Compilers for Embedded Systems

Summer Term 2022

Heiko Falk

Institute of Embedded Systems
Electrical Engineering, Computer Science and Mathematics
Hamburg University of Technology

TUHH
Hamburg University of Technology

Institute of Embedded Systems

Chapter 3

Internal Structure of Compilers

Outline

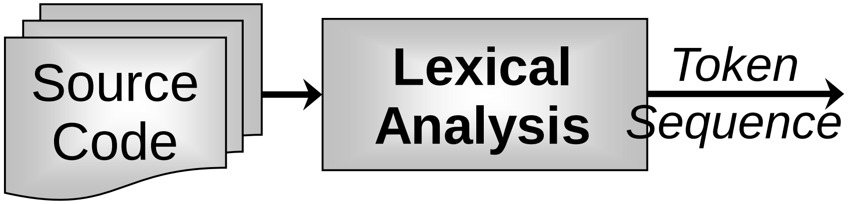
1. Introduction & Motivation
2. Compilers for Embedded Systems – Requirements & Dependencies
3. **Internal Structure of Compilers**
4. Pre-Pass Optimizations
5. HIR Optimizations and Transformations
6. Code Generation
7. LIR Optimizations and Transformations
8. Register Allocation
9. WCET-Aware Compilation
10. Outlook

Chapter Contents

3. Internal Structure of Compilers

- Compiler Stages
  - *Front-End:* Lexical Analysis, Syntactical Analysis, Semantical Analysis
  - *Back-End:* Code Generation, Register Allocation, Instruction Scheduling
- Intermediate Representations
  - *High-Level*, *Medium-Level* & *Low-Level IRs*
  - Case Studies: ICD-C, MIR, LLIR
  - Structure of a Highly-Optimizing Compiler
- Optimizations & Objectives
  - Abstraction Levels of Optimizations
  - Average-Case & Worst-Case Execution Time
  - Code Size
  - Energy Consumption

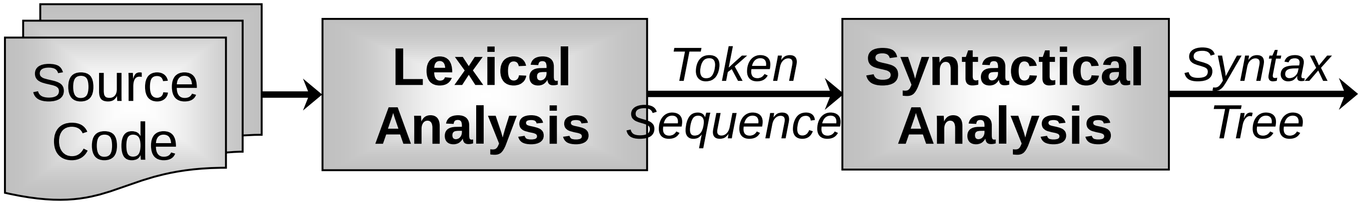
The Front-End (1)



Lexical Analysis (Scanner)

- Decomposition of the source code in lexical units (tokens)
- Detection of tokens (regular expressions, finite automata)
- Tokens represent strings of specific significance for the source language (e.g., identifier, constants, keywords)

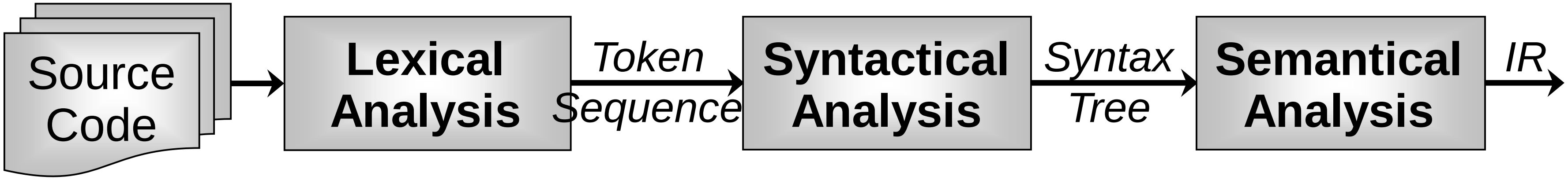
The Front-End (2)



Syntactical Analysis (Parser)

- Let *G* be the grammar of the source language
- Decides whether token sequence can be inferred from *G*.
- Syntax tree: Tree-like code representation based on the production rules of *G* used during inference
- Error processing

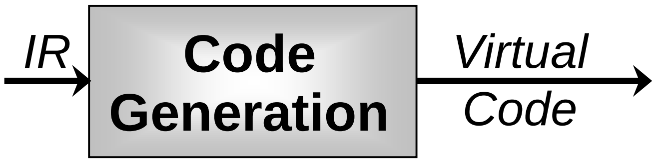
The Front-End (3)



Semantical Analysis (IR Generator)

- Name and scope analysis of symbols
- Type analysis
- Creation of symbol tables (mapping of identifiers to their types and locations)
- Generation of an intermediate representation (IR)

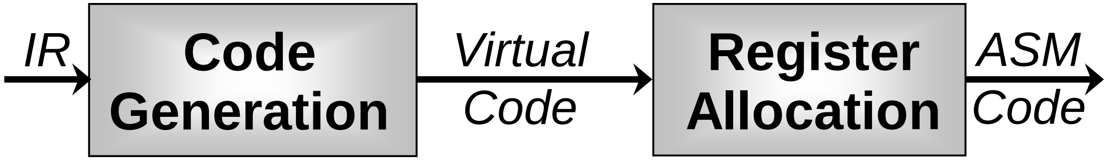
The Back-End (1)



Code Generation (Instruction Selection)

- Selection of machine instructions to implement the given IR
- *Often – Generation of virtual code:* non-executable assembler code; assumes an infinite amount of virtual registers, instead of a processor's limited/finite amount of physical registers
- *Alternatively – Generation of code with stack accesses:* executable assembler code; very restricted use of registers; variables are kept in memory (e.g., older GCCs at optimization level O0)

The Back-End (2)



Register Allocation

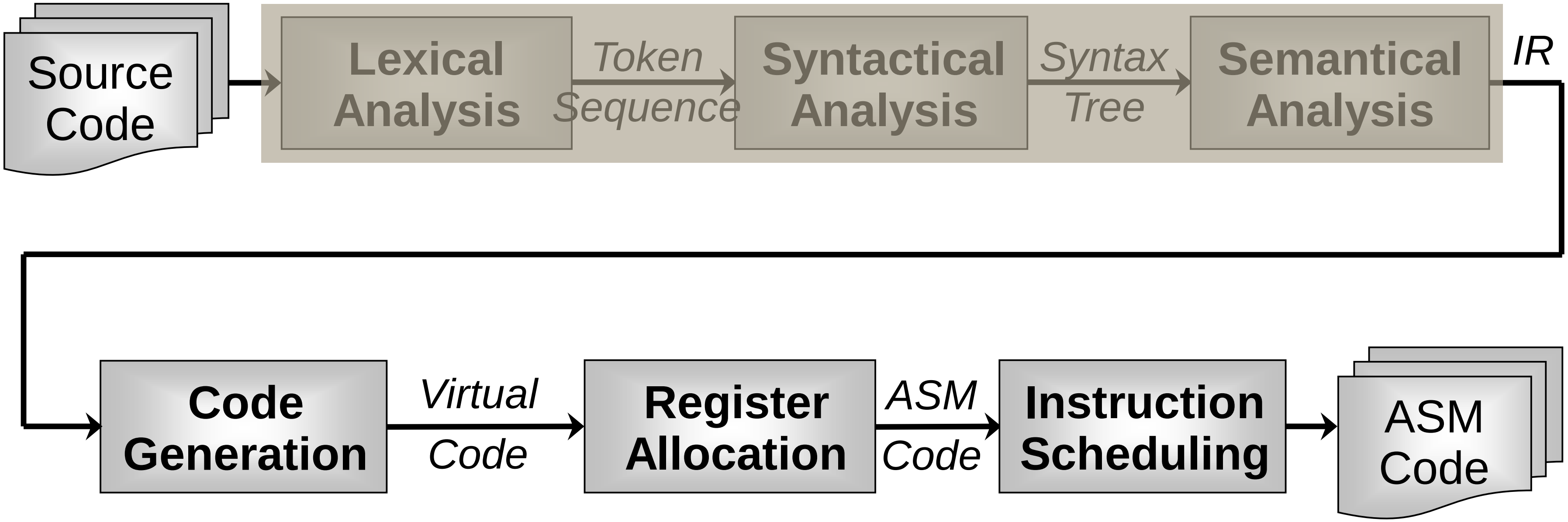
- *Either:* Mapping of virtual to physical registers
- *Or:* Replacement of stack accesses by keeping data in registers
- Insertion of memory transfers (spilling) if number of available physical registers is insufficient

The Back-End (3)

Instruction Scheduling

- Rearrangement of machine instructions in order to increase instruction-level parallelism
- Dependence analysis between machine instructions (data- & control-flow dependencies)

Putting It All Together

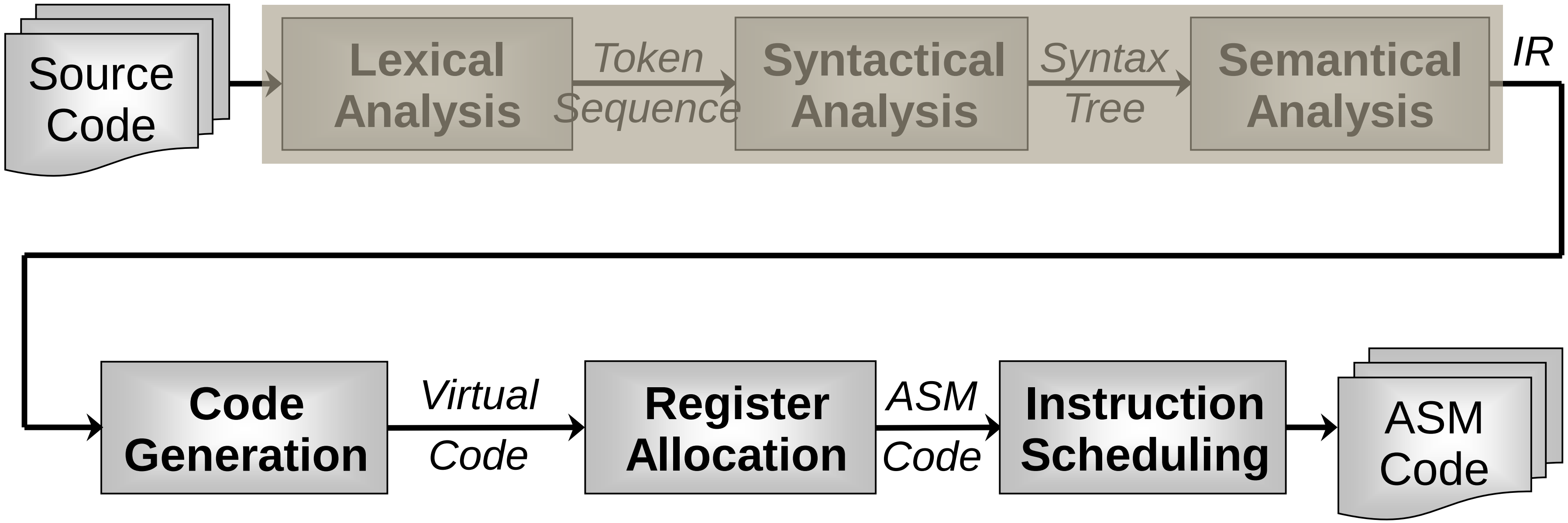


Course "Compilers for Embedded Systems"

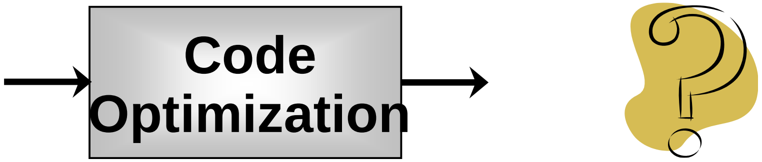
– Front-end not considered any further
(☞ Course "Compiler Construction")

– Focus: Back-end & compiler optimizations

Open Question



Where should optimizations be placed inside the compiler?



Code Optimization

Definition

- Compiler stage that reads code, modifies it and outputs it.
- Code modification aims to *improve* the code.

Remarks

- Optimizations usually do not generate *optimal* code (often undecidable), but (hopefully) *better* code.
- Code improvement is done subject to an *objective function*.

Existence of Formal Code Analyses

- Code modifications must not break correctness of the code.
- Optimizations must decide whether modifications of the code are legal or not.
- *Formal code analyses* are used to take these decisions.
- Examples: Control & data flow analyses, dependence analyses, ...

Prerequisites for Code Optimization

Required Compiler Infrastructure

- Effective internal representation of code that
  - easily supports code manipulation and
  - provides all necessary analyses for optimizations.

☞ Intermediate code Representations (IR)

Where should Optimizations be placed inside the Compiler?

- *Optimizations (usually) take place at the IR level within a compiler.*

Intermediate Representations (IRs)

- Compiler-internal data structures that model/represent the code to be translated or to be optimized.
- Good IRs also provide required code analyses, in addition to optimizations.

Chapter Contents

3. Internal Structure of Compilers

- Compiler Stages
  - *Front-End:* Lexical Analysis, Syntactical Analysis, Semantical Analysis
  - *Back-End:* Instruction Selection, Register Allocation, Instruction Scheduling
- Intermediate Representations
  - *High-Level*, *Medium-Level* & *Low-Level IRs*
  - Case Studies: ICD-C, MIR, LLIR
  - Structure of a Highly-Optimizing Compiler
- Optimizations & Objectives
  - Abstraction Levels of Optimizations
  - Average-Case & Worst-Case Execution Time
  - Code Size
  - Energy Consumption

Abstraction Levels of IRs (1)

```
float a[20][10];
... a[i][j+2] ...;
```

– *High-Level*	– *Medium-Level*	– *Low-Level*
`t1 ← a[i,j+2]`	$t_1 \leftarrow j+2$	`r1 ← [fp-4]`
	$t_2 \leftarrow i*10$	`r2 ← r1+2`
	$t_3 \leftarrow t_1+t_2$	`r3 ← [fp-8]`
	$t_4 \leftarrow 4*t_3$	`r4 ← r3*10`
	$t_5 \leftarrow \texttt{addr a}$	`r5 ← r4+r2`
	$t_6 \leftarrow t_5+t_4$	`r6 ← 4*r5`
	$t_7 \leftarrow *t_6$	`r7 ← fp-216`
		`f1 ← [r7+r6]`

Abstraction Levels of IRs (2)

High-Level IRs

- Representation very close to source code
- Often: Abstract syntax trees
- Variables & types used to store and represent data
- Preservation of complex control & data flow operations (esp. loops, *if-then* / *if-else* statements, *Array* accesses `[]`)
- Back-transformation of a high-level IR into source code easy

*[**S. S. Muchnick.** Advanced Compiler Design & Implementation. Morgan Kaufmann, 1997]*

Abstraction Levels of IRs (3)

Medium-Level IRs

- Three-address code: $a_1 \leftarrow a_2\ op\ a_3$;
- IR independent of source language & target processor
- Temporary variables used to store data
- Complex control & data flow operations simplified and broken down (labels & branches, pointer arithmetic)
- Control flow in form of *basic blocks*

Definition: A *basic block* $B=(I_1, ..., I_n)$ is an instruction sequence of maximal length such that

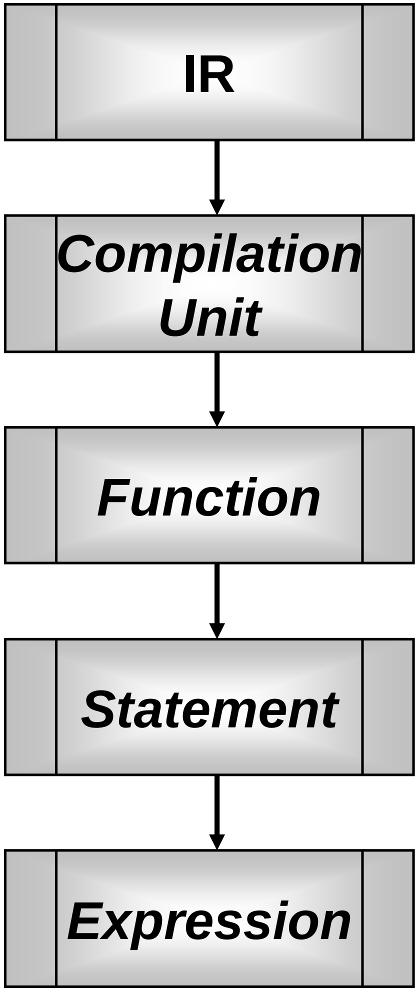
- B is entered only via its very first instruction I_1 and
- B is left only via its very last instruction I_n.

Abstraction Levels of IRs (4)

Low-Level IRs

- Representation of machine code
- Operations correspond to machine instructions
- Registers used to store data
- Transformation of a low-level IR into assembly code easy

High-Level IR: ICD-C (1)

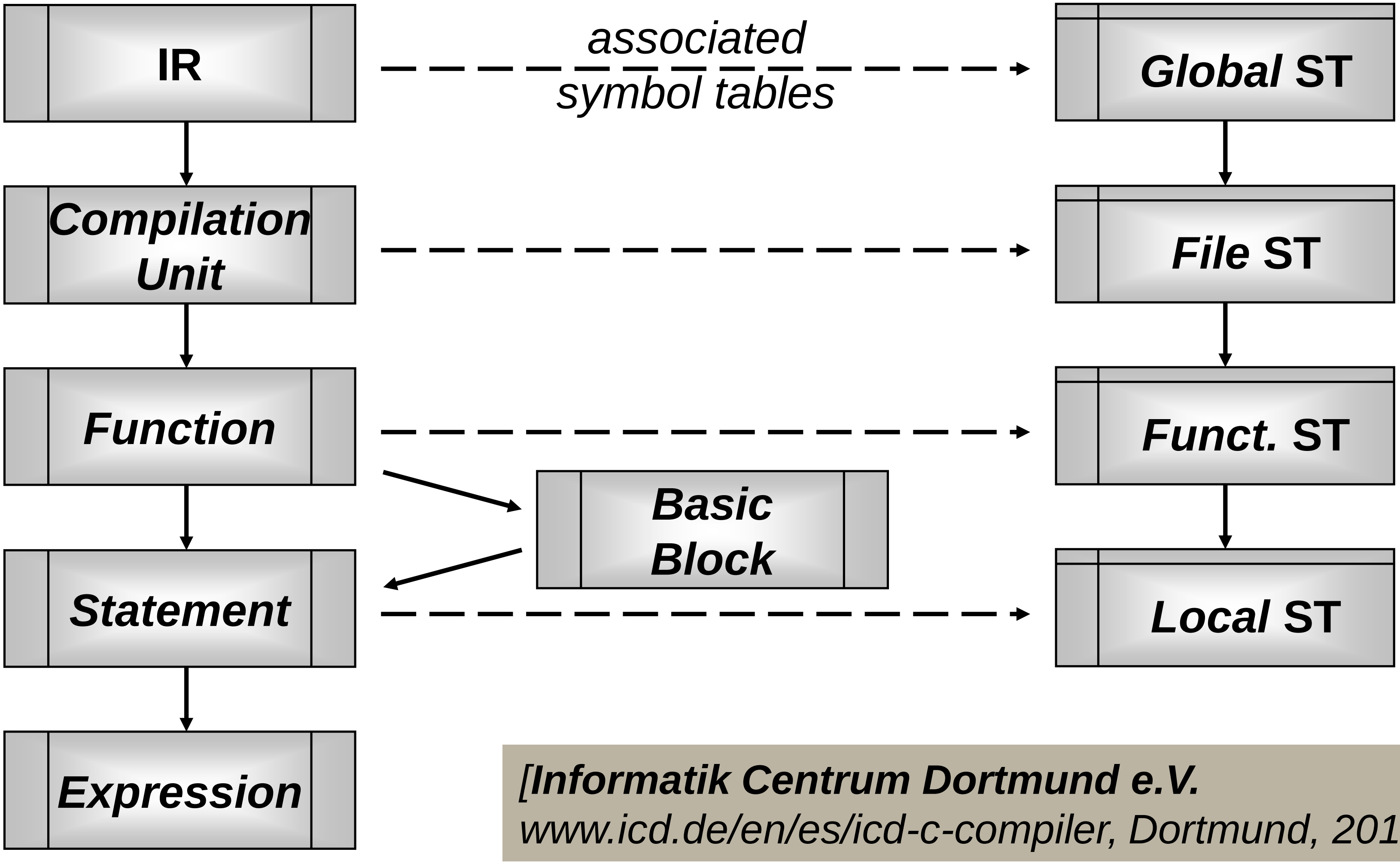


Compilation Unit:
- 1 C file when simultaneously translating several source code files

Statement:
- Loop Statements (for, do-while, while-do)
- Selection Statements (if, if-else, switch)
- Jump Statements (return, break, continue, ...)
- ...

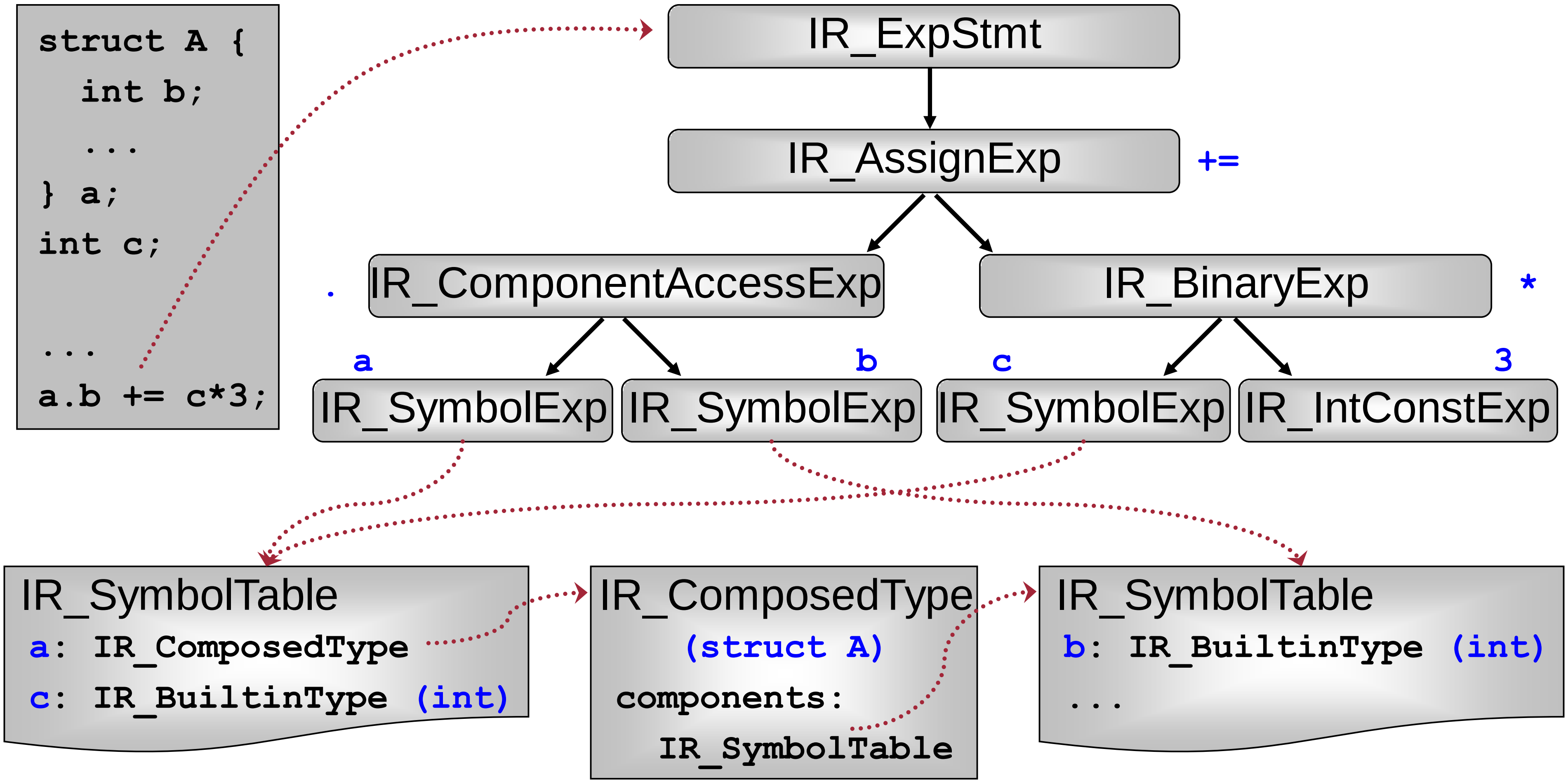
Expression:
- Binary & Unary Expressions (+, -, *, /, ...)
- Assignment operators (=, +=, -=, ...)
- Index & component accesses (*a[x]*, *a.x*, ...)
- ...

High-Level IR: ICD-C (2)



*[**Informatik Centrum Dortmund e.V.**
www.icd.de/en/es/icd-c-compiler, Dortmund, 2015]*

ICD-C: Code Example



```
struct A {
   int b;
   ...
} a;
int c;

...
a.b += c*3;
```

ICD-C: Features

– **ANSI-C Compiler Front-end:**	C89 + C99 standards GNU Inline-assembly
– **Included Analyses:**	Data flow analyses Control flow analyses Loop analyses Pointer analyses

- **Interfaces:**
  - ANSI-C dump of the IR as interface to external tools
  - Interface to code selector in compiler back-ends

- **Internal Structure:**
  - Object-oriented design (C++)

Medium-Level IR: MIR (1)

- **MIR Program:** 1 – *N* Program Units (i.e., functions)
- **Program Unit:** **`begin`** MIRInst* **`end`**
- **MIR Instructions:**
  - Quadruples: 1 operator, 3 operands (3-address code)
  - Types of instructions:
    Assignments, jumps (**`goto`**), conditions (**`if`**),
    function call & return (**`call`**, **`return`**),
    parameter passing (**`receive`**)
  - Can contain MIR expressions

Medium-Level IR: MIR (2)

- **MIR Expressions:**
  - Binary operators: `+`, `-`, `*`, `/`, `mod`, `min`, `max`
  - Comparison operators: `=`, `!=`, `<`, `<=`, `>`, `>=`
  - Shift & logical operators: `shl`, `shr`, `shra`, `and`, `or`, `xor`
  - Unary operators: `-`, `!`, `addr`, `cast`, `*`

- **Symbol Table:**
  - Contains variables and symbolic registers
  - Entries have types: `integer`, `float`, `boolean`

*[**S. S. Muchnick.** Advanced Compiler Design & Implementation. Morgan Kaufmann, 1997]*

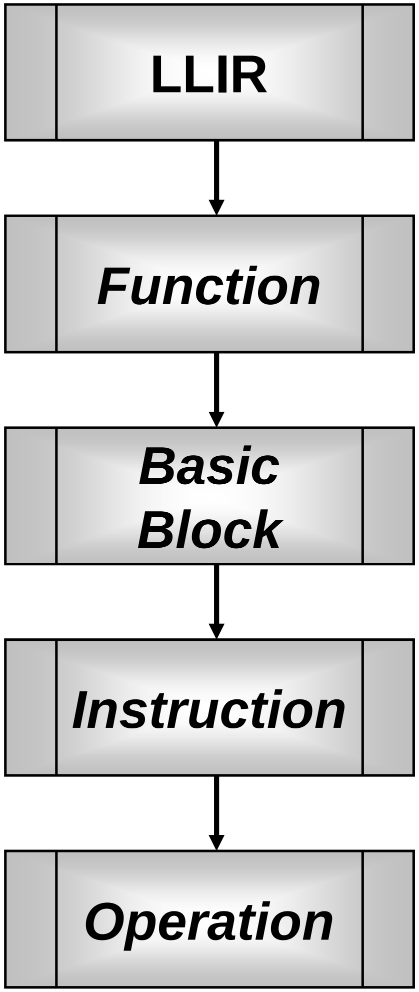
MIR: Properties

- **MIR is not a High-Level IR**
  - Closeness to source language lacking
  - High-level constructs are missing: Loops, array accesses, ...
  - Only few and mostly simple operators present

- **MIR is not a Low-Level IR**
  - Closeness to target architecture lacking: Behavior of operands is defined in machine-independent way
  - Concepts of symbol tables, variables and types not low-level
  - Abstract mechanisms for function calls, returns, and parameter passing

☞ **MIR is a Medium-Level IR.**

Low-Level IR: LLIR (1)



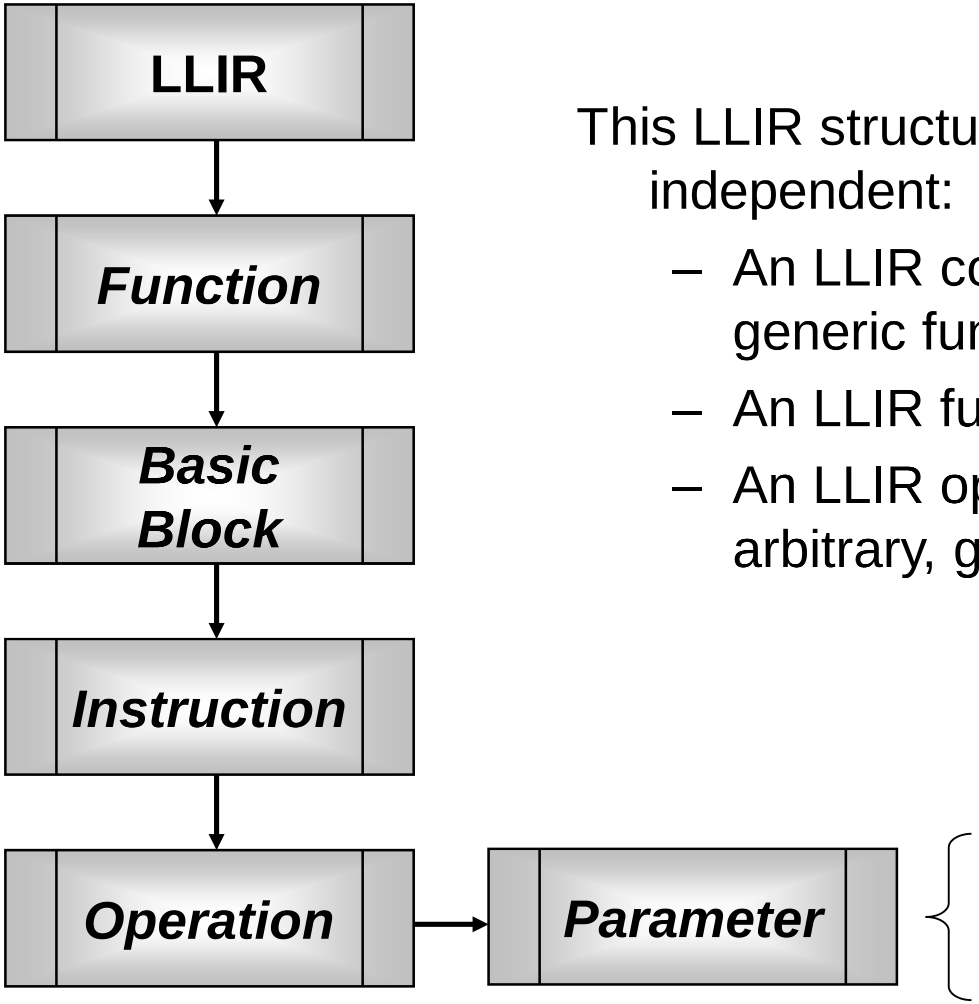
Instruction:

- Machine instruction
- Contains 1-*N* machine operations
- Operations are executed in parallel (☞ VLIW)

Operation:

- Machine operation
- Contains assembly opcode (e.g., **`ADD`**, **`MUL`**, …)
- Contains 0-*M* parameters

Low-Level IR: LLIR (2)



LLIR

Function

Basic Block

Instruction

Operation

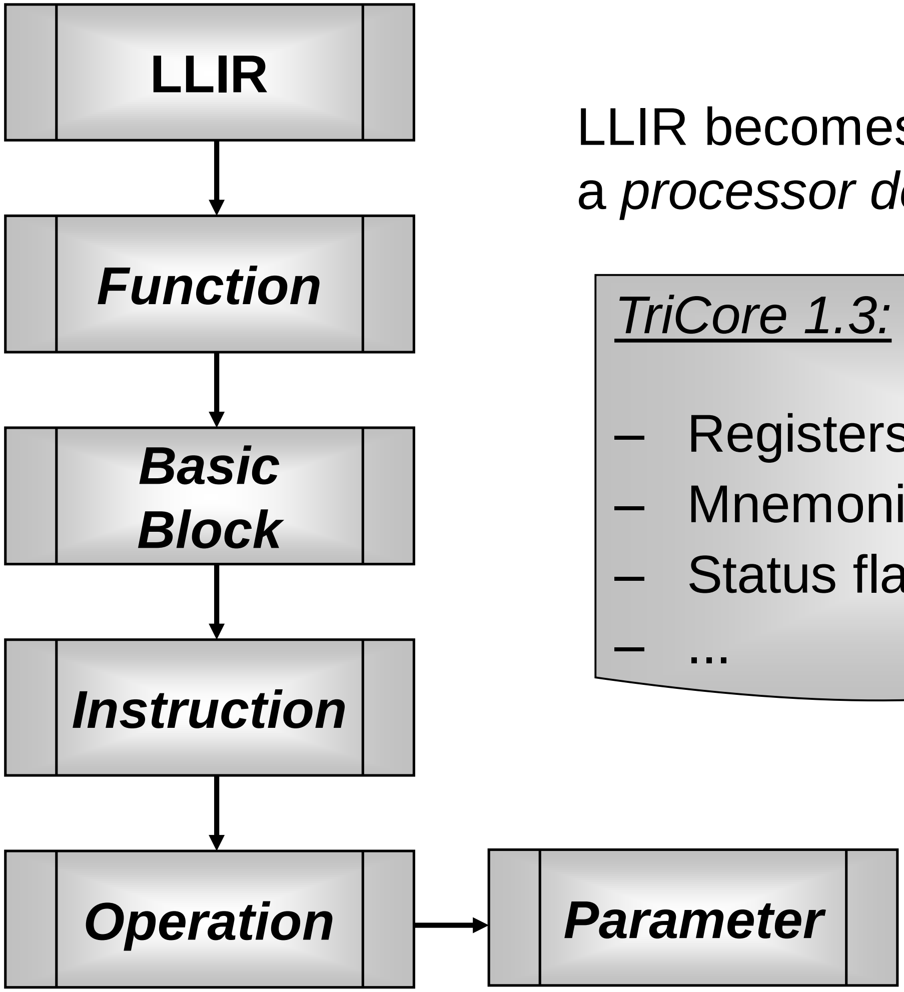
Parameter

This LLIR structure is completely processor-independent:

- An LLIR consists of some arbitrary, generic functions
- An LLIR function consists of...
- An LLIR operation consists of some arbitrary, generic parameters

- Registers
- Integer constants & labels
- Addressing modes
- ...

Low-Level IR: LLIR (3)



LLIR becomes processor-specific by providing a *processor description*:

TriCore 1.3:

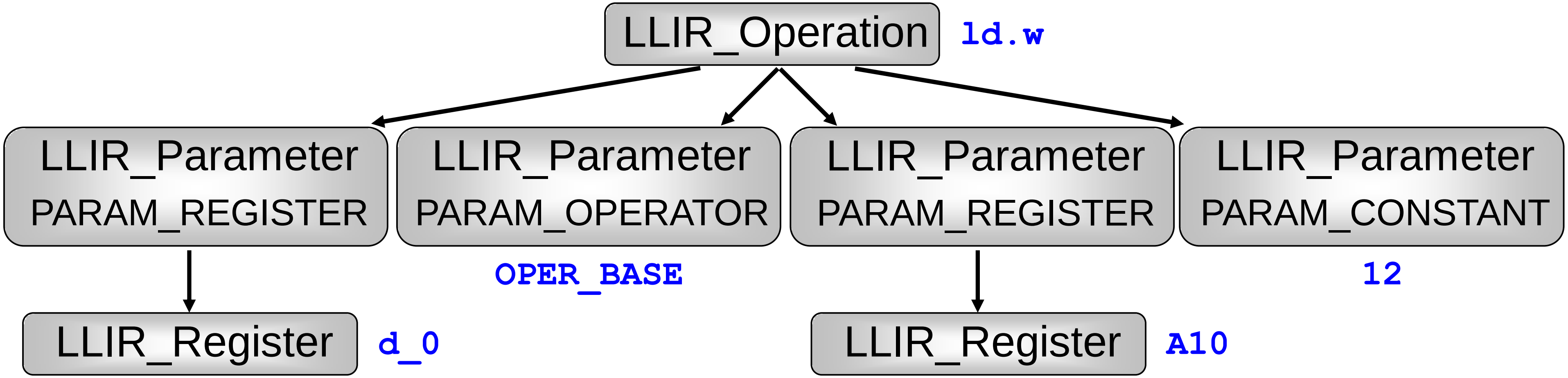
- Registers = {**`D0`**, ..., **`D15`**, **`A0`**, ..., **`A15`**}
- Mnemonics = {**`ABS`**, **`ABS.B`**, ..., **`XOR.T`**}
- Status flags = {**`C`**, **`V`**, ..., **`SAV`**}
- ...

*[**Informatik Centrum Dortmund e.V.** www.icd.de/en/es/icd-c-compiler, Dortmund, 2015]*

LLIR: Code Example *(Infineon TriCore 1.3)*

```
ld.w %d_0, [%A10] 12;
```

- Load memory contents from address `[%A10] 12` in register `d_0`
- <u>Recall:</u> Register `A10` = *Stack pointer* ☞ Physical register
- Address `[%A10] 12` = *Stack pointer* + 12 Bytes
  (so-called Base + Offset Addressing)
- TriCore features no register `d_0` ☞ Virtual data register



LLIR_Operation `ld.w`

LLIR_Parameter PARAM_REGISTER

LLIR_Register `d_0`

LLIR_Parameter PARAM_OPERATOR `OPER_BASE`

LLIR_Parameter PARAM_REGISTER

LLIR_Register `A10`

LLIR_Parameter PARAM_CONSTANT `12`

LLIR: Features

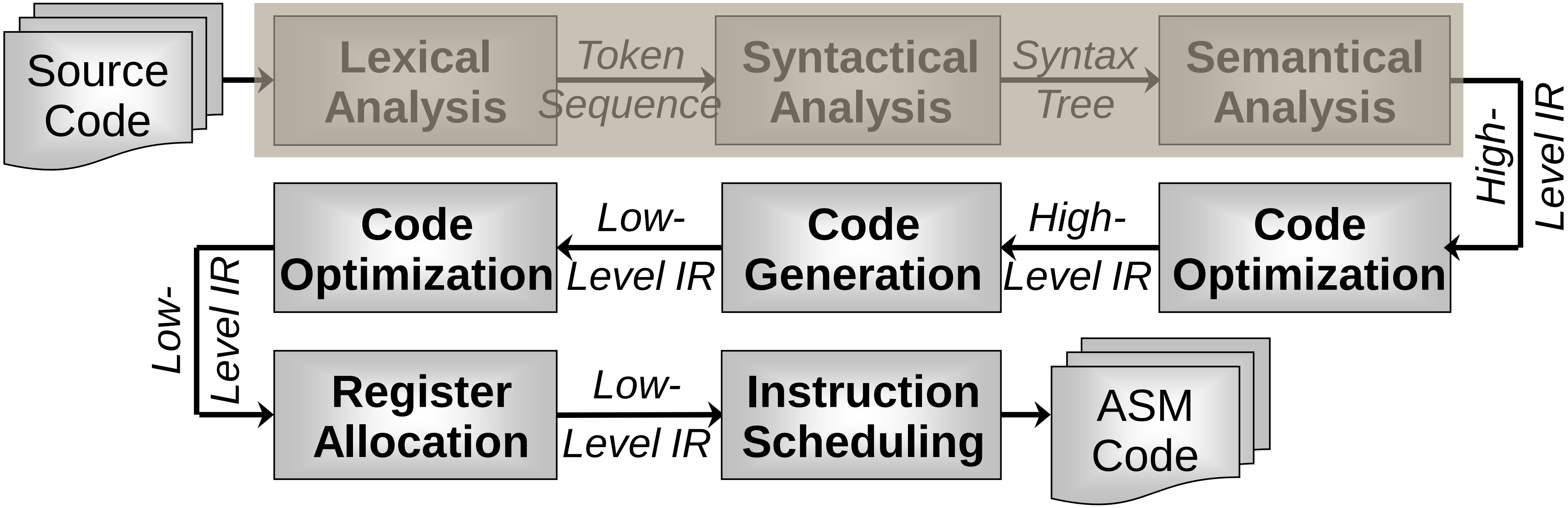
- **Retargetability:**
  - Adaptability to support various distinct processors (e.g., DSPs, VLIWs, NPUs, ...)
  - ☞ Modelling of various instruction set architectures (ISAs)
  - ☞ Modelling of various kinds of register sets
- **Included Analyses:**
  - Data flow analyses
  - Control flow analyses
- **Interfaces:**
  - Import and export of assembly files
  - Interface to code generation

Back to the Open Question...

Where should Optimizations be placed inside the Compiler?

– *Optimizations (usually) take place at the IR level within a compiler.*

☞ ***Structure of an Optimizing Compiler with 2 IRs:***

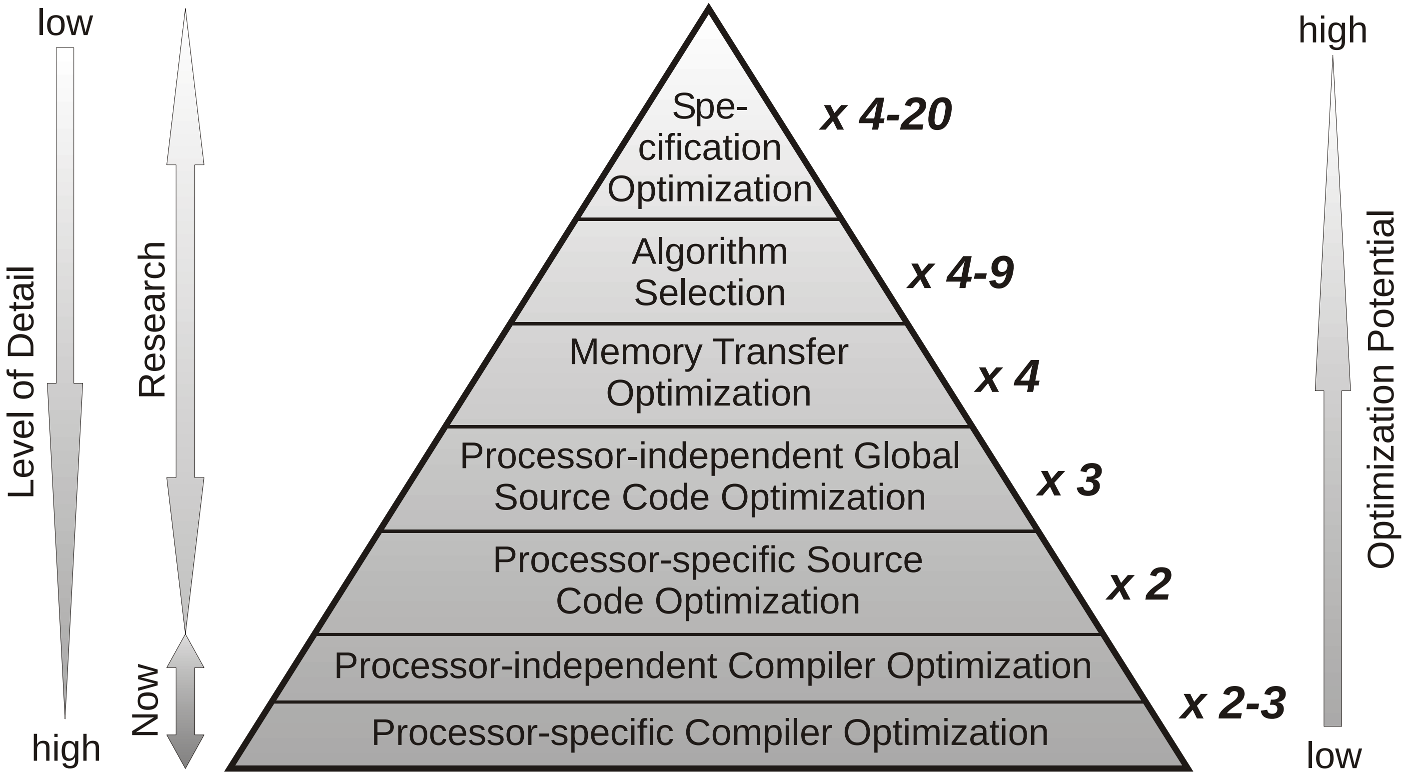


Chapter Contents

3. Internal Structure of Compilers

- Compiler Stages
  - *Front-End:* Lexical Analysis, Syntactical Analysis, Semantical Analysis
  - *Back-End:* Instruction Selection, Register Allocation, Instruction Scheduling
- Intermediate Representations
  - *High-Level*, *Medium-Level* & *Low-Level IRs*
  - Case Studies: ICD-C, MIR, LLIR
  - Structure of a Highly-Optimizing Compiler
- Optimizations & Objectives
  - Abstraction Levels of Optimizations
  - Average-Case & Worst-Case Execution Time
  - Code Size
  - Energy Consumption

Abstraction Levels of Optimizations (1)



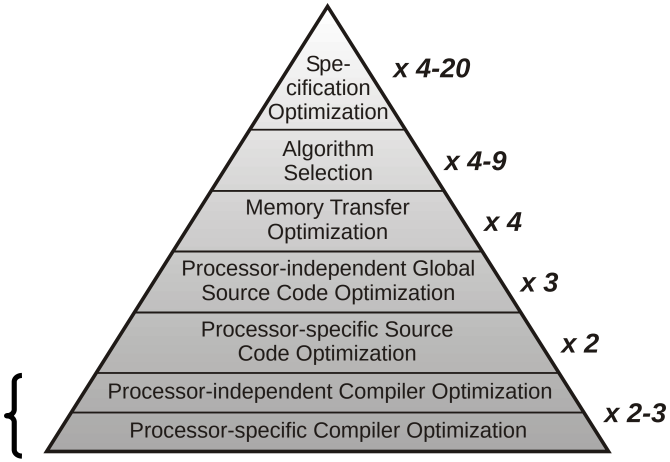
*[**H. Falk.** Source Code Optimization Techniques for Data Flow Dominated Embedded Software. Kluwer, 2004]*

Abstraction Levels of Optimizations (2)

Compiler Optimization

- Everything that is included in today's compilers
- Processor-specific: low-level
- Processor-independent: high-level
- Typical speed-ups: Factor 2 to 3 altogether

☞ *Cf. chapters 5 – 9*

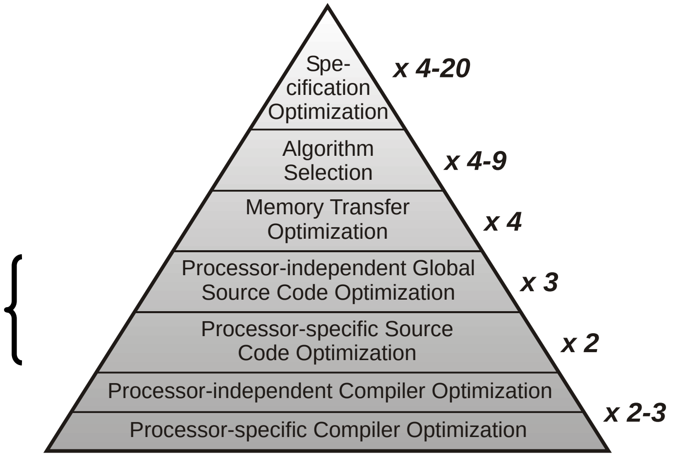


Abstraction Levels of Optimizations (3)

Source Code Optimization

- Transformation of source code such that subsequent compiler generates more efficient code
- Processor-specific: Support the compiler in mapping source to target language
- Processor-independent: Machine-independent improvement of the source code's structure
- Partly automated, partly manual
- Typical speed-ups: 2x or 3x

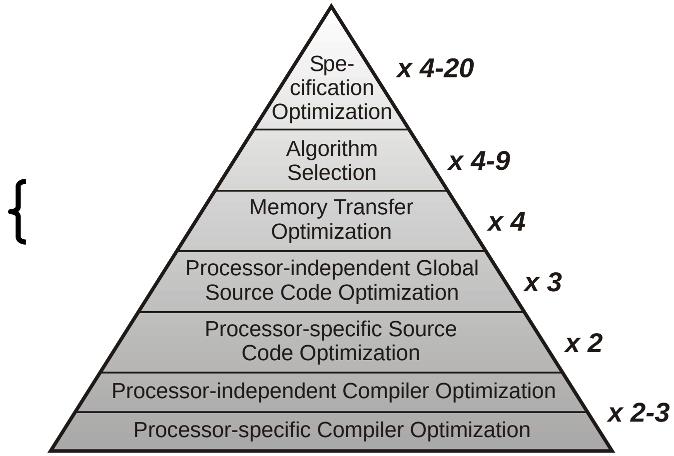
☞ *Cf. chapter 4*



Abstraction Levels of Optimizations (4)

Memory Transfer Optimization

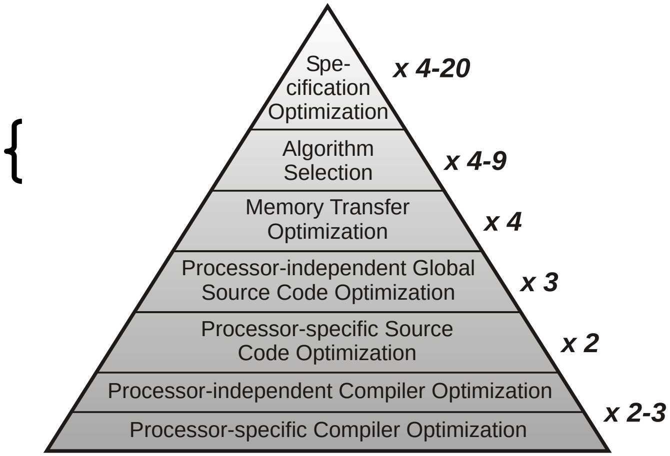
- Reduction of data and code transfers from memory to processor at a very abstract level
- E.g., restructuring of an algorithm’s data structures, reorganizing of (multi-dimensional) arrays in memory, merging or splitting of arrays
- Only manually
- Typical speed-ups: ca. 4x



Abstraction Levels of Optimizations (5)

Algorithm Selection

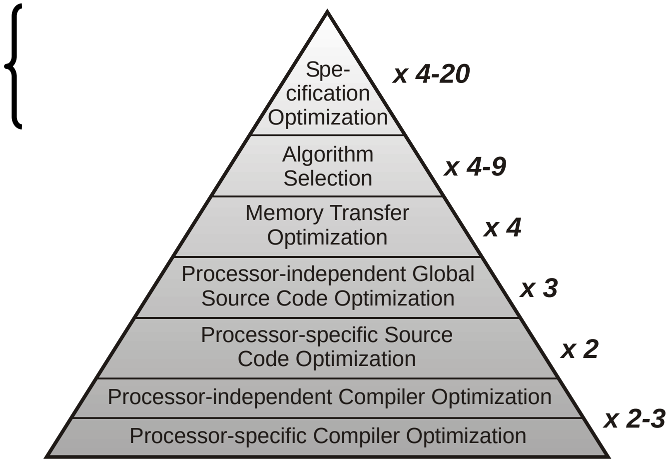
- Replacement of entire algorithms of a program by other, more efficient implementations
- E.g., Bubblesort → Quicksort
- Replacement must preserve functional behavior of the program
- Only manually
- Typical speed-ups: 4x – 9x



Abstraction Levels of Optimizations (6)

Specification Optimization

- Replacement of algorithms just as during "Algorithm Selection"
- *But:* Replacement is allowed to change the program's functional behavior
- E.g., Replacement of `double` floating-point numbers by single-precision or `integer` values; replacement of complex formulae by simpler approximations (`sin`, `cos`)
- Only manually
- Also known as Approximate Computing
- Typical speed-ups: 4x – 20x



Objective Function: (Typical) Run-Time

- **Average-Case Execution Time (ACET)**
  An ACET-optimized program shall run faster during a “typical” execution using “typical” input data.
- ***The*** **Objective Function of Optimizing Compilers per se. Strategy:**
  “Greedy”, i.e., whenever the execution of code at run-time can be saved somewhere, this is actually also done.
- **ACET-optimizing Compilers usually do not have a precise ACET Model.**
  Exact impacts of optimizations on the effective run-time are completely unknown to the compiler.

☞ **ACET optimizations are usually beneficial, but sometimes only neutral or even disadvantageous**

Example: Function Inlining

```
main() {
   ...
   a = min( b, c );
   ...
   ...min( f, g )...
}
```

```
int min( int i,
         int j ) {
  return(
    i<j ? i : j );
}
```

```
main() {
   ...
   a = b<c ? b : c;
   ...
   ...f<g ? f : g;
}
```

Potential Run-Time Reduction due to:

- Code for parameter and return value passing redundant
- Code to jump into the called function redundant
- Memory allocation at beginning of called function eventually redundant
- Potential enabling of other optimizations that otherwise fail due to function boundaries

Objective Function: Code Size

- **Generation of a minimal Amount of Code, measured in Bytes**
- **Trivial Modelling:**

  Compiler knows exactly which machine instructions it generates and how many bytes each individual instruction takes.

Often in Conflict with Run-Time Minimization: Example Inlining

- Inlining copies a function’s body to the place of the function call
- For large functions and/or many calls of a function in the code: Heavy increases in terms of code size!
- Code size-minimizing optimizing compilers:

  ☞ *Completely deactivated Function Inlining*

Objective Function: Energy Consumption (1)

- **Generation of Code that consumes minimal Electrical Energy**
- **Modelling usually includes both Processor and Memories**

Simple Energy Model for Processors:

- *Base Costs* of a machine instruction: Energy consumption of the processor during execution of this single machine instruction
- Determination of base costs (e.g., for an `ADD` instruction):

```
.L0:
  ...
  ADD d0, d1, d2;
  ADD d0, d1, d2;
  ADD d0, d1, d2;
  ...
  LOOP a5, .L0;
```

- Loop that contains the examined instruction *very often*.
- Execution on real hardware
- Energy measurement: Ampere meter
- Breakdown of result to one single `ADD`
- Repetition for the entire instruction set

Objective Function: Energy Consumption (2)

Simple Energy Model for Processors:

- *Inter-instruction Costs* between two successive machine instructions: Model activation and deactivation of Functional Units (FUs)
- Example: `ADD` executed in ALU, `MUL` in dedicated multiplier

```
.L0:
  ADD d0, d1, d2;
  MUL d3, d4, d5;
  ADD d0, d1, d2;
  MUL d3, d4, d5;
  ...
  LOOP a5, .L0;
```

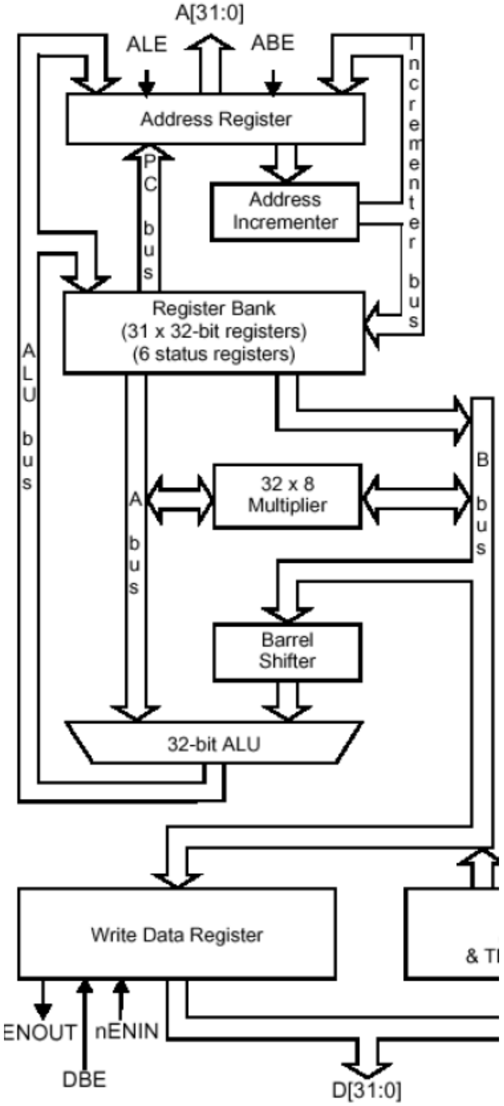
- Loop that contains the examined instruction pair *very often*.
- Execution and measurement as done for base costs
- Breakdown of result to one single pair of `ADD` and `MUL`
- Repetition for all possible combinations of FUs

Objective Function: Energy Consumption (3)

Functional Units of ARM7 Processors

- Functional units: Address incrementer, 32x8 multiplier, barrel shifter and ALU
- *Example from previous slide:* power-on ALU for **`ADD`**, perform addition.
- *Hereafter:* Power-on multiplier for **`MUL`**, charge busses to/from multiplier.
- *Finally:* Power-down multiplier after **`MUL`**, discharge busses.

☞ Power-on/-down of FUs & charging/discharging of wires costs lots of electrical energy!



*[**ARM Limited.** ARM7TDMI Technical Reference Manual. 2004]*

Objective Function: Energy Consumption (4)

Computation of the CPU Energy by Compiler

- Sum up base costs of all generated machine instructions
- Sum up inter-instruction costs of all successive instruction pairs
- Multiply by estimated execution counts per basic block

*[**V. Tiwari et al.** Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. IEEE Transactions on VLSI, December 1994]*

Computation of Memory Energy by Compiler

- Either using data sheets from manufacturers, or based on measurements
- Principle: Energy consumption per load or store memory access
- Simple for static RAMs (SRAM), difficult for Caches and dynamic RAMs (DRAM)

*[**S. Steinke et al.** An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. PATMOS Workshop, September 2001]*

Objective Function: Worst-Case Run-Time (1)

Worst-Case Execution Time (WCET):

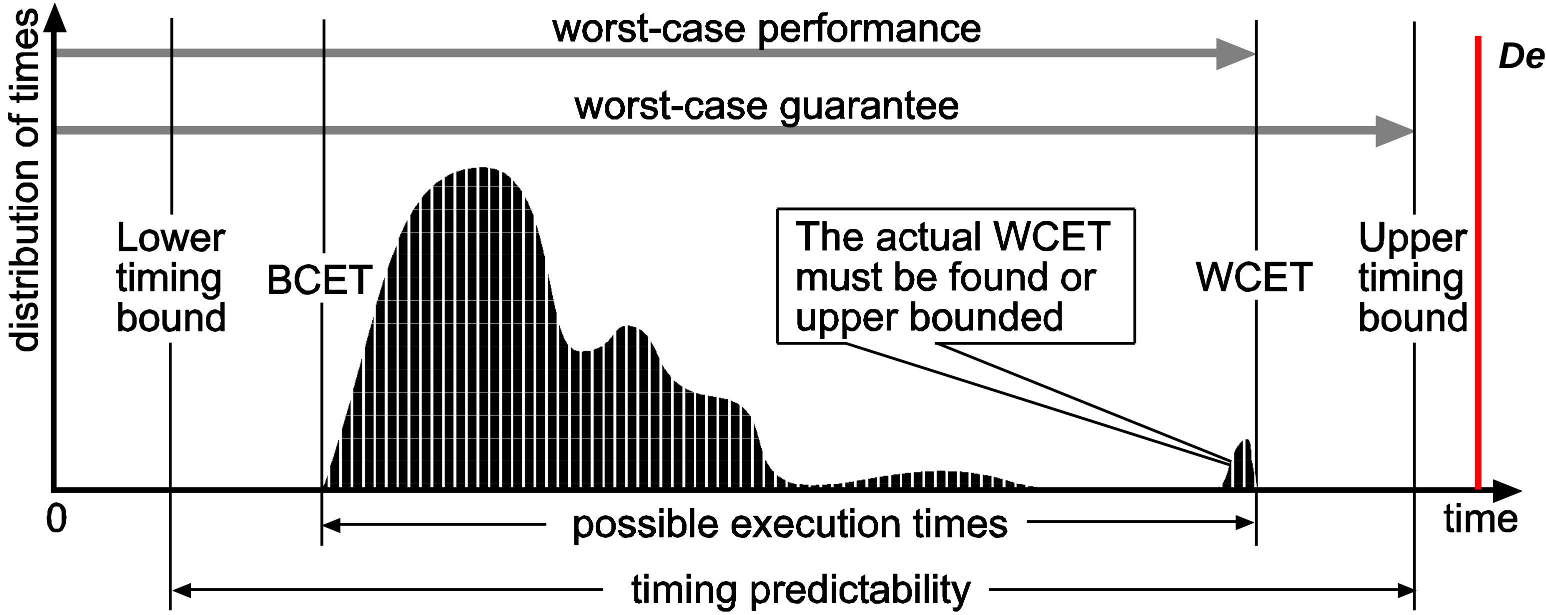
Maximal run-time of a program over all possible input data.

Problem:

Determination of a program’s WCET intractable! (Would include solving the Halting problem)

Objective Function: Worst-Case Run-Time (2)

Solution: Estimation of upper bounds of the actual (unknown) WCET



Requirements on WCET Estimates:

- Safeness: $WCET \leq WCET_{EST}$!
- Tightness: $WCET_{EST} - WCET \rightarrow$ minimal

References

Compiler Stages and IRs

- Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
  ISBN 1-55860-320-4
- Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.
  ISBN 0-521-58390-X

Abstraction Levels of Optimizations

- H. Falk. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic Publishers, 2004.
  ISBN 1-4020-2822-9

Summary (1)

Compiler Stages

- Significance of individual phases within a compiler
- Focus here on compiler back-end
- Location of optimizations within the compiler

Intermediate Representations

- Effective compiler-internal representations of code; facilitate manipulation and analysis of code
- Different abstraction levels: Close to source language; independent of source language and processor architecture; close to processor architecture

Summary (2)

Optimizations & Objective Functions

- Many classes of optimizations having high potential are not automatable
- Focus here on compiler and source code optimizations
- Average-Case Execution Time: Objective function of almost every compiler; compilers, however, to not feature an ACET timing model
- Code size: Often in contradiction with ACET
- Energy consumption: Energy models for processors (base & inter-instruction costs) and memory
- Worst-Case Execution Time: not computable; WCET estimation