TUHH
Hamburg University of Technology

Institute of Embedded Systems

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

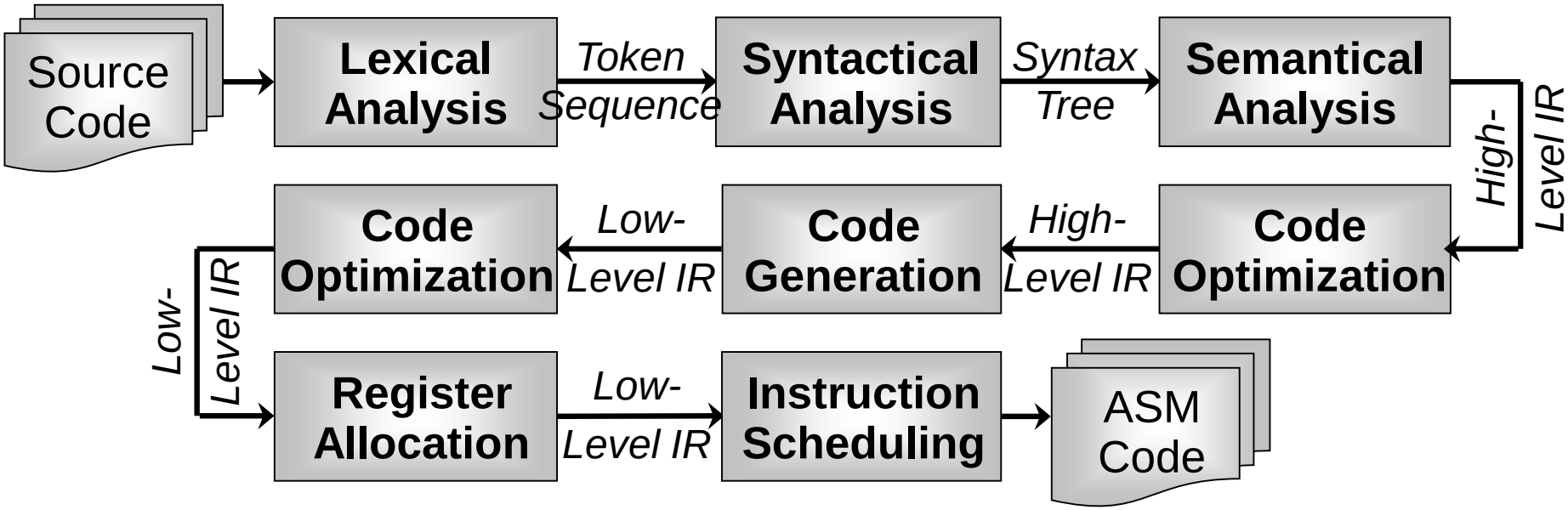
Chapter 10

Outlook

Outline

Chapter Contents

Motivation

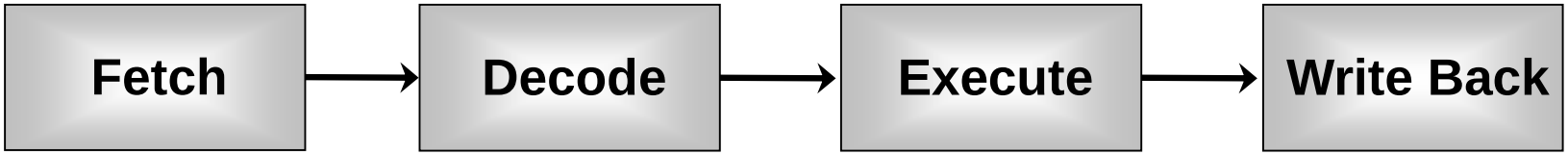


Instruction Scheduling

- Rearrangement of machine instructions in order to increase instruction-level parallelism
- Insertion of NOP operations to keep code correct
- Due to shortage of time not treated in detail here

Pipeline Processing (1)

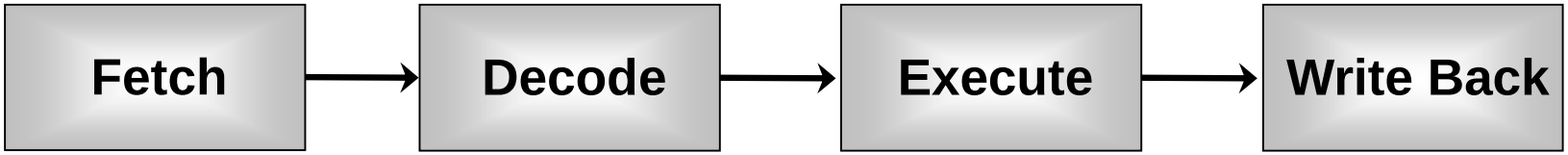
A simple Processor Pipeline



- **Fetch:** Fetches the instruction to be executed next from memory
- **Decode:** Decodes the fetched instruction and deconstructs it in opcodes, operands, addressing modes, ...
- **Execute:** Executes an instruction according to its opcodes and operands. The result of the execution is temporarily stored in an internal buffer register
- **Write Back:** Writes back the content of the internal buffer register into the processor's register file

Pipeline Processing (2)

A simple Processor Pipeline



- Each pipeline stage usually operates within one clock cycle
- At some point in time *t*, the pipeline can process four different instructions in parallel:
  Instruction i_1 in Fetch stage, i_2: Decode, i_3: Execute, i_4: Write Back
- Ideally, a fully loaded pipeline finishes the execution of one instruction in each clock cycle
- However, this ideal case can be perturbed rather easily...

Potential Perturbations of Pipeline Processing

Data Dependencies of Instructions

– An instruction i_1 producing a value in register r is in the execute stage. Another instruction i_2 that requires that register r is in the decode stage.
– Since i_1 has not yet written back its result in r, the pipeline has to be stalled before i_2 can proceed to the execute stage itself.

Branches

– A branch is taken during the execute stage so that other instructions directly following the branch have already been loaded into the fetch and decode stages of the pipeline.
– If the branch is taken, the fetch and decode stages have to be flushed and to be re-filled from the address of the branch target.

Re-Ordering of Instructions

Basic Idea

- Machine instructions can be ordered arbitrarily within a program, as long as for any pair of instructions i_1 and i_2 ...
  - ... data dependencies are respected.
    If, e.g., i_2 uses a register that i_1 defines, i_1 must not be placed after i_2 in the code.
  - ... control dependencies are respected.
    If, e.g., a program's control flow defines that an execution of i_1 always entails the execution of i_2, i_1 and i_2 must not be re-ordered such that this relationship is violated.

Instruction Scheduling inside a Compiler

Basic Idea

- For each instruction *i*, an interval [*f*, *b*] can be determined:
  By how many instructions can *i* be moved to front (*f*) / back (*b*) in the code without violating the correctness of the program?
- A scheduler inside a compiler can exploit such intervals to re-order the code such that the processor's pipeline is perturbed as little as possible
- Example **Data dependencies**: Move some other instruction i_3 between load instruction i_1 and i_2
- Example **Branches**: Move 1 or 2 instructions from which the branch *i* is not data or control dependent after *i* to fill the so-called delay slots
- Example **TriCore**: Schedule instructions such that all three pipelines are always kept busy

Chapter Contents

Motivation

Current View on a Compiler

– A target processor *P* is given fixed.
  Then, we focused on the task to construct a compiler that translates the source language into *P*'s machine language and produces highest code quality.

– Most compiler components required detailed knowledge about the architecture of *P*. This knowledge is hard-coded into the different phases / optimizations of the compiler.

– What if we now need a compiler for *P'* instead of one for *P*?

  ☞ Re-implement all compiler components from scratch for *P'* ?

Retargetability

Retargetability

Is the ability to adapt a compiler to another target architecture.

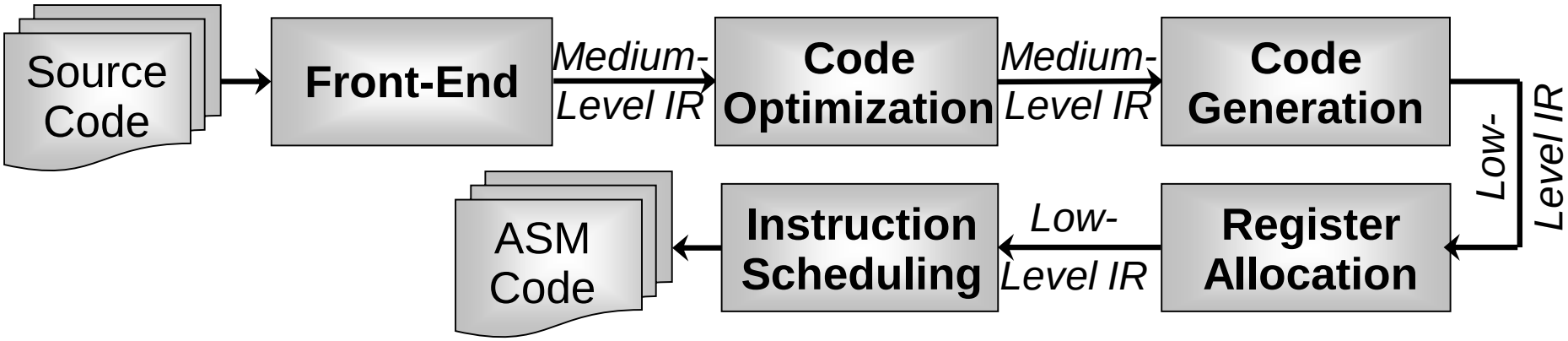
Current Compiler Structure

The structure of compilers considered so far during this course *(☞ cf. slide 5)* is poorly retargetable. In order to retarget such a compiler,

– code generator,
– all LIR optimizations,
– register allocator and
– scheduler

need to be re-implemented in large part.

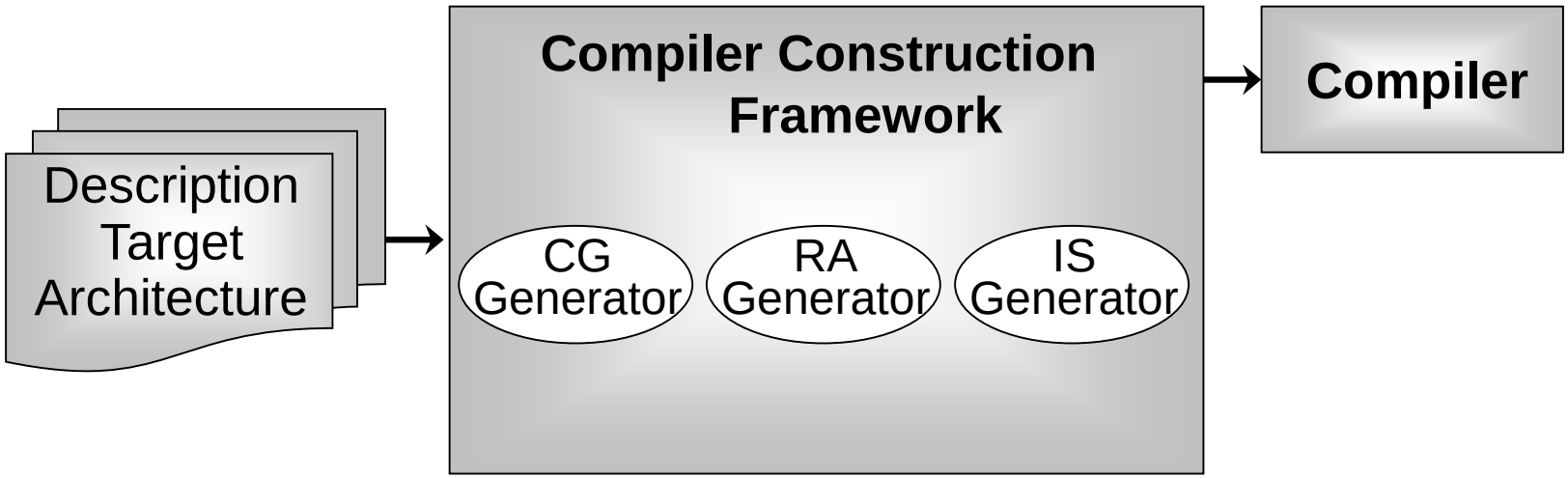
A better retargetable Compiler



Properties

- Instead of an HIR, such a compiler features an MIR and an LIR
- All optimizations take place at MIR level and are thus processor-independent and do not need to be retargeted
- Highly specific optimizations that exploit complex processor features are difficult to integrate, if at all.
- Overhead to retarget the back-end remains

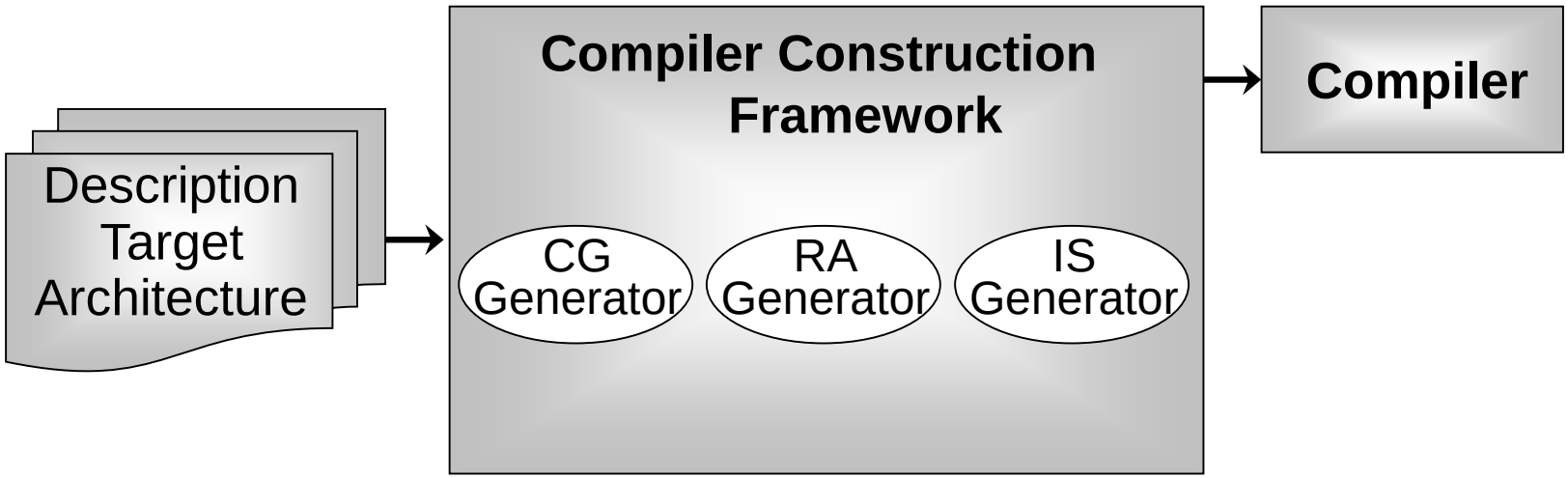
Automatic Compiler Construction (1)



Compiler Construction

- Starting point is a description of the target processor
  - either in a hardware description language (e.g., VHDL) or
  - in a system description language (e.g., SystemC or Lisa)
- From this description, a compiler back-end is generated automatically.

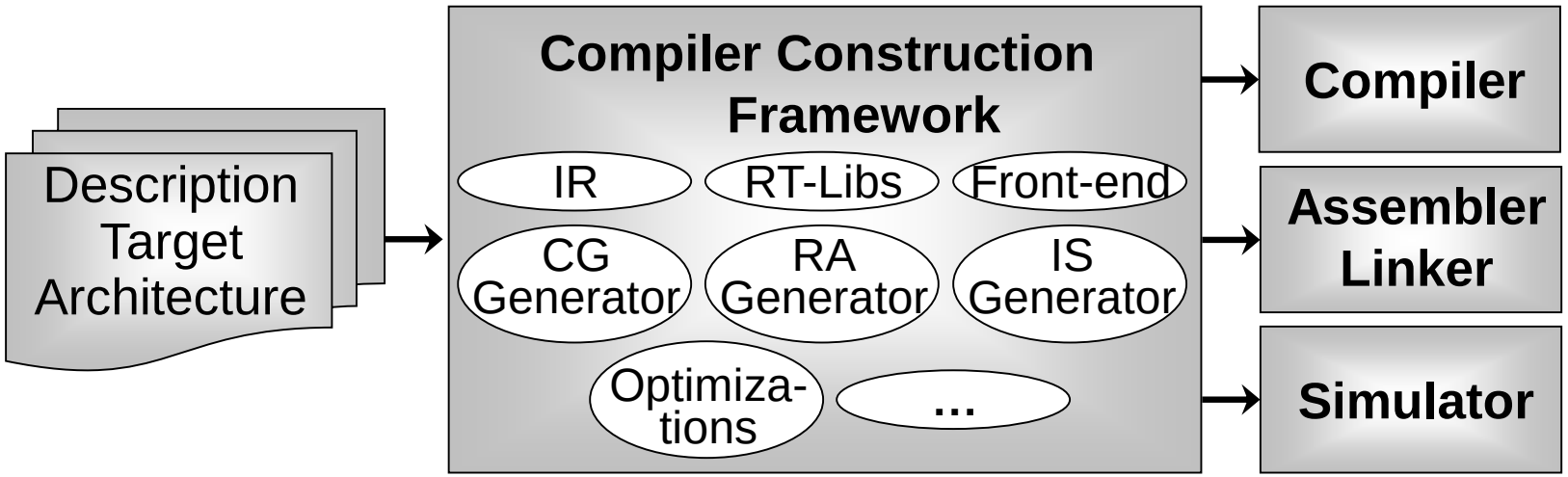
Automatic Compiler Construction (2)



Compiler Construction *(ctd.)*

- From the processor description, the entire instruction set of the processor can be extracted. Using this information, a complete tree grammar for a code generator generator is produced.
- Analogously, properties of the register file are extracted to generate a register allocator. *(Same for scheduler)*

Automatic Compiler Construction (3)



Compiler Construction *(ctd.)*

- These generated back-end modules are coupled with „ready-made“ standard components that provide the central IR, the front-end, IR optimizations and run-time libraries.
- Besides a compiler, such a framework can additionally produce assemblers, linkers and cycle-true simulators.