1 Introduction and Overview

Contents of this module

1. Course organization
   - Planned course schedule
   - Personnel
   - Course resources
   - Exercises
   - Exam
2. Course overview
   - Goals of this course
   - Structure of a compiler
   - Three phases of compilation

Week	Date	Topic	Project Milestones
1	22.10.2018	Introduction and Overview	
2	29.10.2018	Scanners 1: From Regular Expression to Scanner	
3	05.11.2018	Scanners 2: Implementation Techniques	
4	12.11.2018	Parsers 1: Top-Down Parsing	M1: C++ and Scanner
5	19.11.2018	Parsers 2: Bottom-Up Parsing	
6	26.11.2018	Elaboration: Type Checking	
7	03.12.2018	Intermediate Representations	M2: Parser
8	10.12.2018	Code Shape 1: Procedures	
9	17.12.2018	Code Shape 2: Operators, Complex Values, and Control Flow	
10	07.01.2019	Instruction Selection	M3: Intermediate Representation
11	14.01.2019	Instruction Scheduling	
12	21.01.2019	Register Allocation	
13	28.01.2019	Optimization 1: Introduction	
14	04.02.2019	Optimization 2: Data-Flow Analysis	M4: Code Generation
15	11.02.2019	Optimization 3: Scalar Optimizations	

Personnel

- Michael Grossniklaus

  Office PZ 806

  E-mail `michael.grossniklaus@uni-konstanz.de`

Michael Grossniklaus

Curriculum vitæ

1996–2008 ETH Zürich
- Dipl. Inf.-Ing. ETH (MSc) in Informatik
- Dr. sc. techn. (PhD) in Informatik

2008–2010 Politecnico di Milano

2010–2012 Portland State University

2012–2013 TU Wien

2013– Universität Konstanz

Course resources

- Registration
  - **ZEuS** (`https://zeus.uni-konstanz.de`)
    $\longrightarrow$ course-related e-mails and notifications
  - **Ilias** (`https://ilias.uni-konstanz.de`)
    $\longrightarrow$ access to course materials
  - **StudIS** (`https://studis.uni-konstanz.de`)
    $\longrightarrow$ admittance to the exam
- Literature
  - lecture slides can be downloaded from Ilias
  - further readings will also be published in Ilias

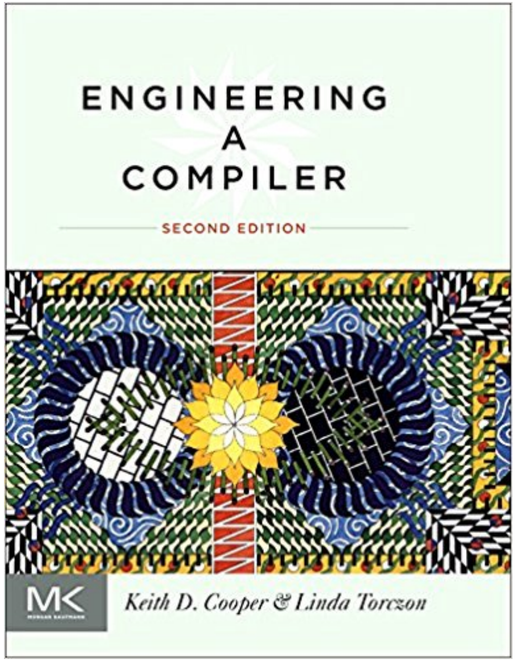
Literature

This course is closely based on the following textbook.

- **Keith D. Cooper** and **Linda Torczon**
  Egineering a Compiler
  Morgan Kaufmann, 2nd Edition (February 21, 2011)

Other recommended textbooks include the following.

- **Niklaus Wirth**
  Compiler Construction
  Addison-Wesley (June 1, 1996)
- **Alfred V. Aho**, **Monica S. Lam**, **Jeffrey D. Ullman**, and **Ravi Sethi**
  Compilers: Principles, Techniques, and Tools
  Pearson, 2nd Edition (January 11, 2011)



Prerequisites

In order to attend this course, students should meet the following prerequisites.

- **Concepts of Programming** (INF-12070 or equivalent)
  - formal semantics, λ-calculus, type theory, evaluation strategies, *etc*.
- **Computer Systems** (INF-11880 or equivalent)
  - computer architecture, machine language, assembler, operating systems, *etc*.
- **Theory of Computing** (INF-2210 or equivalent)
  - formal language theory, Chomsky hierarchy, grammars, automatons, *etc*.
- **Systems Programming** (INF-11740 or equivalent)
  - the accompanying project requires advanced programming skills
  - students should have the ability to program in C++, *i.e.*, read, understand, design, and write high-quality code
- **Key Competences** (INF-10175 or equivalent)
  - Subversion, Git, LaTeX, *etc*.

Rooms and Dates

- Lectures
  - Monday 10:00 am – 11:30 am Room P 1138
- Tutorials
  - Tuesday 10:00 am – 11:30 am Room P 602

Project

- Description
  - implement a simple non-optimizing, *i.e.*, two-phase compiler
  - C++ programming language
- Milestones
  - **M1**: Master C++ and understand provided scanner classes 12.11.2018
  - **M2**: Implementation of a recursive-descent parser 03.12.2018
  - **M3**: Produce three-address code as intermediate representation 07.01.2019
  - **M4**: Generate assembler code that can be executed 04.02.2019
- Grade
  - project grade is 50% of final grade

Exam

- Written exam
  - 90-minutes closed-book exam
  - one hand-written double-sided sheet of A4 paper with notes is permitted
- Content
  - lecture and lecture slides
  - textbook
- Schedule
  - first exam XX. XX. XXXX
  - second exam XX. XX. XXXX
- Grade
  - exam grade is 50% of final grade

Learning Objectives

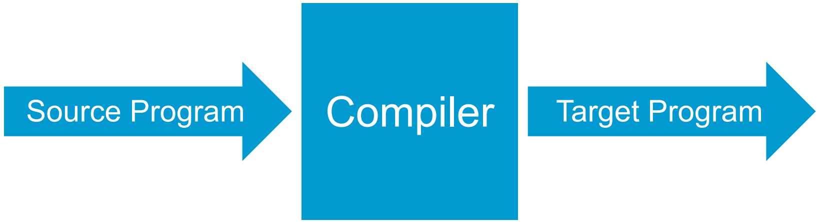
Course Goal

This course teaches the design and implementation of a **compiler**. Building a compiler is a substantial exercise in software engineering.

A good compiler contains a **microcosm** of computer science

- finite automata and push-down automata (scanning and parsing)
- greedy algorithms (register allocation)
- dynamic programming (instruction selection)
- heuristic search techniques (instruction scheduling)
- fixed-point algorithms (data-flow analysis)
- graph algorithms (dead-code elimination)

What Is a Compiler?



A compiler takes as input a source program written in some language and produces as its output an **equivalent** target program.

- typical source languages are Java, Haskell, C, C++, *etc*.
- typical target language is the instruction set of some processor

A compiler that targets programming languages rather than the instruction set of a compiler is often called a **source-to-source translators**.

Compiler vs. Interpreter

An interpreter takes as input an executable specification and produces as output the **result** of executing this specification.

- interpreters and compilers perform many of the same tasks
- some languages use both compilation and interpretation

Example The Java compiler (`javac`) translates source programs into bytecode that is executed by the Java Virtual Machine (JVM), an interpreter for bytecode.

Virtual Machines and Just-in-Time Compilation

A virtual machine is a **simulator** for a processor. It is an **interpreter** for the instruction set of that processor.

Advantages of virtual machines

- platform independence, *i.e.*, portable executable
- compact bytecode representation (for *e.g.*, downloads, embedded devices, *etc.*)

Many virtual machines include a so-called **just-in-time compiler** (JIT)

- just-in-time compiler executes at run-time
- translates heavily used bytecode sequences into native code

Fundamental Principles of Compilation

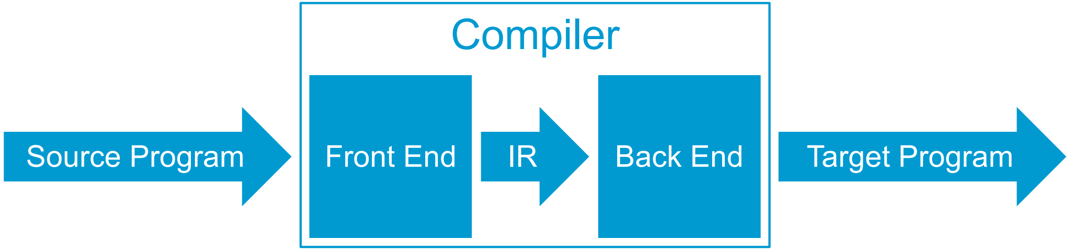
The compiler must preserve the meaning of the program being compiled.

- preserve correctness by faithfully implementing the "meaning" of the input program
- "social contract" between the compiler writer and compiler user

The compiler must improve the input program in some discernible way.

- improve input program by making it directly executable on some target machine
- produce code that is, in some measure, better than the input program

Two-Phase Compiler



Compilation is often decomposed in two major components: a **front end** and a **back end**.

Front end

- analyze the source-language program and ensure that it is well-formed
- map input code to intermediate representation (IR)

Back end

- map IR program into instruction set and finite resources of target machine
- can assume that the ir contains no syntactic or semantic errors

Intermediate Representation

A compiler uses a set of data structures to represent the code that it processes. That form is called an **intermediate representation**, or IR.

Lifecycle

- front end uses IR form to encode its knowledge of source program
- compiler can make multiple passes over the IR form and record relevant details
- back end emits target program from IR form

Note The compiler will use **different** intermediate representations as compilation progresses. But at each point in compilation, it will have **one** definitive representation for the code it translates.

Retargeting

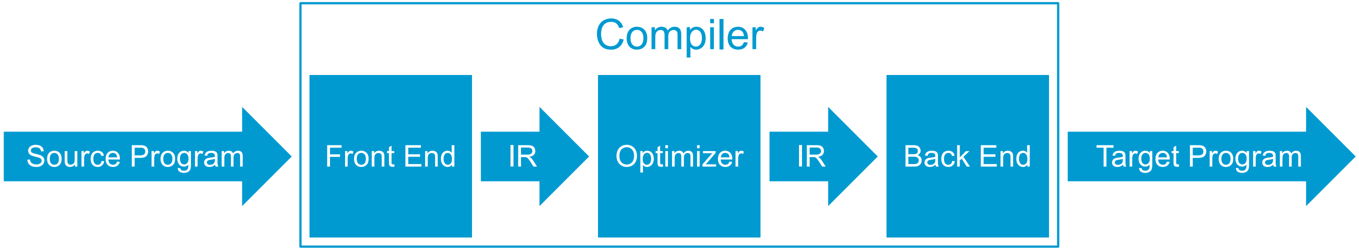
The taks of changing the compiler to generate code for a new processor is often called **retargeting** the compiler.

Decomposition of the compiler into front end and back end facilitates retargeting

- **multiple back ends, one front end**: compile same language for many processors
- **multiple front ends, one back end**: compile many languages for one processor

Note These scenarios assume that one IR works for several source-target pairs. In practice, however, both language-specific and machine-specific details usually find their way into the IR.

Three-Phase Compiler



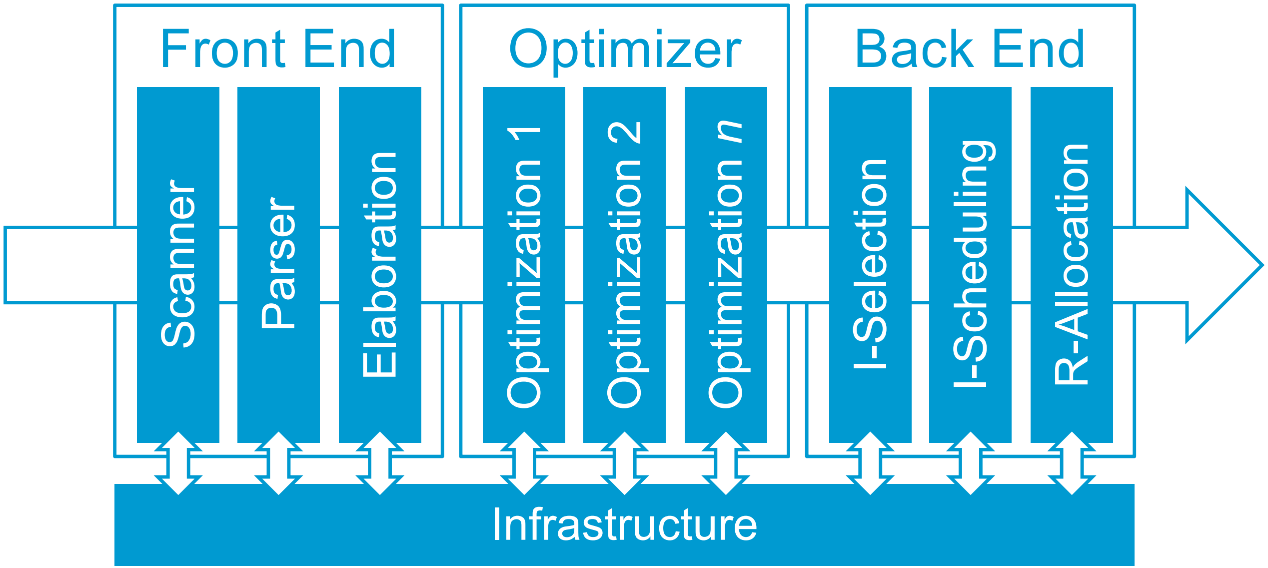
By using the IR as an interface, an **optimizer** can be inserted with minimal disruption to the front end and back end.

- optimizer analyzes and transforms IR program in order to improve it
- according to our definition, optimizer itself is a compiler

An optimizing compiler will **almost always fail** to produce optimal code

- optimization techniques are complex, interrelated, and sometimes even at odds
- a good optimizing compiler can improve the code relative to unoptimized version

Typical Structure of a Compiler



Overview of Translation

In the remainder of this lecture, we examine each component of a compiler in more detail.

- **Front end**: Scanner, Parser, Elaboration
- **Optimizer**
- **Back end**: Instruction Selection, Instruction Scheduling, Register Allocation

Running Example

As a running example, we use the translation of the following expression into executable code:

$$v_1 \leftarrow v_1 \times 2 \times v_2 \times v_3 \times v_4,$$

where v_i are variables, $\leftarrow$ denotes the assignment, and $\times$ is the operator for multiplication.

Scanner

The **scanner** takes a stream of characters and converts it to a stream of tokens.

A **token** is a $\langle t, v\rangle$-tuple

- t is the token's type
- v is the token's value

Running Example

A scanner would tokenize the string `"v1 <- v1 * 2 * v2 * v3 * v4"` to the following stream of tokens:

$$\langle \text{var}, \texttt{"v1"}\rangle, \langle \text{op}, \leftarrow\rangle, \langle \text{var}, \texttt{"v1"}\rangle, \langle \text{op}, \times\rangle, \langle \text{num}, 2\rangle, \langle \text{op}, \times\rangle, \langle \text{var}, \texttt{"v2"}\rangle,$$
$$\langle \text{op}, \times\rangle, \langle \text{var}, \texttt{"v3"}\rangle, \langle \text{op}, \times\rangle, \langle \text{var}, \texttt{"v4"})$$

Parser

The **parser** checks the syntax of the source program by matching the token stream against the rules that specify the grammar of the input language.

Running Example

Expressions as the one used in our running example can be described by the following grammatical rules, or **productions**.

$$
\begin{array}{lllll}
1 & \textit{Assignment} & \rightarrow & \langle \text{var} \rangle \ \langle \text{op}, \leftarrow \rangle \ \textit{Expression} \\
2 & \textit{Expression} & \rightarrow & \textit{Factor} \ \{ \ \langle \text{op}, \times \rangle \ \textit{Factor} \ \} \\
3 & \textit{Factor} & \rightarrow & \langle \text{var} \rangle \\
4 & \textit{Factor} & \rightarrow & \langle \text{num} \rangle
\end{array}
$$

Parser

Parsing is the process of automatically finding a **derivation** from the grammar rules to the source program.

Running Example

The following sequence is a possible derivation for our example expression.

–	*Assignment*
1	$\langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \leftarrow \rangle\ \textit{Expression}$
2	$\langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \leftarrow \rangle\ \textit{Factor}\ \{\ \langle \text{op}, \times \rangle\ \textit{Factor}\ \}$
3	$\langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \leftarrow \rangle\ \langle \text{var}, \texttt{"v1"} \rangle\ \{\ \langle \text{op}, \times \rangle\ \textit{Factor}\ \}$
{}	$\langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \leftarrow \rangle\ \langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \times \rangle\ \textit{Factor}\ \{\ \langle \text{op}, \times \rangle\ \textit{Factor}\ \}$
4	$\langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \leftarrow \rangle\ \langle \text{var}, \texttt{"v1"} \rangle\ \langle \text{op}, \times \rangle\ \langle \text{num}, 2 \rangle\ \{\ \langle \text{op}, \times \rangle\ \textit{Factor}\ \}$
...	

Elaboration

A grammatically correct expression can still be meaningless. After checking the **syntax** of the source program, a compiler therefore also needs to check its **semantics**.

A compiler builds mathematical models that detect specific kinds of inconsistency

- type inconsistencies
- number inconsistencies: array dimensions, procedure declaration vs. call, *etc*.
- ...

Running Example

The expession

$$v_1 \leftarrow v_1 \times 2 \times v_2 \times v_3 \times v_4$$

is well formed, but if v_2 is a string or if v_4 is a Boolean value, the expression still might be invalid.

Intermediate Representations

The final task of the front end is to generate an **IR form** of the source program. Depending on the intended use, different types of intermediate representations are appropriate.

- **graphical** intermediate representation, *e.g.*, for data-flow analysis
- **linear** intermediate representation, *e.g.*, for instruction selection

For **every** source-language construct, the compiler needs a strategy for how to implement that construct in IR form.

Intermediate Representations

Running Example

A possible **linear** intermediate representation for our example expression would be as follows.

$$
\begin{array}{rcl}
t_0 & \leftarrow & v_1 \times 2 \\
t_1 & \leftarrow & t_0 \times v_2 \\
t_2 & \leftarrow & t_1 \times v_3 \\
t_3 & \leftarrow & t_2 \times v_4 \\
a & \leftarrow & t_3
\end{array}
$$

Note This particular type of intermediate representation is known as **three-address code**.

Optimizer

While the front end handles each statement of the source program in isolation, the **optimizer** uses contextual information to transform the code into a more efficient form.

What is "efficient"?

- execution time
- binary size
- energy consumption
- . . .

Optimization techniques typically target one of these efficiency criteria. For example, the loop-unrolling technique improves execution time but increases the size of the binary.

Optimizer

Most optimizations consist of **analysis** and **transformation**.

Analysis

- determines where the compiler can safely and profitably apply the technique
- **data-flow analysis** reasons, at compile-time, about data flow at run-time
- **dependence analysis** creates execution-order constraints between statements

Transformation

- rewrite code into a more efficient form (based on results of analysis)
- many transformations have been invented to improve time or space requirements
- for example, move loop-invariant code to less frequently executed locations

Optimizer

Running Example

In the following code, our example expression occurs inside a loop (left). By leveraging this context, the optimizer can improve the code (right).

$v_2 \leftarrow \ldots$
$v_3 \leftarrow \ldots$
$v_1 \leftarrow 1$
$\texttt{for}\ i = 1\ \texttt{to}\ n$
  $\texttt{read}\ v_4$
  $v_1 \leftarrow v_1 \times 2 \times v_2 \times v_3 \times v_4$
$\texttt{end}$

$v_2 \leftarrow \ldots$
$v_3 \leftarrow \ldots$
$v_1 \leftarrow 1$
$t \leftarrow 2 \times v_2 \times v_3$
$\texttt{for}\ i = 1\ \texttt{to}\ n$
  $\texttt{read}\ v_4$
  $v_1 \leftarrow v_1 \times v_4 \times t$
$\texttt{end}$

Instruction Selection

Instruction selection maps each IR operation, in its context, into one or more target machine operations.

Running Example

loadAI	r_{arp}, @v1	$\Rightarrow$	r_1	// load v_1 with offset @v1 from address r_{arp}
loadI	2	$\Rightarrow$	r_5	// load constant 2 into r_5
loadAI	r_{arp}, @v2	$\Rightarrow$	r_2	// load b
loadAI	r_{arp}, @v3	$\Rightarrow$	r_3	// load c
loadAI	r_{arp}, @v4	$\Rightarrow$	r_4	// load d
mult	r_1, r_5	$\Rightarrow$	r_1	// $r_1 \leftarrow v_1 \times 2$
mult	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2) \times v_2$
mult	r_1, r_3	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2) \times v_3$
mult	r_1, r_4	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2 \times v_3) \times v_4$
storeAI	r_1	$\Rightarrow$	r_{arp}, @v1	// write r_1 back to v_1

Instruction Selection

Virtual Registers

- compiler assumes unlimited supply of registers and gives them symbolic names
- register allocator will map these virtual registers to the actual registers

Special Operations

- instruction selector can take advantage of special operations on the target machine
- replace $\texttt{mult}\ r_1,\ r_5 \Rightarrow r_1$ with $\texttt{multI}\ r_1,\ 2 \Rightarrow r_1$
- replace $\texttt{multI}\ r_1,\ 2 \Rightarrow r_1$ with $\texttt{add}\ r_1,\ r_1 \Rightarrow r_1$

Register Allocation

Instruction selection may create demand for **more** registers than the target-machine can support.

Register Allocator

- maps those virtual registers onto actual target-machine registers
- decides which values should reside in the target-machine registers
- rewrites the code to reflect its decisions

Register Allocation

Running Example

This following code sequence only uses **three** registers instead of six.

`loadAI`	r_{arp}, @v1	$\Rightarrow$	r_1	// load v_1
`add`	r_1, r_1	$\Rightarrow$	r_1	// $r_1 \leftarrow v_1 \times 2$
`loadAI`	r_{arp}, @v2	$\Rightarrow$	r_2	// load v_2
`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2) \times v_2$
`loadAI`	r_{arp}, @v3	$\Rightarrow$	r_2	// load v_2
`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2) \times v_3$
`loadAI`	r_{arp}, @v4	$\Rightarrow$	r_2	// load v_4
`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2 \times v_3) \times v_4$
`storeAI`	r_1	$\Rightarrow$	r_{arp}, @a	// write r_1 back to v_1

Register Allocation

Exercise

Minimizing registers may be counterproductive. Based on our running example, can you think of situations where this might be the case?

Register Allocation

Exercise

Minimizing registers may be counterproductive. Based on our running example, can you think of situations where this might be the case?

- if any of the named values, v_1, v_2, v_3, or v_4, are already in registers, the code should reference those registers directly
- if some nearby expression also computed $a \times 2$, it might be better to preserve that value in a register than to recompute it later

Instruction Scheduling

The process of **reordering** the instruction sequence to reduce execution time is called instruction scheduling.

Not all instructions execute in one clock cycle of the processor

- memory access operations can take tens or hundreds of cycles
- even some arithmetic operations, *e.g.*, division, take several cycles

Processors can typically initiate new instructions while a **long-latency** operation executes.

- if results of long-latency operation are not referenced, execution proceeds normally
- otherwise, the processor delays the instruction that tries to read these results

Instruction Scheduling

Running Example

Assume, for the moment, that a `loadAI` or `storeAI` operation requires three cycles, a `mult` requires two cycles, and all other operations require one cycle.

Start	End					
1	3	loadAI	r_{arp}, @v1	$\Rightarrow$	r_1	// load v_1
4	4	add	r_1, r_1	$\Rightarrow$	r_1	// $r_1 \leftarrow v_1 \times 2$
5	7	loadAI	r_{arp}, @v2	$\Rightarrow$	r_2	// load v_2
8	9	mult	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2) \times v_2$
10	12	loadAI	r_{arp}, @v3	$\Rightarrow$	r_2	// load v_2
13	14	mult	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2) \times v_3$
15	17	loadAI	r_{arp}, @v4	$\Rightarrow$	r_2	// load v_4
18	19	mult	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2 \times v_3) \times v_4$
20	22	storeAI	r_1	$\Rightarrow$	r_{arp}, @a	// write r_1 back to v_1

Instruction Scheduling

Running Example

A good scheduler might produce the following sequence that only requires 13 cycles to execute.

Start	End					
1	3	`loadAI`	r_{arp}, @v1	$\Rightarrow$	r_1	// load v_1
2	4	`loadAI`	r_{arp}, @v2	$\Rightarrow$	r_2	// load v_2
3	5	`loadAI`	r_{arp}, @v3	$\Rightarrow$	r_2	// load v_2
4	4	`add`	r_1, r_1	$\Rightarrow$	r_1	// $r_1 \leftarrow v_1 \times 2$
5	6	`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2) \times v_2$
6	8	`loadAI`	r_{arp}, @v4	$\Rightarrow$	r_2	// load v_4
7	8	`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2) \times v_3$
9	10	`mult`	r_1, r_2	$\Rightarrow$	r_1	// $r_1 \leftarrow (v_1 \times 2 \times v_2 \times v_3) \times v_4$
11	13	`storeAI`	r_1	$\Rightarrow$	r_{arp}, @a	// write r_1 back to v_1