSy

- The outer call layers of the compiler are generated from view description specifications
  - Adapter, coordination, communication, encapsulation
  - Sequential and parallel implementation can be exchanged
  - There is also a non-commercial prototype
    [Martin Alt: *On Parallel Compilation*. PhD thesis, 1997, Univ. Saarbrücken]

- Access layer to the repository must be efficient
  (solved by *generation* of source code for access macros)

- Because of views, a CoSy-compiler is very simply extensible
  - That's why it is expensive
  - Reconfiguration of a compiler within an hour